Extending SPL

The last example introduced 4 new statements into SPL. Let us make this introduction formal.

- The statement S = **Critical** in process P_i contributes the transition relation disjunct

$$\pi_i = pre(S) \ \wedge \ \pi'_i = post(S) \ \wedge \ pres(V - \{\pi_i\})$$

  and the justice requirement $J_S: \ \pi_i \neq pre(S)$, implying that the critical section always terminates.

- The statement S = **Non-critical** in process P_i contributes the transition relation disjunct

$$\pi_i = pre(S) \ \wedge \ \pi'_i = post(S) \ \wedge \ pres(V - \{\pi_i\})$$

  and no justice requirement, implying that the non-critical section may choose not to terminate.

Course G22.3033.007 Lecture 2 R. Dewar and A. Pnueli

Sempahore Statements

- The statement S = **request** y in process P_i contributes the transition relation disjunct

$$\pi_i = pre(S) \ \wedge \ y > 0 \ \wedge \ y' = y - 1 \ \wedge \ \pi'_i = post(S) \ \wedge \ pres(V - \{\pi_i, y\})$$

  no justice requirement, and the compassion requirement

$$C_S: \ (\pi_i = pre(S) \ \wedge \ y > 0, \quad \pi_i \neq pre(S)),$$

  implying that, if this statement is infinitely often enabled, it will be eventually executed.

- The statement S = **release** y in process P_i contributes the transition relation disjunct

$$\pi_i = pre(S) \ \wedge \ y' = y + 1 \ \wedge \ \pi'_i = post(S) \ \wedge \ pres(V - \{\pi_i, y\})$$

  and the justice requirement $J_S: \ \pi_i \neq pre(S)$.

Demonstrating what can be achieved by Formal Verification

We will illustrate how formal verification (when it works) can aid us in the development of reliable programs.

Consider the following program TRY-1 which attempts to solve the mutual exclusion problem by shared variables:

$$\textbf{local}\quad y_1, y_2 \quad : \textbf{boolean where } y_1 = y_2 = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1 : \textbf{Non-Critical} \\ \ell_2 : \textbf{await } \neg y_2 \\ \ell_3 : y_1 := 1 \\ \ell_4 : \textbf{Critical} \\ \ell_5 : y_1 := 0 \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1 : \textbf{Non-Critical} \\ m_2 : \textbf{await } \neg y_1 \\ m_3 : y_2 := 1 \\ m_4 : \textbf{Critical} \\ m_5 : y_2 := 0 \end{array}\right] \end{array}\right]$$

Variables y_1 and y_2 signify whether processes P_1 and P_2 are interested in entering their critical sections.

Program Properties: Invariance

A state s is said to be reachable by program P (P-reachable) if it appears in some computation of P.

Let p be an assertion (state formula). Assertion p is called an invariant of program P if every P-reachable state satisfies p.

For program TRY-1, the property of mutual exclusion can be specified by requiring that the assertion

$$\varphi_{exclusion} : \quad \neg(at\_\ell_4 \;\wedge\; at\_m_4)$$

be an invariant of TRY-1. This implies that no execution of TRY-1 can ever get to a state in which both processes execute their critical sections at the same time.

Invoking TLV

To check whether assertion $\varphi_{exclusion}$ is an invariant of program TRY-1, we invoke the model checking tool TLV, a model checker based on the SMV tool developed in CMU by Ken McMillan and Ed Clarke.

We prepare two input files: `try1.spl` which contains the SPL representation of TRY-1, and `try1.pf`, a proof script file. The proof script file contains some printing commands, definition of the assertion $\varphi_{exclusion}$ and a command to check its invariance over the program.

We will present each of these input files.

