Written examination, December 9, 2020

Course: Concurrent Programming Course no. 02158

Aids allowed: All

Exam duration: 4 hours

Weighting:	PROBLEM 1:	approx. 15 %	PROBLEM 3:	approx. 35 %
	PROBLEM 2:	approx. 20 %	PROBLEM 4:	approx. 30 %

PROBLEM 1 (approx. 15 %)

Three processes P_A, P_B, and P_C execute three operations A, B, and C respectively. Furthermore, P_C executes a fourth operation D. The operations are to be synchronized, which is accomplished by means of semaphores:

var SA, SB, SC, SD : *semaphore*;

$SA := 0;\;\; SB := 0;\;\; SC := 0;\;\; SD := 0;$

process P_A;	**process** P_B;	**process** P_C;
repeat	**repeat**	**repeat**
A;	P(SA);	C;
V(SA);	P(SC);	V(SC);
P(SD)	B;	D;
forever	V(SB)	P(SB);
	forever	V(SD)
		forever

Question 1.1:

Draw a Petri Net in which the four operations A, B, C, and D are synchronized in the same way as in the above program. In the net, the operations should be represented by transitions.

Question 1.2:

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

Question 1.3:

The operations are now to be executed by four sequential CSP-processes P_1, P_2, P_3, and P_4 respectively:

process P_1;	**process** P_2;	**process** P_3;	**process** P_4;
repeat	**repeat**	**repeat**	**repeat**
A	B	C	D
forever	**forever**	**forever**	**forever**

Show how the processes may exchange void messages using CSP's synchronous communication so that A, B, C, and D are synchronized in the same way as in the above, semaphore-based program.

PROBLEM 2 (approx. 20 %)

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:

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

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

(b) Determine all possible final values of (x, y), if the two statements a and b are executed concurrently starting in the state $(0, 0)$.

Question 2.2:

Consider the concurrent program:

$$\begin{array}{l}
\textbf{var } x, y : \mathit{integer} := 0; \\[1ex]
\textbf{co} \\
\quad \textbf{repeat } a_1 \text{: } \langle\, x < 2 \rightarrow x := x + 1; \;\; y := x \,\rangle \textbf{ forever} \\
\| \\
\quad \textbf{repeat } a_2 \text{: } \langle\, x := y; \;\; y := 0 \,\rangle \textbf{ forever} \\
\| \\
\quad \textbf{repeat } a_3 \text{: } \langle\, x = 1 \wedge y > 0 \rightarrow y := 2 \,\rangle \textbf{ forever} \\
\textbf{oc}
\end{array}$$

(a) Prove inductively that $I \triangleq (x = 0 \Rightarrow y = 0)$ is an invariant of the program.

(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) Consider the following temporal logic properties:

$$\begin{array}{lll} F \triangleq \Box\Diamond\,(\, y > x \,) & \qquad & H \triangleq \Box\Diamond\,(\, y = 0 \wedge x > 0 \,) \\ G \triangleq x = 1 \leadsto x = 2 & \qquad & J \triangleq x + y = 2 \leadsto x + y = 4 \end{array}$$

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.

(d) Assume that the action a_3 cannot be considered atomic as a whole.

Draw a transition diagram representing the refinement of a_3 into (conditional) atomic actions.

(e) Show that I is **not** an invariant of the refined program.

PROBLEM 3 (approx. 35 %)

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

In a system, there is a synchronization component called an *(extended) work group* which may be seen as a shared object WG with two operations *waitAdd* and *add* specified by:

object WG;

var $count : integer := 0$;

op $waitAdd(k : integer) : \langle\, count \leq 0 \rightarrow count := count + k \,\rangle$;

op $add(k : integer) : \langle\, count := count + k \,\rangle$;

end

Question 3.1:

Implement WG as a monitor.

Question 3.2:

Implement the operations of WG using semaphores for synchronization. The technique of *passing-the-baton* should be applied for that.

Question 3.3:

In a CSP program it is desired to synchronize a family of processes, $Client[i : 1..N]$, by a component behaving like WG. This is to be implemented by a dedicated process called *Server*. The *Client* processes should use the component through communications with the *Server* processs on different ports:

$waitAdd(k)$:	$Server\,!\,WaitAdd(k)$
$add(k)$:	$Server\,!\,Add(k)$

Write the *Server* process such that it functions like the given WG component as seen from the *Client* processes.

