UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in:	INF4140 — Models of Concurrency
Day of examination:	7. December 2016
Examination hours:	14.30 – 18.30

This problem set consists of 12 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.
- The set has 100% without the last question. That question may give up to 5% bonus points.
- 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 General Questions (weight 25)

1a The at-most-once property (weight 5)

Define the at-most-once property and explain what it can be used for.

Solution: See book.

1b Interference (weight 5)

Define the concept of *interference* between two processes.

Solution: One process interferes with another it is executes an assignment that invalidates an assertion in the the other process. (See page 63 in the textbook)

1c Signal and continue (weight 5)

Write down the Hoare rules for signal and wait, assuming the "signal and continue" discipline. Explain the rules briefly.

Solution:

$$\{I\}wait(c)\{I\}$$

$$\{P\}signal(c)\{P\}$$

Wait needs to maintain the invariant, since execution does not continue. Signal has no local effect in partial correctness since execution continues.

1d Signal and wait (weight 5)

Explain briefly why the rules of the previous question are not suitable for signal and wait with the "signal and wait" discipline.

Solution: Signal also needs to maintain the invariant, since execution does not continue.

$$\{I \wedge b_c\}signal(c)\{I\}$$

$$\{I\}wait(c)\{I \wedge b_c\}$$

Here b_c is the condition associated with c. We may rely on this condition after wait assuming signalling ensures it (using the rules above).

1e RPC and rendezvous (weight 5)

Explain briefly the main differences between RPC and rendezvous.

(Continued on page 3.)

Solution: For RPC there is no waiting on the callee side, since a new process is started. The callee side may need to deal with shared variable concurrency is more than one process can be active at the same time. With rendezvous, caller or callee may need to wait.

Problem 2 Shared variables: Pizza making (weight 30)

We consider here the following description of a synchronization problem:

> A pizza is made in three steps: making the dough, cutting the ingredients, and grating the cheese. First, a dough maker will make enough dough for a pizza. When it is finished, a cutter tool will prepare ingredients in suitable pieces and put them on top of the dough. Finally, a grater will grate the cheese on top. Note that making such a simple pizza does not involve concurrency, but many pizzas may be made concurrently.



2a Semaphore Solution (weight 10)

Provide an implementation of the synchronization part of the cooking processes by using semaphores. Extend the sketch below by replacing the dots (where needed) by suitable code. Remember to declare and initialize the semaphores.

```
process DoughMaker[i=1 to N]{
  while (true){
    ...
    ‘‘make dough’’
    ...
  }
}

process Cutter[i=1 to N]{
  while (true){
    ...
    ‘‘cut the ingredients’’
    ...
  }
}
```

(Continued on page 4.)

```
process Grater[i=1 to N]{
  while (true){
    ...
    ‘‘grate cheese’’
    ...
  }
}
```

Solution:

```
sem dough = 0,
sem Ing = 0;

process DoughMaker[i=1 to N]{
while (true){
‘‘make dough’’
    V(dough);
}
}

process Cutter[i=1 to N]{
while (true){
    P(dough);
‘‘cut the ingredients’’
    V(Ing);
}
}

process Grater[i=1 to N]{
while (true){
    P(Ing);
‘‘grate cheese’’
}
}
```

2b Deadlocks (weight 4)

Does your program (from problem 2a) contain deadlocks? Explain.

Solution: The solution given here is deadlock free. Each dough maker is not depending on any other process and may increase the value of `dough`. Each Cutter is only dependent on the `dough` semaphore, which may always be increased. And a similar argument may be made for the Grater.

(Continued on page 5.)

2c Monitor Solution (weight 10)

Implement a solution to the pizza making problem again by using a monitor for synchronization. Remember that several pizzas might be made concurrently. Apart from the monitor, there should be processes for making dough, ingredients, and cheese, similar to those of problem 2a.

Solution:

```
process DoughMaker[i=1 to N]{
‘‘make dough’’
  Chef.doneDough();
}

process Cutter[i=1 to N]{
 Chef.startIng();
‘‘cut the ingredients’’
 Chef.doneIng();
}

process Grater[i=1 to N]{
 Chef.startGrt();
‘‘grate cheese’’
}
}


monitor Chef{
  int dough = 0, Ings = 0;
  cond cutIngs, grtcheese;

  procedure doneDough(){
    dough = dough + 1;
    signal(cutIngs);
  }

  procedure doneIngs(){
    Ings = Ings + 1;
    signal(grtcheese);
  }

  procedure startIngs(){
    while(dough < 1){
      wait(cutIngs);
    }
    dough = dough - 1;
```

(Continued on page 6.)

```
  }

  procedure startchees(){
    while(Ings < 1){
      wait(grtcheese);
    }
    Ings = Ings - 1;
  }
}
```

2d Fairness (weight 6)

Does your solution from problem 2c allow sneaking? Explain your answer. If your solution allows sneaking then explain how you can prevent it.