File `try1.spl`

```
local y1 : bool where y1 = F;
      y2 : bool where y2 = F;

P1:: [l_0: loop forever do [
        l_1: noncritical;
        l_2: await !y2;
        l_3: y1 := T;
        l_4: critical;
        l_5: y1 := F       ]
     ]
||
P2:: [m_0: loop forever do [
        m_1: noncritical;
        m_2: await !y1;
        m_3: y2 := T;
        m_4: critical;
        m_5: y2 := F       ]
     ]
```

File try1.pf

```
Print "Check for Mutual Exclusion\n";

Let exclusion := !(at_l_4 & at_m_4);
Call Invariance(exclusion);
```

The call to procedure `Invariance` invokes the process which checks whether any reachable state violates the assertion `exclusion`.

Course G22.3033.007 Lecture 2 R. Dewar and A. Pnueli

Results of Verifying TRY-1

The results of model-checking TRY-1 are

```
>> Load "try1.pf";
Check for Mutual Exclusion
Model checking Invariance Property
*** Property is NOT VALID ***
Counter-Example Follows:
---- State no. 1 =
pi1 = l_0,   pi2 = m_0,   y1 = 0,      y2 = 0,
---- State no. 2 =
pi1 = l_1,   pi2 = m_0,   y1 = 0,      y2 = 0,
---- State no. 3 =
pi1 = l_1,   pi2 = m_1,   y1 = 0,      y2 = 0,
---- State no. 4 =
pi1 = l_1,   pi2 = m_2,   y1 = 0,      y2 = 0,
---- State no. 5 =
pi1 = l_1,   pi2 = m_3,   y1 = 0,      y2 = 0,
---- State no. 6 =
pi1 = l_2,   pi2 = m_3,   y1 = 0,      y2 = 0,
---- State no. 7 =
pi1 = l_3,   pi2 = m_3,   y1 = 0,      y2 = 0,
---- State no. 8 =
pi1 = l_3,   pi2 = m_4,   y1 = 0,      y2 = 1,
---- State no. 9 =
pi1 = l_4,   pi2 = m_4,   y1 = 1,      y2 = 1,
```

Expressed in a More Readable Form

$$\textbf{local}\quad y_1, y_2 \quad : \textbf{boolean where } y_1 = y_2 = 0$$

$$P_1 :: \left[\begin{array}{l}\ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l}\ell_1 : \textbf{Non-Critical} \\ \ell_2 : \textbf{await } \neg y_2 \\ \ell_3 : y_1 := 1 \\ \ell_4 : \textbf{Critical} \\ \ell_5 : y_1 := 0\end{array}\right]\end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l}m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l}m_1 : \textbf{Non-Critical} \\ m_2 : \textbf{await } \neg y_1 \\ m_3 : y_2 := 1 \\ m_4 : \textbf{Critical} \\ m_5 : y_2 := 0\end{array}\right]\end{array}\right]$$

The counter example is:

$$\langle \ell_0,\ m_0,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_1,\ m_0,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_1,\ m_1,\ y_1 : 0,\ y_2 : 0\rangle,$$
$$\langle \ell_1,\ m_2,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_1,\ m_3,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_2,\ m_3,\ y_1 : 0,\ y_2 : 0\rangle,$$
$$\langle \ell_3,\ m_3,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_3,\ m_4,\ y_1 : 0,\ y_2 : 1\rangle, \langle \ell_4,\ m_4,\ y_1 : 1,\ y_2 : 1\rangle$$

reaching the state $\langle \ell_4,\ m_4,\ y_1 : 1,\ y_2 : 1\rangle$ which violates mutual exclusion!

Obviously, the problem is that the processes test each other's y value first and only later set their own y.

Second Attempt: Set first and Test Later

The following program TRY-1 interchange the order of testing and setting:

$$\textbf{local}\quad y_1, y_2 \quad : \textbf{boolean where } y_1 = y_2 = 0$$

$$P_1 :: \left[\begin{array}{l}\ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l}\ell_1 : \textbf{Non-Critical} \\ \ell_2 : y_1 := 1 \\ \ell_3 : \textbf{await } \neg y_2 \\ \ell_4 : \textbf{Critical} \\ \ell_5 : y_1 := 0\end{array}\right]\end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l}m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l}m_1 : \textbf{Non-Critical} \\ m_2 : y_2 := 1 \\ m_3 : \textbf{await } \neg y_1 \\ m_4 : \textbf{Critical} \\ m_5 : y_2 := 0\end{array}\right]\end{array}\right]$$

Let us see whether the program is now correct.

