Suggested Solutions for

Written Exam, December 11, 2018

PROBLEM 1

Question 1.1

A direct translation of the CSP program to a Petri Net yields:

\[ I \triangleq |a - c| \leq 1 \]

Question 1.2

As \( A \) and \( C \) are seen to synchronize with each other for each round and as \( A \) and \( C \) may actually be executed concurrently (if \( P_2 \) executes \( B \) and then synchronizes with \( P_3 \)), the following predicate is a characteristic invariant of the program:

\[ I \triangleq |a - c| \leq 1 \]

Question 1.3

The selective synchronization by \( P_2 \) with \( P_1 \) and \( P_3 \) can standarly be realized with a pair of semaphores. Likewise, the synchronization of \( A \) and \( C \) can be obtained by cross-signalling:

```plaintext
var S_A, S_B, S_C, S_AC : semaphore;
S_A := 0; S_B := 0; S_C := 0; S_AC := 0;

process P_A;
repeat
    A;
    P(S_AC);
    V(S_B);
    V(S_C);
    P(S_A)
forever

process P_B;
repeat
    B;
    V(S_AC);
    P(S_B)
forever

process P_C;
repeat
    P(S_AC);
    V(S_B);
    C;
    V(S_A);
    P(S_C)
forever
```
PROBLEM 2

Question 2.1

(a) The statement pairs are checked for critical references with respect to each other:

<table>
<thead>
<tr>
<th>Pair</th>
<th>Mutually atomic</th>
<th>Rationale [not required]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b</td>
<td>NO</td>
<td>Two critical references in a.</td>
</tr>
<tr>
<td>a, c</td>
<td>YES</td>
<td>Only one critical reference in each statement.</td>
</tr>
<tr>
<td>a, d</td>
<td>NO</td>
<td>Both reading and writing of x in a are critical.</td>
</tr>
<tr>
<td>b, c</td>
<td>NO</td>
<td>In c, writing y and reading x are both critical.</td>
</tr>
<tr>
<td>b, d</td>
<td>YES</td>
<td>Only one critical reference in d.</td>
</tr>
<tr>
<td>c, d</td>
<td>YES</td>
<td>In c, only the reading of x is critical.</td>
</tr>
</tbody>
</table>

[Being executed indivisibly, all reads and writes in statement b together only count as a single critical reference.]

(b) Transition diagrams:

=c⃣onian

(c) Going through the three possible interleavings of the atomic actions, the possible final values of (x, y) are found to be:

(2, 5), (2, 3), (5, 3)

Question 2.2

(a) I holds initially since x = 0 ∧ y = 0 ⇒ x = y.

Checking all atomic actions:

a₁: By cases of I:

Case y = x: In terms of the old values the new sum is y + (3 - x), but as x = y, this equals 3. Thus I is preserved.

Case x + y = 3: The new value of y is 3 minus the old value of x, but this must then be equal to the old value of y. Thus, both variables are assigned the old value of y and hence I holds.

a₂: Executed only if x = y and since both variables are incremented by 1, this property is preserved and therefore I holds.

a₃: This action always sets (x, y) to (3, 0) and thus the x + y = 3 case of I holds.

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

```
(0, 3)  a1  (3, 3)
   /     \
(1, 2)  a1  (2, 2)
   \
   (1, 1)  a1  (2, 1)
   /     \
(0, 0)  a1  (3, 0)  a3
```

(c) Defining

\[ H \triangleq 0 \leq x \leq 3 \land 0 \leq y \leq 3 \]

it is seen that \( H \land I \) becomes a characteristic invariant. [In fact, \( H \triangleq 0 \leq x \leq 3 \) suffices.]

(d) Assuming weak fairness

- **F does not** hold. [From the state \((0, 0)\) the execution may go to state \((1, 1)\) and from there follow any combination of the two inner cycles:

\[
(1, 1) \xrightarrow{a_1} (1, 2) \xrightarrow{a_1} (2, 2) \xrightarrow{a_1} (2, 1) \xrightarrow{a_1} (1, 1) \rightarrow \cdots \quad (\ast)
\]

and

\[
(1, 1) \xrightarrow{a_2} (2, 2) \xrightarrow{a_1} (2, 1) \xrightarrow{a_1} (1, 1) \rightarrow \cdots \quad (\ast\ast)
\]

Since any such execution path does not remain at a single state, it satisfies weak fairness. In all cases, \( x = 3 \) will never occur.]

