UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in:	INF9145 — Models of Concurrency
Day of examination:	17. December 2012
Examination hours:	14.30 – 18.30

This problem set consists of 15 pages.

Appendices:	Inference Rules for program verification
Permitted aids:	None

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

Some general advice and remarks:

- This problem set consists of three independent problems and one optional extra problem which gives you extra points.
- The points from problems one, two and three sum up to a total of 100 points. The number of points stated on each part indicates the weight of that part.
- Problem four is optional and can give you extra points on top of your final score.
- Use your time wisely and take into consideration the weight of each question.
- You should read the whole problem set before you start solving the problems.
- You can make your own clarifications if you find the examination text ambiguous or imprecise. Such clarifications must be written clearly in the delivered answer.
- Make short and clear explanations!

Good Luck and Merry Christmas!

(Continued on page 2.)

Problem 1 Shared variables: Santa's Factory (weight 40)

The Snow-globe Production:

We here consider the following description of the snow-globe production as a synchronization problem:

In Santa's Factory the elves are going to start the production of snow-globes. They divide the production in different tasks:
Some elves are in charge of collecting the different materials and organizing them in different groups (each group contains glass, fake snow and ceramic).

Some other elves are in charge of assembling the snow-globes and for that they take a group of materials and hand-make a snow-globe. Other elves are in charge of hand-making boxes for the different snow-globes. Finally some elves are in charge of wrapping the snow-globes in the boxes.

The assembling of a snow-globe has the following precedence graph:

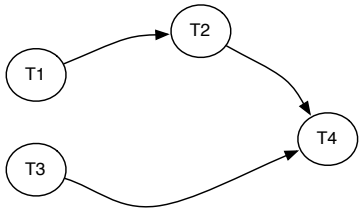

Figure 1: Precedence graph

Where:
T1: collect a group of materials
T2: hand-make a snow-globe
T3: hand-make a box
T4: wrap a snow-globe

Note that many snow-globes may be assembled concurrently.

1a Semaphore Solution (weight 13)

Implement the synchronization part of the snow-globe production by extending the sketch below. Fill in code for the dots where needed. Use semaphores for synchronization. Remember to declare and initialize the semaphores.

```
process MaterialCollector[i=1 to N]{
    while (true){
    ...
    ‘‘collect a group of materials (glass, fake snow and ceramic)’’
    ...
  }
}
```

(Continued on page 3.)

```
process GiftAssembler[i=1 to N]{
  while (true){
    ...
    ‘‘hand-make a snow-globe’’
    ...
  }
}

process BoxMaker[i=1 to N]{
  while (true){
    ...
    ‘‘hand-make a box’’
    ...
  }
}

process GiftWrapper[i=1 to N]{
  while (true){
    ...
    ‘‘wrap a snow-globe’’
    ...
  }
}
```

Solution:

```
sem materials = 0, gift = 0, box = 0;

process MaterialCollector[i=1 to N]{
    while (true){
    ‘‘collect a group of materials (glass, fake snow and ceramic)’’
    V(materials);
  }
}

process GiftAssembler[i=1 to N]{
  while (true){
    P(materials);
    ‘‘hand-make a snow-globe’’
    V(gifts);
  }
}

process BoxMaker[i=1 to N]{
  while (true){
```

(Continued on page 4.)

```
    ‘‘hand-make a box’’
     V(boxes);
  }
}

process GiftWrapper[i=1 to N]{
  while (true){
    P(gifts);
    P(boxes);
     ‘‘wrap a snow-globe’’
  }
}
```

1b Safety (weight 3)

Explain briefly why your implementation (from exercise 1a) is safe, i.e., why a snow-globe is only wrapped in a box after both the box and the snow-globe have been made, and why a `GiftAssembler` will only start making a snow-globe when there is a group of materials that he can use.

Solution: The processes will just wait for their baton which is controlled by the V and P operations in the different semaphores. GiftAssembler wait for available groups of materials and GiftWrapper wait for both available boxes and available snow-globes

1c Binary semaphores (weight 3)

Does your solution (from exercise 1a) use binary semaphores? Explain briefly.