Program Properties: Absence of Deadlock

A state s is said to be a deadlock state if no process can perform any action. In our FDS model, the idling transition is always enabled. Therefore, we define s to be a deadlock state if it has no $\mathcal{D}$-successor different from itself.

Mathematically, we can characterize all deadlock states by the assertion

$$\delta : \quad \neg \exists V' \neq V : \rho(V, V')$$

and then check for the invariance of the assertion $\neg\delta$.

To check for the interesting properties of program TRY-2, we prepare the following script file:

```
Print "Check for Mutual Exclusion\n";
Let exclusion := !(at_l_4 & at_m_4);
Call Invariance(exclusion);
Run check_deadlock;
```


Model Checking TRY-2

We obtain the following results:

```
>> Load "try2.pf";
Check for Mutual Exclusion
Model checking Invariance Property
*** Property is VALID ***
 Check for the absence of Deadlock.
Model checking Invariance Property
*** Property is NOT VALID ***
Counter-Example Follows:
---- State no. 1 =
pi1 = l_0,    pi2 = m_0,    y1 = 0,    y2 = 0,
---- State no. 2 =
pi1 = l_1,    pi2 = m_0,    y1 = 0,    y2 = 0,
---- State no. 3 =
pi1 = l_1,    pi2 = m_1,    y1 = 0,    y2 = 0,
---- State no. 4 =
pi1 = l_1,    pi2 = m_2,    y1 = 0,    y2 = 0,
---- State no. 5 =
pi1 = l_1,    pi2 = m_3,    y1 = 0,    y2 = 1,
---- State no. 6 =
pi1 = l_2,    pi2 = m_3,    y1 = 0,    y2 = 1,
---- State no. 7 =
pi1 = l_3,    pi2 = m_3,    y1 = 1,    y2 = 1,
```

In a More Readable Form

$$\textbf{local} \quad y_1, y_2 \quad : \textbf{boolean where } y_1 = y_2 = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1 : \text{Non-Critical} \\ \ell_2 : y_1 := 1 \\ \ell_3 : \textbf{await } \neg y_2 \\ \ell_4 : \text{Critical} \\ \ell_5 : y_1 := 0 \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1 : \text{Non-Critical} \\ m_2 : y_2 := 1 \\ m_3 : \textbf{await } \neg y_1 \\ m_4 : \text{Critical} \\ m_5 : y_2 := 0 \end{array}\right] \end{array}\right]$$

The counter example is:

$\langle \ell_0,\ m_0,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_1,\ m_0,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_1,\ m_1,\ y_1 : 0,\ y_2 : 0\rangle,$
$\langle \ell_1,\ m_2,\ y_1 : 0,\ y_2 : 0\rangle, \langle \ell_1,\ m_3,\ y_1 : 0,\ y_2 : 1\rangle, \langle \ell_2,\ m_3,\ y_1 : 0,\ y_2 : 1\rangle,$
$\langle \ell_3,\ m_3,\ y_1 : 1,\ y_2 : 1\rangle$

reaching the deadlock state $\langle \ell_3,\ m_3,\ y_1 : 1,\ y_2 : 1\rangle$!

Try a Different Approach

The following program TRY-3 uses a variable *turn* to indicate which process has the higher priority.

$$\textbf{local} \quad turn \quad : [1..2] \textbf{ where } turn = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1 : \text{Non-Critical} \\ \ell_2 : \textbf{await } turn = 1 \\ \ell_3 : \text{Critical} \\ \ell_4 : turn := 2 \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1 : \text{Non-Critical} \\ m_2 : \textbf{await } turn = 2 \\ m_3 : \text{Critical} \\ m_4 : turn := 1 \end{array}\right] \end{array}\right]$$

Program Properties: Response

This property refers to two assertions p and q. Written $p \leadsto q$, it means

> Every occurrence of a p-state must be followed by an occurence of a q-state

The response construct can be used to specify the property of accessibility. For example, the response property

$$at\_\ell_2 \;\leadsto\; at\_\ell_3$$

requires for program TRY-3 that every visit to ℓ_2 must be followed by a visit to ℓ_3.

