UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in:	INF4140 — Models of Concurrency
Day of examination:	20. December 2017
Examination hours:	9 – 13 (4 hours)
This problem set consists of 10 pages.	
Appendices:	None
Permitted aids:	No written or printed material

Please make sure that your copy of the problem set is complete before you attempt to answer anything.

Some general advice and remarks:

- Before you start to solve the problems, take a look at the whole problem set to schedule your time.
- The number of points stated on each part indicates the weight of that part.
- You can make your own clarifications if needed. Please write down any such clarifications.
- Make brief and clear explanations!

Good luck!

(Continued on page 2.)

Problem 1 Shared variables and interference (weight 30)

We consider the setting of shared variables and co-begin statements, and let x denote a shared variable with initial value 0.

1a A first example (weight 8)

Consider the example

$$\textbf{co } x := x + 1 \;||\; x := x + 2; x := x + 2 \textbf{ oc}$$

where each read and write to x is atomic.

1. Does the statement $x := x+1$ satisfy the AMO property (at-most-once property)?

   Solution: No, it has two critical references.

2. How many interleavings are possible for this co-begin program?

   Solution: 4+2 atomic operations, so we get (4+2)!/4!*2!, which is 5*6/2= 15.

3. How many possible final states (i.e., final values of x) are possible here?

   Solution: x may end up with 1,3,4,5 so 4 different final states.

4. Write down a postcondition capturing all possible final values of x assuming the precondition $\{x = 0\}$.

   Solution: $\{x \in \{1, 3, 4, 5\}\}$ or $\{x = 1 \vee x = 3 \vee x = 4 \vee x = 5\}$.

1b Another example (weight 8)

Consider the example

$$\textbf{co } < x := x + 1 > \;||\; < x := x + 2 >; < x := x + 2 > \textbf{ oc}$$

where the assignments are atomic. (We let $< ... >$ denote atomic regions).

1. How many interleavings are possible for this program?

   Solution: (1+2)!/2 = 3

2. How many possible final states (i.e., final values of x) are possible here?

   Solution: just one.

(Continued on page 3.)

3. Write down a postcondition capturing all possible final values of x assuming the precondition $\{x = 0\}$.

   Solution: $\{x = 5\}$

4. Does this example satisfy the interference criterion defined by

$$\mathcal{V}(S_1) \cap \mathcal{W}(S_2) = \mathcal{W}(S_1) \cap \mathcal{V}(S_2) = \emptyset$$

   Solution: No

1c Reasoning about interference (weight 4)

The reasoning rule for the co-begin statement is given by

$$\frac{\{P_i\}\ \mathtt{S_i}\ \{Q_i\} \quad \text{is } \textit{interference free} \text{ for each } i}{\{P_1 \wedge \ldots \wedge P_n\}\ \ \mathbf{co}\ \mathtt{S_1} \parallel \ldots \parallel \mathtt{S_n}\ \mathbf{oc}\ \ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

where the premise is given informally. Write down what the premise means, i.e., what exactly must be verified in order to conclude that $\{P_i\}\ \mathtt{S_i}\ \{Q_i\}$ is *interference free*?

Solution: We must prove

$$\{C \wedge \mathit{pre}_s\}\ \mathtt{s}\ \{C\}$$

for each statement s in any process P and each condition C used (as a pre-, post-, or intermediate condition) in the verification of another process than P. Here pre_s denotes the precondition for s used in the verification of P.

1d Verification (weight 10)

Use the technique of non-interference for Hoare-style program reasoning (the co-statement rule above) to prove

$$\{x = 0\}\ \mathbf{co}\ \ < \mathtt{x} := \mathtt{x} + 1 > \ \ \|\ \ < \mathtt{x} := \mathtt{x} + 2 >; < \mathtt{x} := \mathtt{x} + 2 >\ \ \mathbf{oc}\ \{x = 5\}$$

First specify the pre- and postcondition of each process and any intermediate condition needed. Then specify the verification conditions needed. It suffices to write down the verification conditions. You need not verify these, but you must say if they are provable or not.

