Suggested Solutions for

Written Exam, December 10, 2019

PROBLEM 1

Question 1.1

A direct translation yields:

[By recognizing the cross-signalling between \( P_A \) and \( P_B \) as well as the back-and-forth signalling between \( P_B \) and \( P_C \) to be true synchronizations, this may also be expressed as:]

Question 1.2

The following inequalities may be seen, e.g. by applying the semaphore invariant:

\[ I \triangleq c \leq a \leq c + 3 \]

Question 1.3

The above Petri Net is readily implemented using synchronous communications:

\[
\begin{align*}
\text{process } P_1; & \quad \text{process } P_2; & \quad \text{process } P_3; \\
\text{repeat} & \quad \text{repeat} & \quad \text{repeat} \\
A; & \quad B; & \quad \text{repeat} \\
P_2!(); & \quad P_1?(); & \quad P_2?(); \\
\text{forever} & \quad P_3!(); & \quad C \\
\text{forever} & \quad \text{forever}
\end{align*}
\]
PROBLEM 2

Question 2.1

(a) Transition diagrams:

![Transition diagrams](attachment:image.png)

[Location and action labels not required. Note that \( a_2 \) can be considered atomic.]

(b) [Rather than working through the 35 possible interleavings, we observe that the final value of \( y \) may be set by either \( a_1 \) (to 1) or \( b_4 \). In the latter case, the value depends on readings of \( x \) and \( y \). If \( x \) has not been changed, it is 0 and \( y \) may be read as either 1 or 4 yielding values 2 and 5 to \( y \). If \( x \) has been changed, \( P_1 \) may have read \( y \) as 1 giving \( x = 3 \). As \( P_2 \) may read \( y \) as 1 or 4, this yields final values 5 and 8. Finally \( P_1 \) may read \( y \) as 4 giving \( x = 6 \) and \( P_2 \) will then also read \( y = 4 \) yielding a final \( y \)-value of 11.]

It is found that the final value of \( y \) may be either of

\[ 1, 2, 5, 8, 11 \]

Question 2.2

(a) \( I \) holds initially since \( x = 0 \land y = 0 \).

Checking all atomic actions:

- \( a_1 \): Since both \( x \) and \( y \) are non-negative before, so are the expressions \( x + y \) and \( y + 1 \) and hence \( 0 \leq x \land 0 \leq y \) holds after the action.
  Before the action \( x \leq 1 \) and \( y \leq x + 1 \) implies \( y \leq 2 \) and therefore \( x + y \leq 3 \) why \( x \leq 3 \) holds after the action.
  If \( y = x + 1 \), \( a_1 \) increments \( x \) by at least 1 and \( y \) by 1 preserving the inequality \( y \leq x + 1 \). If \( y < x + 1 \), incrementing \( y \) by 1 and not decrementing \( x \) ensures \( y \leq x + 1 \) after the action.

- \( a_2 \): After the action, \( (x, y) = (1, 1) \) which satisfies \( I \).

- \( a_3 \): By \( I \) and the condition, \( 0 \leq y \leq 3 \) before the action and hence \( 0 \leq x \leq 3 \) holds after the action. As \( y \) becomes 0, also \( 0 \leq y \leq x + 1 \) is satisfied after the action.

Since \( I \) holds initially and is preserved by all atomic actions, \( I \) is an invariant of the program.
(b) Transition graph:

(c) The predicate \( I \wedge y \leq 3 \) is seen to encompass all the reachable states and is thus an invariant of the program. However, as it allows states which is are not reachable (e.g. \((2,1)\)), the predicate is not a characteristic invariant of the program.

(d) Assuming weak fairness

- **F holds.** [As control cannot remain at a state with outgoing transitions, any execution will lead to \((0,0)\) from where \(a_1\) takes the program to \((0,1)\).]
- **G does not hold.** [The only reachable states satisfying \(y = 2(x - 1)\) are \((1,0)\) and \((2,2)\). These may be avoided by an execution following an outer cycle:

\[
(0,0) \xrightarrow{a_1} (0,1) \xrightarrow{a_1} (1,2) \xrightarrow{a_1} (3,3) \xrightarrow{a_3} (3,0) \xrightarrow{a_1} (0,0) \xrightarrow{a_1} \cdots
\]

Since this execution path does not remain at a single state, it satisfies weak fairness.]
- **H does not hold.** [The cycle (*) also avoids \(x = 2\).]
- **J does not hold.** [From \((0,1)\) or \((1,1)\), the execution may follow the inner cycle

\[
(0,1) \xrightarrow{a_2} (1,1) \xrightarrow{a_3} (1,0) \xrightarrow{a_1} (0,0) \xrightarrow{a_1} (0,1) \xrightarrow{a_2} \cdots
\]

avoiding states where \(x \geq 2\).]