To model check this property, we prepare the following file try3.pf:

```
Print "Check for Mutual Exclusion\n";
Let exclusion := !(at_l_3 & at_m_3);
Call Invariance(exclusion);
Run check_deadlock;
Print "\n Check Accessibility for P1\n";
Call Temp_Entail(at_l_2,at_l_3);
Print "\n Check Accessibility for P2\n";
Call Temp_Entail(at_m_2,at_m_3);
```


Model Checking TRY-3

We obtain the following results:

```
>> Load "try3.pf";
Check for Mutual Exclusion
Model checking Invariance Property
*** Property is VALID ***
 Check for the absence of Deadlock.
Model checking Invariance Property
*** Property is VALID ***
 Check Accessibility for P1
Model checking...
*** Property is NOT VALID ***
Counter-Example Follows:
---- State no. 1 : pi1 = l_0,    pi2 = m_0,    turn = 1,
---- State no. 2 : pi1 = l_1,    pi2 = m_0,    turn = 1,
---- State no. 3 : pi1 = l_2,    pi2 = m_0,    turn = 1,
---- State no. 4 : pi1 = l_3,    pi2 = m_0,    turn = 1,
---- State no. 5 : pi1 = l_4,    pi2 = m_0,    turn = 1,
---- State no. 6 : pi1 = l_0,    pi2 = m_0,    turn = 2,
---- State no. 7 : pi1 = l_1,    pi2 = m_0,    turn = 2,
---- State no. 8 : pi1 = l_2,    pi2 = m_0,    turn = 2,

Loop back to state 8
```

In a More Readable Form

$$\textbf{local} \quad turn \quad : [1..2] \ \textbf{where}\ turn = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{Non-Critical} \\ \ell_2 : \textbf{await}\ turn = 1 \\ \ell_3 : \textbf{Critical} \\ \ell_4 : turn := 2 \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} m_1 : \textbf{Non-Critical} \\ m_2 : \textbf{await}\ turn = 2 \\ m_3 : \textbf{Critical} \\ m_4 : turn := 1 \end{array}\right] \end{array}\right]$$

The counter example is:

$$\begin{array}{lll} \langle \ell_0,\ m_0,\ turn : 1\rangle, & \langle \ell_1,\ m_0,\ turn : 1\rangle, & \langle \ell_2,\ m_0,\ turn : 1\rangle \\ \langle \ell_3,\ m_0,\ turn : 1\rangle, & \langle \ell_4,\ m_0,\ turn : 1\rangle, & \langle \ell_0,\ m_0,\ turn : 2\rangle \\ \langle \ell_1,\ m_0,\ turn : 2\rangle, & \langle \ell_2,\ m_0,\ turn : 2\rangle & \end{array}$$

Finally a good program for Mutual Exclusion

Following is a good shared variables solution to the mutual exclusion problem.

Peterson's for 2 Processes:

$$\begin{array}{lll} \textbf{local} & y_1, y_2 & : \textbf{boolean where}\ y_1 = y_2 = 0 \\ & s & : \{1,2\}\ \textbf{where}\ s = 1 \end{array}$$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{Non-Critical} \\ \ell_2 : (y_1, s) := (1, 1) \\ \ell_3 : \textbf{await}\ y_2 = 0 \ \lor \ s \neq 1 \\ \ell_4 : \textbf{Critical} \\ \ell_5 : y_1 := 0 \end{array}\right] \end{array}\right] \quad \| \quad \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} m_1 : \textbf{Non-Critical} \\ m_2 : (y_2, s) := (1, 2) \\ m_3 : \textbf{await}\ y_1 = 0 \ \lor \ s \neq 2 \\ m_4 : \textbf{Critical} \\ m_5 : y_2 := 0 \end{array}\right] \end{array}\right]$$

$$- \ P_1 \ - \qquad\qquad\qquad - \ P_2 \ -$$

Variables y_1 and y_2 signify whether processes P_1 and P_2 are interested in entering their critical sections. Variable s serves as a tie-breaker. It always contains the signature of the last process to enter the waiting location (ℓ_3, m_3). Model checking this program, we find that it satisfies the three properties of (invariance of) mutual exclusion, absence of deadlock, and accessibility.