Solution: (Page 210 of Andrews): Sneaking happens when newly arrived processes get access before processes that are waiting on the queues of the monitor. An argument about not increasing the counters unless the queues are empty should be presented. The students should also change from a while loop to an if test when checking if a process should wait or not.

```
monitor Clerk { % fair
  int dough = 0, Ings = 0;
  cond cutIngs, grtcheese;

  procedure doneDough(){
    if(empty(cutIngs)){
      dough = dough + 1; }
    else  {
       signal(cutIngs);}
  }

  procedure doneIngs(){
    if(empty(grtcheese)){
      Ings = Ings + 1; }
    else  {
       signal(grtcheese);}
  }

  procedure startIngs(){
    if(dough < 1){
      wait(cutIngs); }
    else  {
       dough = dough - 1; }
  }
```

(Continued on page 7.)

```
  procedure startchees(){
    if(Ings < 1){
      wait(grtcheese); }
    else  {
       Ings = Ings - 1; }
  }
}
```

Problem 3 Asynchronous Communication: Pizza making (weight 20)

We here consider asynchronous message passing using the language with **send** and **await** statements.[1]

Consider the following implementation of a `Cutter` agent (called `Cu`):

```
D : Agent;  // assumed initialized to a DoughMaker
G : Agent;  // assumed initialized to a Grater

while true do
  await D:startIngs;
  // cut the ingredients
  send G:startchees
od
```

You may assume that no communication occurs while cutting.

3a Agent structure (weight 3)

Agent `Cu` here cooperates with fixed agents `D` and `G`. Would it be easy to generalize the implementation so that `Cu` may cooperate with any agents `D` and `G`? Explain briefly.

Solution:

Not so easy: We may receive from any agent (using **await** D?startIngs), but the language has no primitive for sending to any agent.

3b Events (weight 3)

What are the local events of `Cu`, using the notation of the lectures?

[1] We write **send** $A:m$ for sending the message m to the agent A, and **await** $A:m$ for receiving the message m from the agent A.

(Continued on page 8.)

Solution:

$D \downarrow Cu$: `startIng` and $Cu \uparrow G$: `startchees`

3c Program Analysis (weight 10)

Consider the following loop invariant for the Cutter:

$$\#(h/(\downarrow\texttt{startIng})) = \#(h/(\uparrow\texttt{startchees}))$$

where h is the local history of `Cu`, the projection $h/(\downarrow\texttt{startIng})$ restricts h to `startIng` receive events, the projection $h/(\uparrow\texttt{startchees})$ restricts h to `startchees` send events, and $\#(a)$ returns the length of the sequence a.

Explain why this is a suitable loop invariant, and use Hoare Logic to verify this loop invariant for the `Cutter` implementation given above.

Solution:

Taking the invariant as the postcondition for the loop body, a precondition for the loop body may be derived by backward construction:

$\{\#(h/(\downarrow\texttt{startIng})) + 1 = \#(h/(\uparrow\texttt{startchees})) + 1\}$
await `G:startIng`
$\{\#(h/(\downarrow\texttt{startIng})) = \#(h/(\uparrow\texttt{startchees})) + 1\}$
`// cut the ingrediants`
$\{\#((h; Cu \uparrow G : \texttt{startchees})/(\downarrow\texttt{startIng}))$
$\quad = \#((h; Cu \uparrow G : \texttt{startchees})/(\uparrow\texttt{startchees}))\}$
sent `G:startchees`
$\{\#(h/(\downarrow\texttt{startIng})) = \#(h/(\uparrow\texttt{startchees}))\}$

We are then left with the trivial verification condition:

$$\#(h/(\downarrow\texttt{startIng})) = \#(h/(\uparrow\texttt{startchees})) \Rightarrow$$
$$\#(h/(\downarrow\texttt{startIng})) + 1 = \#(h/(\uparrow\texttt{startchees})) + 1$$

3d History invariant (weight 4)

Explain why the loop invariant is not a prefix-closed history invariant. Formulate a prefix-closed history invariant (by weakening the loop invariant).

Solution:

$$\#(h/(\downarrow\texttt{startIng})) \leq \#(h/(\uparrow\texttt{startchees})) \leq \#(h/(\downarrow\texttt{startIng})) + 1$$

(Continued on page 9.)

Problem 4 Objects and Agents (weight 25)

We assume a number of pizza *makers* preparing pizzas in parallel, and a shared oven for baking the pizzas. The baking is handled by a shared pizza *baker* of interface *Baker*, given below. A new baking request is made by the operation *bake*. Since there is only one oven, there will in general be a queue (with FIFO ordering) of baking requests. The *baker* object will bake the pizzas in the oven one by one. Since the pizza makers can be impatient, we allow them to ask the baker object for the baking status, letting the operation *check* return (a reference to) the pizza maker currently being served.