Solution:

co $\{x = 0 \vee x = 2 \vee x = 4\}$ x:=x+1 $\{x = 1 \vee x = 3 \vee x = 5\}$
|| $\{x = 0 \vee x = 1\}$ x:=x+2; $\{x = 2 \vee x = 3\}$ x:=x+2 $\{x = 4 \vee x = 5\}$ oc

We need to verify that the given assertions are OK when assuming non-interference and that this gives the desired overall pre- and post-conditions. And third, we need to prove non-interference.

(Continued on page 4.)

All sequential steps are OK for each process, using the assignment axiom.

Overall usage of the co-begin rule gives as precondition:

$$\{(x = 0 \vee x = 2 \vee x = 4) \wedge (x = 0 \vee x = 1)\}$$

which reduces to $\{x = 0\}$, and as postcondition:

$$\{(x = 1 \vee x = 3 \vee x = 5) \wedge (x = 4 \vee x = 5)\}$$

which reduces to $\{x = 5\}$. This gives the desired overall specification.

It remains to prove non-interference: We must prove the four following verification obligations:

1. $\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2 \vee x = 4)\}\texttt{x} := \texttt{x} + 2\{x = 0 \vee x = 2 \vee x = 4\}$ which is OK since $\{x = 0\}\ \texttt{x} := \texttt{x} + 2\ \{x = 2\}$.

2. $\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2 \vee x = 4)\}\texttt{x} := \texttt{x} + 2\{x = 0 \vee x = 2 \vee x = 4\}$ which is OK since $\{x = 2\}\ \texttt{x} := \texttt{x} + 2\ \{x = 4\}$.

3. $\{(x = 0 \vee x = 2 \vee x = 4) \wedge (x = 0 \vee x = 1)\}\ \texttt{x} := \texttt{x} + 1\ \{x = 0 \vee x = 1\}$ which is OK since $\{x = 0\}\ \texttt{x} := \texttt{x} + 1\ \{x = 1\}$.

4. $\{(x = 0 \vee x = 2 \vee x = 4) \wedge (x = 2 \vee x = 3)\}\ \texttt{x} := \texttt{x} + 1\ \{x = 2 \vee x = 3\}$ which is OK since $\{x = 2\}\ \texttt{x} := \texttt{x} + 1\ \{x = 3\}$.

5.$\{(x = 0 \vee x = 2 \vee x = 4) \wedge (x = 4 \vee x = 5)\}\ \texttt{x} := \texttt{x} + 1\ \{x = 4 \vee x = 5\}$ which is OK since $\{x = 2\}\ \texttt{x} := \texttt{x} + 1\ \{x = 3\}$.

6. $\{(x = 0 \vee x = 1) \wedge (x = 1 \vee x = 3 \vee x = 5)\}\texttt{x} := \texttt{x} + 2\{x = 1 \vee x = 3 \vee x = 5\}$ which is OK.

7. $\{(x = 2 \vee x = 3) \wedge (x = 1 \vee x = 3 \vee x = 5)\}\texttt{x} := \texttt{x} + 2\{x = 1 \vee x = 3 \vee x = 5\}$ which is OK.

Problem 2 Monitors (weight 33)

In this task you should program a monitor that allows N processes to synchronize by calling a monitor procedure, *sync*.

The *sync* procedure should let the calling process wait until N processes have called *sync*. Thus the processes that are calling *sync* should all wait as long as there are less than N waiting processes, and when the Nth process calls *sync*, they should all continue.

2a Programming (weight 10)

Program a monitor *SyncControl* with a procedure *sync* as described above. The monitor should allow multiple rounds of synchronization, possibly

(Continued on page 5.)

involving different processes each time. Assume a "signal-and-continue" discipline.

Solution:

```
monitor SyncControl {
cond s  // synchronize control
Nat c  :=0 // counter

procedure sync { c:= c+1;
 if c<N then wait(s) else c:= 0; signal_all(s) fi }
}
```

2b Monitor invariant (weight 5)

Formulate a monitor invariant for your program. The invariant may refer to the length of the queue associated with any condition variables.

