Written examination, December 11, 2018

Course: Concurrent Programming Course no. 02158

Aids allowed: All written works of reference

Exam duration: 2 hours

Weighting: PROBLEM 1: approx. 30 % PROBLEM 3: approx. 40 %
PROBLEM 2: approx. 30 %

PROBLEM 1 (approx. 30 %)

Three processes P_1, P_2, and P_3 execute three operations A, B, and C respectively. The operations are to be synchronized which is accomplished by exchanging void messages using CSP's synchronous communication:

```
process P1;          process P2;                  process P3;
  repeat               repeat                       repeat
    A;                   B;                           P2 ! ();
    P2 ! ();             if P1 ? () → skip            C;
    P3 ? ()              [] P3 ? () → skip            P1 ! ()
  forever                fi                         forever
                       forever
```

Question 1.1:

Draw a Petri Net in which the three operations A, B, and C are synchronized in the same way as in the above program. In the net, the operations must appear as transitions.

Question 1.2:

Let the number of times the operations A and C have been executed be denoted by a and c respectively. Define a predicate I which characterizes the reachable combinations of a and c in the above program.

[I should thus be an invariant of the program, but you need not show this.]

Question 1.3:

The operations are now to be executed by three sequential processes P_A, P_B, and P_C:

```
process PA;     process PB;     process PC;
  repeat          repeat          repeat
    A               B               C
  forever         forever         forever
```

Show how semaphores can be used to synchronize the three processes so that A, B, and C become synchronized in the same way as in the above, CSP-based program.

PROBLEM 2 (approx. 30 %)

The questions in this problem can be solved independently of each other.

Question 2.1:

Let x and y be integer variables. Consider the four statements a, b, c, and d:

$$\begin{array}{ll} a\text{:} & x := x + 1 \\ b\text{:} & \langle\, x := y + 2 \,\rangle \\ c\text{:} & y := x + 3 \\ d\text{:} & x := 4 \end{array}$$

[Above, $\langle \ldots \rangle$ indicates that the statement is executed indivisibly.]

(a) For each of the six possible selections of two different statements, determine whether the two statements are *mutually atomic*.

(b) Assume that the statements b and c are executed concurrently. For each of the two statements, draw a transition diagram showing its atomic actions.

(c) Determine all possible final values of (x, y), if the concurrent execution of b and c is started in the state $(0, 0)$.

Question 2.2:

Consider the concurrent program:

var x, y : *integer* $:= 0$;
co
  repeat a_1: $\langle\ (x, y) := (y, 3 - x)\ \rangle$ **forever**
‖
  repeat a_2: $\langle\, x = y \wedge x < 3 \rightarrow x := x + 1;\ \ y := y + 1 \,\rangle$ **forever**
‖
  repeat a_3: $\langle\, y < x \rightarrow x := 3;\ \ y := 0 \,\rangle$ **forever**
oc

(a) Prove inductively that following predicate I is an invariant of the program:

$$I \ \triangleq\ x = y \ \vee\ x + y = 3$$

(b) Draw the (reachable part of) the transition graph for the program. Only the (x, y) part of the state has to be shown.

(c) Define a predicate H constraining the ranges of x and y so that $H \wedge I$ becomes a *characteristic invariant* of the program (i.e. exactly describes the set of reachable (x, y) states).

(d) Consider the following temporal logic properties:

$$F \ \triangleq\ \Box\Diamond\,(x = 3) \qquad\qquad H \ \triangleq\ \Box\Diamond\,(x + y \geq 5)$$
$$G \ \triangleq\ \Diamond\Box\,(x \neq 1) \Rightarrow \Diamond\,(x = 0) \qquad\qquad J \ \triangleq\ x = 2 \leadsto x = 1$$

Determine for each of F, G, H, and J whether the property holds for the program under the assumption of weak fairness. Do the same under the assumption of strong fairness.

PROBLEM 3 (approx. 40 %)

The questions in this problem can be solved independently of each other.

Below, a monitor implementation of a *read/write lock* is given. Readers call *lock*(*false*) before reading and writers call *lock*(*true*) before writing. When done, both readers and writers call *unlock*(). The implementation ensures fairness by handling *lock* calls in strictly *first-come-first-served* (FCFS) order. This is accomplished by using one condition queue, *head*, to hold the first waiting *lock* call and another condition queue, *tail*, to hold the subsequent calls in FIFO order.

```
monitor FairRWLock

  var r, w : integer := 0;
      heading : boolean := false;
      head, tail : condition;

  procedure lock(write : boolean) {
    if heading then wait(tail);
    while w > 0 ∨ (write ∧ r > 0) do { heading := true;  wait(head) };
    if write then w := w + 1 else r := r + 1;
    if empty(tail) then heading := false
                   else signal(tail)
  }

  procedure unlock() {
    if w > 0 then w := w − 1 else r := r − 1;
    if r = 0 then signal(head)
  }

end
```

Question 3.1:

(a) Determine the values of r and w as well as the numbers of waiting processes, *waiting*(*head*) and *waiting*(*tail*), if the following sequence of monitor calls has been issued and the monitor has come to rest:

lock(*true*), *lock*(*false*), *lock*(*false*), *lock*(*true*), *lock*(*false*), *unlock*(), *lock*(*false*)

(b) Argue that $I \triangleq waiting(tail) > 0 \Rightarrow heading$ is a monitor invariant.

(c) State informally (*i.e. in words*) a monitor invariant which expresses that calls of *lock* do not wait unnecessarily on the *head* condition queue.

Question 3.2:

It is desired to limit the number of active readers. For a given limit k, no further readers may pass the *lock* operation while the number of active readers, r, has reached (or exceeds) k. The initial limit is given by a positive integer constant L. It must be possible to change the limit dynamically using an operation *setLimit*(k : *posinteger*) where *posinteger* is the type of positive integers. A call of *setLimit*(k) should immediately set the limit to k.

Write such a *setLimit* operation and state which changes must be made to the given monitor *FairRWLock* in order to implement the reader limitation efficiently.

The problem is continued on the next page

Question 3.3:

The read/write locking is now to be implemented by a module specified by:

```
module RWLock
  op lock(write : boolean);
  op unlock();
end
```

(a) Write a server process for the module *RWLock* which services the operations by rendezvous in such a way that it functions like a basic read/write lock. The calls of *lock* may be served in any feasible order and the solution does not have to be fair.

(b) Write an alternative server process for the module *RWLock* so that the module functions like the given monitor *FairRWlock*. In particular, calls of *lock* should be served in FCFS order.

[If your solution to (a) already fulfils this requirement, you may just refer to that.]