You are given the following Creol implementation of the pizza baker:

```
interface Maker begin ... end // the interface of the pizza makers

interface Oven begin
  op inoven() // bake in oven until ready baked
end

interface Baker begin
 with Maker
  op bake(out n:Nat) // to place a new order for baking a pizza
     // then bake the pizza in the oven,
     // and return the order number, when the pizza is finished
  op check(out current:Maker)
     // return the pizza maker currently being served
end

class BAKER (o:Oven) implements Baker begin
 var last : Nat=0; // last order
 var next : Nat=1; // next order
 var serving : Maker = null; // the pizza maker being served

 with Maker
  op bake(out myorder:Nat) ==
       last:=last+1; myorder:=last;
       await next=myorder;
       serving:= caller;
       await o.inoven();
       next:= next+1; serving:= null

  op check(out current:Maker) == current:= serving
end
```

4a About the Creol implementation (weight 5)

1. Explain the purpose of the cointerface(s) used in the code.

(Continued on page 10.)

2. Explain the purpose of the keyword **await** in front of the call to *inoven*, and what it implies.

3. Explain whether this use of **await** is a good idea or not in this example.

Solution: The cointerface allows us to use the caller as a reference of interface Maker. and then to ensure that next of check is of that interface.

```
await o.inoven();
```

is a non-blocking call, to allow other activity while a pizza is in the oven. So a good idea!

4b Re-implementation (weight 10)

Re-implement the pizza baker in the setting of asynchronous agents with message passing (send and receive). The agent should correspond (as much as possible) to an object of class `BAKER`, letting each *method call* and each *method reply* correspond to a message. For a method m, you may let m_reply denote the corresponding reply message. In particular, the agent should serve all incoming messages without being blocked, and there should be at most one uncompleted call to `inoven`.

You may use guarded receive statements of the form

await *message* **when** *guard*

where *message* is of the form $A : m(x)$ when receiving message m from a specific agent A, or $A?m(x)$ when receiving message m from an unknown agent A, and *guard* is a boolean condition referring to the parameters of m and agent variables. The execution of this statement requires that the guard holds. Thus you may rely on the guard immediately after the statement.

Solution: Agent Baker can be implemented as follows:

```
// agent variables
var m : Maker;
var n : Nat=0; // next order, i.e. the one being served
var l : Nat=1; // last order
var s : Agent=null; // the one being served

//agent body
while true do
 (  await m?bake(); l:=l+1; send myself:continue(m,l);
 [] await myself:continue(a,x) when x=n; send o:inoven(); s:=a;
 [] await o:inoven_reply(); send s:bake_reply(n); n:=n+1; s:=null;
 [] await m?check(); send m:check_reply(s);
 ) od
```

(Continued on page 11.)

Here *myself* could be the name of the agent, i.e., *Baker*.

4c History functions (weight 5)

Define the queue of unfinished baking requests as a function of the local history of the pizza *Baker* agent, using the code you made in 4b, letting each element in the queue be represented by the identity of calling *Maker* agent. Define also the next order to be finished as a function of the local history of the baker agent.

First, specify the events local to the baker agent.

Solution:

Here (and below) we use Creol histories and events. (for the agent solution we also get the continue events, which in a way gives more explicit information.

$$\begin{array}{lcl} qu(empty) & = & empty \\ qu(h;(c{\twoheadrightarrow}bake)) & = & qu(h);c \\ qu(h;(c{\leftarrow}bake)) & = & rest(qu(h)) \\ qu(h;others) & = & qu(h) \end{array}$$

$$\begin{array}{lcl} next(empty) & = & 1 \\ next(h;(c{\leftarrow}bake)) & = & next(h)+1 \\ next(h;others) & = & next(h) \end{array}$$

$$last(h) \quad = \quad \#(h/(\twoheadrightarrow bake))$$

Next may also be defined by

$$next(h) = \#(h/(\leftarrow bake)) + 1$$

Serving (when not null) is given by

$$serving(h) = qu[next(h)]$$

4d Invariant (weight 5)

We would like to verify that there is at most one pizza in the oven at any time, i.e., that there is at most one uncompleted *inoven* call to the oven *o*.

Formulate an invariant for the pizza baker agent such that the invariant implies this condition. Explain briefly (informally) why the suggested invariant actually is an invariant. The invariant may refer to the history, and you may use the functions from the previous question.

Solution:

Informal answer:

(Continued on page 12.)

A call to *inoven* is only made when n is the number of completed inoven calls/ completed bake calls, a new inoven call is made for the ith order when $i = n + 1$. Therefore there is at most one uncompleted inoven call.

Invariant (using prefix of reg expr.):

$h/\{\twoheadrightarrow bake, \leftarrow bake, \rightarrow inoven, \twoheadleftarrow inoven\}$
prefixof $(\twoheadrightarrow \text{bake}); ((\twoheadrightarrow \text{bake})^*; \rightarrow \text{o.inoven}; (\twoheadrightarrow \text{bake})^*; \twoheadleftarrow \text{o.inoven}; \leftarrow \text{bake})^*$

This implies that there is at most one uncompleted inoven call to o.

4e **Verification** (weight 5 bonus points)

Verify the agent invariant from the previous question using Hoare Logic.