- **G holds.** [From \( \Diamond \Box (x \neq 1) \) it follows that at some point the inner cycles are left and then the execution must follow the outer cycle:

\[
(0, 0) \xrightarrow{a_1} (0, 3) \xrightarrow{a_1} (3, 3) \xrightarrow{a_1} (3, 0) \xrightarrow{a_1} (0, 0) \xrightarrow{a_1} \cdots \quad (\ast\ast\ast)
\]

where \( x \) will become 0. (Actually, infinitely often.)]

- **H does not** hold. [The only reachable state for which \( x + y \geq 5 \) is \((3, 3)\) and by the argument for \( F \), \( x = 3 \) might never occur under weak fairness.]

- **J does not** hold. [From the states with \( x = 2 \), \((2, 2)\) and \((2, 1)\), the outer cycle \((\ast\ast\ast)\) may be reached immediately and then followed forever, avoiding \( x = 1 \).]

**Assuming strong fairness**

- **F holds.** [The execution cannot any longer stay in the inner cycles \((\ast)\) and \((\ast\ast)\) as \( a_3 \) will then be enabled infinitely often and hence must be taken leading to the state \((3, 0)\).]

- **G holds.** [By weak fairness.]
• **H does not** hold. [In the cycle
\[(0, 0) \xrightarrow{a_2} (1, 1) \xrightarrow{a_2} (2, 2) \xrightarrow{a_1} (2, 1) \xrightarrow{a_3} (3, 0) \xrightarrow{a_1} (0, 0) \rightarrow \cdots\]
all actions are executed and hence strong fairness is satisfied. However, the state \((3, 3)\) is never reached.]

• **J holds.** [From \((2, 2)\) and \((2, 1)\), the only way to avoid the state \((1, 1)\) would be to enter and remain in the outer loop (*****). However, in this, \(a_2\) would be infinitely often enabled without being taken violating strong fairness. ]

**PROBLEM 3**

**Question 3.1**

(a) [The monitor will let the first writer start writing and then queue up the first reader on the *head* condition queue and the second reader on the *tail* condition queue where it will be followed by the second writer and the third reader. When the first writer is done, the first two readers are allowed to read while the second writer is moved to the *head* queue. The third reader is still in the *tail* queue where it is joined by the fourth reader. Thus:]

\[r = 2 \land w = 0 \land waiting(head) = 1 \land waiting(tail) = 2\]

(b) Initially, \(I\) holds trivially because *tail* is empty. In *lock*, calls only wait on *tail* if *heading* is true. Also, *heading* is set to false only when *tail* is empty. Hence \(I\) is preserved in *lock* and as it is trivially preserved by *unlock*, \(I\) is an invariant of the monitor.

(c) The following should be an invariant of the monitor:

For any (single) call *lock*(write) waiting in *head* either \(w > 0\) or *write* is true while \(r > 0\).

**Question 3.2**

In the declaration part of the monitor, a limit variable \(lim\) is added:

\[lim : \text{posinteger} := L;\]

The operation *setLimit* is defined as:

```
procedure setLimit(k : \text{posinteger}) {
    if \(lim \leq r < k\) then signal(head);
    lim := k;
}
```

In *lock*, the given wait condition must be weakened to take the limit into account:

\[\text{while } w > 0 \lor (\text{write} \land r > 0) \lor r \geq lim \text{ do } \{ \ldots \};\]

Finally, the signalling in *unlock* must be extended:

\[\text{if } r = 0 \lor r = lim - 1 \text{ then signal(head)}\]

[The conditions constrain the signalling such that superfluous wakeups will take place only in the very special situation that a writer is waiting in *head* and the number of active readers has exactly reached the current limit.]
Question 3.3

(a) In basic read/write locking, lock calls are accepted whenever the conditions are legal:

```verbatim
module RWLock
    op lock(write : boolean);
    op unlock();
body
    process BasicControl;
    var r, w : integer := 0;
    repeat
        in lock(write) and w = 0 ∧ (write ⇒ r = 0) →
            if write then w := w + 1 else r := r + 1
        unlock() → if w > 0 then w := w - 1 else r := r - 1
    ni
    forever;
end RWLock;
```

(b) To handle lock calls in FCFS order these must be accepted without filtering on the write parameter. If actually a write, all reader activity must cease before ending the rendezvous.

```verbatim
module RWLock
    op lock(write : boolean);
    op unlock();
body
    process FCFS_Control;
    var r, w : integer := 0;
    repeat
        in lock(write) and w = 0 → while write ∧ r > 0 do
            in unlock() → r := r - 1 ni;
            if write then w := w + 1
            else r := r + 1
        unlock() → if w > 0 then w := w - 1 else r := r - 1
    ni
    forever;
end RWLock;
```