Assuming strong fairness

- **F holds.** [By weak fairness.]
- **G holds.** [As \((0,1)\) is reached infinitely often (cf. \(F\)), \(a_2\) is enabled infinitely often and hence \((1,1)\) is reached infinitely often. From \((1,1)\) either \((1,0)\) or \((2,2)\) must be reached.]
- **H holds.** [As \((1,1)\) is reached infinitely often, if \((1,0)\) is to be avoided forever, the execution must continue to \((2,2)\).]
- **J does not hold.** [In the sequence (**) all actions are executed infinitely often and hence also strong fairness is satisfied.]
PROBLEM 3

Question 3.1

(a) The if-statement ensures that all pending calls of pass() will be accepted when a multiple is reached.

(b) In the module header the declaration op set(l : integer); is inserted. The operation may be implemented by adding the following branch to the outermost in-construct:

\[
\begin{align*}
\text{set}(l : \text{integer}) \text{ and } l \geq 2 \rightarrow \\
\text{while } \text{count} \neq 0 \text{ do} \\
\text{in incr()} \rightarrow \text{count} := (\text{count} + 1) \mod k \ni l; \\
k := l
\end{align*}
\]

[It is not specified what calling set(l) with l < 2 should do. Here such calls just block.
As an alternative solution, set may be accepted only when count = 0 and then preventing calls of incr() from being accepted when there are pending calls of of set when count = 0.]

Question 3.2

(a) Synchronization code for each process \( P_i \) [\( i : 1..n \)]:

\[
\begin{align*}
\text{ModCount}.\text{incr}(); \\
\text{ModCount}.\text{pass}(); \\
\end{align*}
\]

The constant \( K_0 \) must be defined to \( n \).

(b) If the above synchronization code is used for more than one synchronization point, it cannot be guaranteed that all processes have called pass(), before incr() is called for the next synchronization point rendering the processes stuck with a non-zero value of count.

This issue is solved by using two of these one-time barriers in each round:

\[
\begin{align*}
\text{ModCount}_1.\text{incr}(); \\
\text{ModCount}_1.\text{pass}(); \\
\text{ModCount}_2.\text{incr}(); \\
\text{ModCount}_2.\text{pass}(); \\
\end{align*}
\]

For both module instances, \( K_0 \) must be defined to \( n \).
Question 3.3

(a) The ModCount module is readily implemented as a monitor:

```plaintext
monitor ModCount

var count : integer := 0;
    Zero : condition;

procedure incr() {
    count := (count + 1) mod K;
    if count = 0 then signal_all(Zero)
}

procedure pass() {
    if count ≠ 0 then wait(Zero)
}
end
```

(b) Calls of pass() should be waiting only if count is not zero:

\[ I \triangleq waiting(Zero) > 0 \Rightarrow count \neq 0 \]

I holds initially as Zero is empty. The queue size is only incremented when count is non-zero and whenever count becomes zero, the condition queue is completely emptied. Hence I is a monitor invariant.

(c) The solution is obviously not robust towards spurious wakeups, as the wait condition in pass() is not rechecked. Changing the if-statement to a while-statement does remedy this, but then the behaviour of ModCount is changed, as not all pending calls of pass() are guaranteed to be released when the next multiple is reached.

A robust solution can be obtained by holding back calls of incr() and pass() while Zero is being emptied:

```plaintext
monitor ModCount

var count, rem : integer := 0;
    Zero, Hold : condition;

procedure incr() {
    while rem > 0 do wait(Hold);
    count := (count + 1) mod K;
    if count = 0 then {
        rem := length(Zero);
        signal_all(Zero)
    }
}

procedure pass() {
    while rem > 0 do wait(Hold);
    while count ≠ 0 do wait(Zero);
    if rem > 0 then {
        rem := rem - 1;
        if rem = 0 then signal_all(Hold) }
}
end
```
A slightly simpler solution utilizes the round-counting idea for robustness:

```pascal
monitor ModCount

var count, round : integer := 0;
   Zero : condition;

procedure incr() {
   count := (count + 1) mod K0;
   if count = 0 then {
      round := round + 1;
      signal_all(Zero)
   }
}

procedure pass() {
   var myround : integer := round;
   if count ≠ 0 then
      while myround = round do wait(Zero);
}

end
```