Question 3.4:

Assume that processes may be created dynamically by the statement **start** $\{SL\}$ which starts a new process executing the statement list SL concurrently with the initiating process.

Now a main process should start three concurrent subprocesses executing statement lists SL_1, SL_2, and SL_3 respectively and then wait until all three lists have been executed. Show how to use the given WG component to accomplish this.

Question 3.5:

Show how a group of processes may use the given WG component to establish a *critical region*.

Question 3.6:

Show how a system of N $(N \geq 1)$ *reader processes* and a **single** *writer process* may use the given WG component for reader/writer synchronization. You may assume an initial call of *add* to be made in order to bring the component into a desired state.

PROBLEM 4 (approx. 30 %)

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

A given type X of problems have solutions of type Y given by a function $Solve : X \rightarrow Y$. The solution for a given instance of the problem can be calculated by a number of different sequential algorithms having different execution times. However, it cannot be determined in advance, which algorithm is the best one for a given problem instance.

In a system with plenty of processors, these can be used to run several different algorithms in parallel and then use the first solution available. Below, this idea is implemented by a coordinator component *ParSolve* which controls the problem solving activity.

The coordinator provides an operation $solve(x)$ to be called by a single *user process* when it wants to obtain a solution to a problem instance x. A number of *worker processes* run different solution algorithms. Each worker first calls the operation $get()$ to obtain a problem, then solves it using this worker's distinct algorithm, and finally returns the solution by calling a *result* operation. This behaviour is repeated forever. To ensure that late solutions are not used for new problems, the problems are assigned unique *version numbers* to be used when delivering their solutions.

Below, the *ParSolve* component is implemented as a monitor:

```
monitor ParSolve

  var done : boolean := true;
      no : integer := 0;
      prob : X;
      sol : Y;
      solOk, newProblem : condition;

  function solve(x : X) returns Y {
    no := no + 1;
    prob := x;
    done := false;
    signal_all(newProblem);
    while ¬done do wait(solOk);
    return sol
  }

  function get() returns (X, integer) {
    while done do wait(newProblem);
    return (prob, no)
  }

  procedure result(y : Y, ver : integer) {
    if ¬done ∧ ver = no then { sol := y;
                               done := true;
                               signal(solOk) }
  }

end
```

The problem is continued on the next page

Question 4.1:

(a) State a monitor invariant which expresses that calls of *get* do not wait unnecessarily.

(b) Prove that the invariant holds for the monitor.

Question 4.2:

In this question it is desired to limit the number of solution algorithms pursued concurrently for a given problem. This should be done by augmenting the *solve* operation with an ekstra parameter k defining such a limit for the given problem.

Describe which changes must be made to the given monitor *ParSolve* in order to implement the modified operation $solve(x : X, k : posinteger)$ where *posinteger* is the type of positive integers.

Question 4.3:

The given monitor *ParSolve* works under the assumption that there is only one user process calling *solve*. Now suppose that several user processes exist calling *solve* concurrently.

(a) Describe a scenario in which a user process receives a wrong solution when calling *solve*.

(b) Indicate the changes in the given monitor that would be necessary in order to allow the *solve* operation to be safely called from concurrent user processes. Problems should still be solved one at a time.

Question 4.4:

The given coordination strategy does not prevent worker processes from continuing working on problems which have already been solved by other workers. To reduce such unnecessary work, a mechanism for *cancellation* of active workers is requested.

Show how to implement such cancellation by making appropriate changes to the given monitor *ParSolve* and the behaviour of the worker processes. You may assume that all the solving algorithms are iterative, having an outer loop.

Question 4.5:

The functioning of the given monitor *ParSolve* is now to be implemented by a module with the following specification:

> **module** *ParSolve*
>   **op** $solve(X)$ **returns** Y;
>   **op** $get()$ **returns** $(X, integer)$;
>   **op** $result(Y, integer)$;
> **end**

(a) Write a server process for the module *ParSolve* which services the operations by rendezvous in such a way that it functions like the given monitor *ParSolve* as seen from the calling user process and worker processes.

(b) Discuss whether your solution would work correctly if the *solve* operation was called concurrently by several user processes.