Dealing with Atomicity

The standard translation from SPL to the FDS representation, translate each statement into a single atomic transition. Since FDS transitions are executed by interleaving, one may wonder how faithful is this translation to real parallel execution.

Consider the following example:

$$\textbf{local} \quad y \quad : \textbf{integer where } y = 0$$
$$\begin{bmatrix} \ell_0 : & y := y + 1 \\ \ell_1 : & \end{bmatrix} \quad \| \quad \begin{bmatrix} m_0 : & y := y - 1 \\ m_1 : & \end{bmatrix}$$

All interleaving executions of this program terminate with the final value of $y = 0$. However, a real parallel execution of this program may terminate with final results of $y \in \{-1, 0, +1\}$.

Recall that the translation of such a program into machine language instructions may translate the assignment $y := y + 1$ into an instruction sequence such as $reg_1 := y;\ reg_1 := reg_1 + 1;\ y := reg_1$, where reg_1 is a register local to the left process. Thus, the machine program which is finally executed is equivalent to:

$$\textbf{local} \quad y, reg_1, reg_2 \quad : \textbf{integer where } y = 0$$
$$\begin{bmatrix} \ell_0 : & reg_1 := y \\ \ell_1 : & reg_1 := reg_1 + 1 \\ \ell_2 : & y := reg_1 \\ \ell_3 : & \end{bmatrix} \quad \| \quad \begin{bmatrix} m_0 : & reg_2 := y \\ m_1 : & reg_2 := reg_2 - 1 \\ m_2 : & y := reg_2 \\ m_3 : & \end{bmatrix}$$

Dealing with Atomicity – Continued

The Machine Program

$$\textbf{local} \quad y, r_1, r_2 \quad : \textbf{integer where } y = 0$$
$$\begin{bmatrix} \ell_0 : & r_1 := y \\ \ell_1 : & r_1 := r_1 + 1 \\ \ell_2 : & y := r_1 \\ \ell_3 : & \end{bmatrix} \quad \| \quad \begin{bmatrix} m_0 : & r_2 := y \\ m_1 : & r_2 := r_2 - 1 \\ m_2 : & y := r_2 \\ m_3 : & \end{bmatrix}$$

can yield the final results $y \in \{-1, 0, +1\}$, as can seen by the following 3 (interleaved) executions:

$\langle \ell_0, m_0, r_1{:}-, r_2{:}-, y{:}0\rangle, \langle \ell_1, m_0, r_1{:}0, r_2{:}-, y{:}0\rangle, \langle \ell_1, m_1, r_1{:}0, r_2{:}0, y{:}0\rangle$
$\langle \ell_2, m_1, r_1{:}1, r_2{:}0, y{:}0\rangle, \langle \ell_2, m_2, r_1{:}1, r_2{:}-1, y{:}0\rangle,$
$\langle \ell_3, m_2, r_1{:}1, r_2{:}-1, y{:}1\rangle, \langle \ell_3, m_3, r_1{:}1, r_2{:}-1, y{:}-1\rangle$
$\langle \ell_0, m_0, r_1{:}-, r_2{:}-, y{:}0\rangle, \langle \ell_1, m_0, r_1{:}0, r_2{:}-, y{:}0\rangle, \langle \ell_2, m_1, r_1{:}0, r_2{:}-, y{:}0\rangle$
$\langle \ell_3, m_0, r_1{:}1, r_2{:}-, y{:}1\rangle, \langle \ell_3, m_1, r_1{:}1, r_2{:}1, y{:}1\rangle,$
$\langle \ell_3, m_2, r_1{:}1, r_2{:}0, y{:}1\rangle, \langle \ell_3, m_3, r_1{:}1, r_2{:}0, y{:}0\rangle$
$\langle \ell_0, m_0, r_1{:}-, r_2{:}-, y{:}0\rangle, \langle \ell_1, m_0, r_1{:}0, r_2{:}-, y{:}0\rangle, \langle \ell_1, m_1, r_1{:}0, r_2{:}0, y{:}0\rangle$
$\langle \ell_2, m_1, r_1{:}1, r_2{:}0, y{:}0\rangle, \langle \ell_2, m_2, r_1{:}1, r_2{:}-1, y{:}0\rangle,$
$\langle \ell_2, m_3, r_1{:}1, r_2{:}-1, y{:}-1\rangle, \langle \ell_3, m_3, r_1{:}1, r_2{:}-1, y{:}+1\rangle$