Solution:

$$\#s = 0 \Leftrightarrow c = 0$$

2c Steps in verifying a monitor invariant (weight 4)

Given a monitor M, what are the main steps in verifying a monitor invariant, given that the invariant is I_M and that there are k monitor procedures?

Solution: Prove that I_M holds after the initialization code, and prove that I_M is maintained by each of the k monitor procedures.

2d Axioms for signal and wait (weight 4)

Consider the following axioms for signal and wait, where $\#cv$ represents the length of the waiting queue for cv.

$$\{I[\#cv + 1/\#cv]\}\ \texttt{wait(cv)}\ \{I\}$$

$$\{((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P[\#cv - 1/\#cv])\}\ \texttt{signal(cv)}\ \{P\}$$

$$\{P[0/\#cv]\}\ \texttt{signal\_all(cv)}\ \{P\}$$

Explain the meaning of these axioms, and in particular justify the effects on $\#cv$.

(Continued on page 6.)

2e Verification of the invariant (weight 10)

Verify the monitor invariant for your program using Hoare logic. You may use the above axioms for signal and wait.

Solution: We may first observe that $c \geq 0$ holds everywhere, since c is only increased by 1 or reset to 0. Similarly, we know that $\#s \geq 0$.

We must prove that the implementation of *sync* maintains the invariant. Using the invariant as a postcondition we calculate the precondition:

```
{c+1<N =>(#s+1=0 <=> c+1=0)} -- which reduces to true
                                since #s+1>0 and c+1>0
c:= c+1; {c<N => (#s+1=0 <=> c=0)} -- since else part is true
if c<N then   {#s+1=0 <=> c=0}
wait(s)       {#s =0 <=> c=0}
else          { 0 =0 <=> 0=0} -- which reduces to true
c:= 0;        { 0 =0 <=> c=0}
signal_all(s) {#s =0 <=> c=0}
fi            {#s =0 <=> c=0}
```

Must prove that the invariant implies the precondition, which is trivial.

Problem 3 Weak Memory Models (weight 15)

Consider the following Go program:

```
1  package main
2
3  var a string = "a";
4  var done bool = false;
5
6  func setup() {
7          a = "hello, world"
8          done = true
9  }
10
11 func main() {
12         go setup()
13         for !done {  }
14         print(a)
15 }
```

(Continued on page 7.)

3a Behavior (weight 5)

What are the possibles values outputted by the **print** on line 12? Please justify your answer.

Solution: It can print `a` and `hello, world`. That is because `setup` runs asynchronously, and from `main`'s perspective, the assignment to the shared variable `a` on line 7 and the assignment to `done` on line 8 can appear to be executed out of program order.

3b Strong and a weak memory models (weight 5)

What are the differences between a strong and a weak memory model?

Solution: A memory model dictates what values a read from memory can return. There are two main flavors of memory models: strong (or sequentially consistent) and weak (or relaxed) memory.

Strong memory behaves in a sequentially consistent manner, meaning, "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" (Leslie Lamport, *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*)

Weak memory models (also known as relaxed memory models) allow for the reordering of events. Different kinds of weak memory are obtained depending on which reorderings are allowed.

3c Go memory model (weight 5)

What kind of memory model does Go implement?

Solution: Go implements a type of relaxed memory: it allows for reordering of reads and writes and, "because of reordering, the execution order observed by one go routine may differ from the order perceived by another" (Go's memory model, https://golang.org/ref/mem)

Problem 4 Asynchronous systems (weight 22)

Imagine a job announcement system, where jobs are announced by an agent, JobHandler, such that each agent takes care of exactly one job announcement as well as the hiring process, resulting in a "hire" message or a "nohire" message. Job applicants are represented by agents that send job application messages to the JobHandler agent. The JobHandler will not

(Continued on page 8.)

receive applications after the job deadline, given by a "finish" message from the employer. The actions are described in more details below:

> A JobHandler agent should start by waiting for a *newjob* message from an employer E and then it should receive a number of *application* messages from job applicant agents, until it receives a *finish* message from the employer E. The JobHandler should then send an *offer* message to the first applicant, if any, that applied (thus the identity of the applicants should be stored in a queue Q), and wait for an *accept* or *pass* message from that applicant. (We here ignore ranking or the applicants.) In the former case it sends a *hire* message to the employer E with the agent identity as a parameter, in the latter case the JobHandler sends an *offer* message to the next applicant in the queue, and so on, until either a *hire* message is generated or there are no more applicants in the queue, in which case a *nohire* should be sent to the employer E. The same applicant may send more than one *application* message, but should get at most one *offer* message.

4a Programming of the job handler agent (weight 10)

Program the JobHandler agent using the language for asynchronous agents with send and receive statements, in addition to assignments, if, non-deterministic choice, local calls, recursion or loops, as required. For queues (Q), you may use the following statements:

- $Q := append(Q, X)$ for appending an element X to the queue Q,
- $X := first(Q)$ for extracting the first element X of the queue Q,
- $Q := rest(Q)$ for removing the first element of Q,
- $Q = empty$ for testing emptiness of Q,
- X *in* Q for testing membership in Q.

Solution:

```
agent JobHandler {
 Agent E        // a variable local to the JobHandler
 List[Agent] Q  // a variable local to the JobHandler

 // main program
 { loop (await E?newjob; handle) endloop }

 // local method ‘‘handle’’
```

(Continued on page 9.)

```
handle { Agent X
 (wait X?application;
  if not X in Q then Q:=append(Q,X) fi; handle)
 []
 (await E:finish; makeoffer)}

// local method ``makeoffer''
makeoffer { Agent X
 if Q=empty then send E:nohire
 else X:= first(Q); Q:= rest(Q); send X:offer;
  (await X:accept; send E:hire(X); Q:= empty)
  []
  (await X:pass; makeoffer) endif }
could remove X from Q here
}
```

Could use loops in stead of recursion.

4b Alphabet (weight 3)

What is the alphabet of the JobHandler agent? You may let J denote the JobHandler agent.

Solution:

$$
\begin{array}{l}
(E \downarrow J : newjob) \\
(E \downarrow J : finish) \\
(X \downarrow J : application) \\
(X \downarrow J : accept) \\
(X \downarrow J : pass) \\
(J \uparrow E : hire(X)) \\
(J \uparrow E : nohire) \\
(J \uparrow X : offer)
\end{array}
$$

4c A function over the history (weight 5)

Define a function *queue* over the local history that calculates the current value of the queue Q. Thus $Q = queue(h)$ should be an invariant of the JobHandler agent, holding in all states.

Solution: We define a function *queue* over the local history, inductively:

$$
\begin{array}{lcl}
queue(\epsilon) & = & \epsilon \\
queue(h; (X \downarrow J : application)) & = & \textbf{if } X \in queue(h) \textbf{ then } queue(h) \\
 & & \textbf{else } append(queue(h), X) \\
queue(h; (X \downarrow J : pass)) & = & queue(h) - \{X\} \\
queue(h; (E \downarrow J : newjob)) & = & \epsilon \\
queue(h; others) & = & queue(h)
\end{array}
$$

(Continued on page 10.)

4d Invariant (weight 4)

Formulate a local history invariant for the JobHandler agent ensuring that whenever a "hire" event happens, say $(J \uparrow E : hire(X))$, then X is in the queue of the JobHandler agent (as defined above).

Solution: We define an invariant OK over the local history, inductively:

$$\begin{array}{lcl} OK(\epsilon) & = & true \\ OK(h; (J \uparrow E : hire(X))) & = & OK(h) \wedge X \in queue(h) \\ OK(h; others) & = & OK(h) \end{array}$$

where J stands for the JobHandler. Alternatively, one could write

$$(h'; ((J \uparrow E : hire(X)))) \leq h \Rightarrow X \in queue(h')$$