The problem with the original program is that it contains statements such as $y := y + 1$ which perform two accesses to the shared variable y in a single atomic transition.

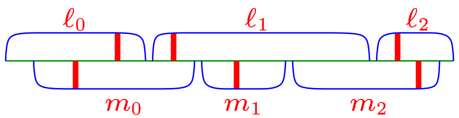
To remedy this situation, we will restrict the number of accesses to shared variables that may occur within each statement.

Limited Critical References (LCR) Programs

A program is called an Limited Critical Access program (an LCR program) if each statement contains at most one reference to a shared variables. Note that that original $y := y + 1$ program was not an LCR program, while the (r_1, r_2)-program is LCR.

Claim 1. *If P is an LCR program, then its interleaved execution is equivalent to a really parallel execution of P.*

To justify the claim, consider the following diagram which depicts a realistic execution of the (r_1, r_2)-program.



In this picture, each instruction takes some positive time to execute. Within each instruction, we marked by red the single access to a shared variable. We assume that such accesses to shared memory are atomic. We claim that the result of such an execution will be equivalent to an interleaved execution in which instructions ordered according to the ordering in time of the critical accesses. For the displayed example, this will be the sequence:

$$m_0,\ \ell_0,\ \ell_1,\ m_1,\ \ell_2,\ m_2$$

Course G22.3033.007 Lecture 2 R. Dewar and A. Pnueli

Extensions of the LCR Definition

There are two points in which we can generalize the LCR definition, such that Claim 1 will still hold.

We define a reference to a variable within process P_i to be critical if it is

- A writing reference to a variable which is accessed (read or written) by a process parallel to P_i, or
- A reading reference to a variable which is modified by a process parallel to P_i.

In particular, we exclude from this definition a reading reference to a variable which can only be modified by P_i itself.

A program is defined to be an LCR program if each transition contains at most one critical reference.

Another extension allows statements of the form **await** $(p \lor q)$, where each of p, q contains at most one critical reference. The justification for this is that every such **await** statement can be replaced by the following LCR segment:

$$
\begin{array}{l}
\ell_1: \quad done := 0 \\
\ell_2: \quad \textbf{while } \neg done \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_3: \quad \textbf{if } p \textbf{ then} \\
\quad \ell_4: \quad done := 1 \\
\ell_5: \quad \textbf{if } q \textbf{ then} \\
\quad \ell_6: \quad done := 1
\end{array}\right]
\end{array}
$$

The Atomic Version of Peterson's Program is not LCR

Reconsider Peterson's program:

$$
\begin{array}{lll}
\textbf{local} & y_1, y_2 & : \textbf{boolean where } y_1 = y_2 = 0 \\
 & s & : \{1,2\} \textbf{ where } s = 1
\end{array}
$$

$$
\left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\left[\begin{array}{l}
\ell_1 : \text{Non-Critical} \\
\ell_2 : (y_1, s) := (1, 1) \\
\ell_3 : \textbf{await } y_2 = 0 \ \vee \ s \neq 1 \\
\ell_4 : \text{Critical} \\
\ell_5 : y_1 := 0
\end{array}\right]
\end{array}\right]
\quad \| \quad
\left[\begin{array}{l}
m_0 : \textbf{loop forever do} \\
\left[\begin{array}{l}
m_1 : \text{Non-Critical} \\
m_2 : (y_2, s) := (1, 2) \\
m_3 : \textbf{await } y_1 = 0 \ \vee \ s \neq 2 \\
m_4 : \text{Critical} \\
m_5 : y_2 := 0
\end{array}\right]
\end{array}\right]
$$

$$- \ P_1 \ - \qquad\qquad - \ P_2 \ -$$

This program is not LCR. The main culprits are the joint assignments ℓ_2 and m_2. Note that the **await** statements do satisfy the (extended) LCR restriction.

There are two ways to transform this program into an LCR program.

Bad LCR Version of Peterson(2)

$$
\begin{array}{lll}
\textbf{local} & y_1, y_2 & : \textbf{boolean where } y_1 = y_2 = 0 \\
 & s & : \{1,2\} \textbf{ where } s = 1
\end{array}
$$

$$
\left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\left[\begin{array}{l}
\ell_1 : \text{Non-Critical} \\
\ell_2 : s := 1 \\
\ell_3 : y_1 := 1 \\
\ell_4 : \textbf{await } y_2 = 0 \ \vee \ s \neq 1 \\
\ell_5 : \text{Critical} \\
\ell_6 : y_1 := 0
\end{array}\right]
\end{array}\right]
\quad \| \quad
\left[\begin{array}{l}
m_0 : \textbf{loop forever do} \\
\left[\begin{array}{l}
m_1 : \text{Non-Critical} \\
m_2 : s := 2 \\
m_3 : y_2 := 1 \\
m_4 : \textbf{await } y_1 = 0 \ \vee \ s \neq 2 \\
m_5 : \text{Critical} \\
m_6 : y_2 := 0
\end{array}\right]
\end{array}\right]
$$

$$- \ P_1 \ - \qquad\qquad - \ P_2 \ -$$

This version violates mutual exclusion, as can be observed by the following computation:

$$
\begin{array}{lll}
\cdots & \xrightarrow{m_2} & \langle \ell_3, m_3, y_1: 0, y_2: 0, s: 2 \rangle \xrightarrow{m_3} \\
\langle \ell_3, m_4, y_1: 0, y_2: 1, s: 2 \rangle & \xrightarrow{m_4} & \langle \ell_3, m_5, y_1: 0, y_2: 1, s: 2 \rangle \xrightarrow{\ell_3} \\
\langle \ell_4, m_5, y_1: 1, y_2: 1, s: 2 \rangle & \xrightarrow{\ell_4} & \boxed{\langle \ell_5, m_5, y_1: 1, y_2: 1, s: 2 \rangle}
\end{array}
$$

Good LCR Version of Peterson(2)

local y_1, y_2 : **boolean where** $y_1 = y_2 = 0$
s : $\{1, 2\}$ **where** $s = 1$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \text{Non-Critical} \\ \ell_2 : y_1 := 1 \\ \ell_3 : s := 1 \\ \ell_4 : \textbf{await } y_2 = 0 \ \lor \ s \neq 1 \\ \ell_5 : \text{Critical} \\ \ell_6 : y_1 := 0 \end{array}\right] \end{array}\right] \quad \| \quad \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} m_1 : \text{Non-Critical} \\ m_2 : y_2 := 1 \\ m_3 : s := 2 \\ m_4 : \textbf{await } y_1 = 0 \ \lor \ s \neq 2 \\ m_5 : \text{Critical} \\ m_6 : y_2 := 0 \end{array}\right] \end{array}\right]$$

$$- \quad P_1 \quad - \qquad\qquad\qquad - \quad P_2 \quad -$$

This program satisfies the properties of mutual exclusion, deadlock absence, and accessibility.

It can be generalized to deal with N processes.