Solution: This solution does not use binary semaphores because the semaphores materials, gift and box can increase depending on the speed of the processes, for example if the process BoxMaker is fast the value of the semaphore box will increase every time it finish a box, in other words the value of the semaphore box reflects the number of available boxes.

(Continued on page 5.)

Gift Storage:

In Santa's factory there are some elves in charge of organizing the gifts for transporting into the storage room. There are N `GiftOrganizer` elves and one `Transporter` elf. They share a common sledge. Each `GiftOrganizer` elf repeatedly gathers one gift and puts it into the sledge, the sledge can hold **G** gifts and it is initially empty.
When the sledge is full the `Transporter` elf moves the sledge with the gifts into the storage room (which means that the `GiftOrganizer` elves have to take a break until the sledge is back), downloads the gifts in one empty bag, closes the bag, and returns with an empty sledge.

1d Monitor Solution (weight 15)

Given the following implementation of the `GiftOrganizers` and `Transporter` elves as processes:

```
process GiftOrganizer[i=1 to N] {
  while (true)  call Gift_Storage.put();
}

process Transporter{
  while (true)  call Gift_Storage.transport();
}
```

Extend the sketch below of the monitor. Fill in code for the dots where needed. Use the Signal and Continue discipline and assume there is an infinite supply of gifts.

code/giftstorage-skeleton.code

```
monitor Gift_Storage {
  int g = G;  # capacity of the sledge
  int counter = 0; # number of gifts in the sledge
  ...
  procedure put() {
    ...
    ‘‘Put gift into the sledge’’
    ...
   }

  procedure transport() {
    ...
    ‘‘move sledge into the storage room, download gifts and return’’
```

(Continued on page 6.)

```
    ...
  }
}
```

Solution:

code/giftstorage1.code

```
process GiftOrganizer[i=1 to N] {
  while (true)  call Gift_Storage.put(); }

process Transporter{
  while (true) call Gift_Storage.transport(); }

monitor Gift_Storage {
  int g = G;  # capacity of the sledge
  int counter = 0; # number of gifts in the sledge
  cond queue; # waiting queue

  procedure put() {
    while(counter == g) wait(queue);
    ''Put gift into the sledge''
    counter = counter + 1;
    signal_all(queue);
  }

  procedure transport() {
    while(counter < g) wait(queue);
    ''move sledge into the storage room, download gifts and return''
    counter = 0;
    signal_all(queue);
  }
}
```

Alternative solution:

```
process GiftOrganizer[i=1 to N] {
  while (true)  call Gift_Storage.put(); }

process Transporter{
  while (true)  call Gift_Storage.transport(); }

monitor Gift_Storage {
  int g = G;  # capacity of the sledge
  int counter = 0; # number of gifts in the sledge
  cond queue0; # waiting queue for GiftOrganizerts
```

(Continued on page 7.)

```
  cond queueT; # waiting queue for Transporter

  procedure put() {
    while(counter >= g) wait(queue0);
    ‘‘Put gift into the sledge’’
    counter = counter + 1;
    if(counter == g) signal(queueT);
  }
  procedure transport() {
    while(counter < g) wait(queueT);
    ‘‘move sledge into the storage room, download gifts and return’’
    counter = 0;
    signal_all(queue0);
   }
 }
```

1e Java (weight 3)

Can your monitor solution from exercise 1d easily translate into a monitor in Java using the built-in statements `wait`, `notify` and `notifyAll`? Explain briefly. (The Java implementation code is not needed)

Hint: How many condition variables are you using in your solution from exercise 1d?

Solution: It will depend on how many condition variables the solution uses, In Java there is only one implicit condition variable per object; in the case of the first implementation it will easily translate into java code because it uses only one condition variable so it will only need some syntax adjustment. In the case of the alternative solution, it is not an straight forward translation because it uses more than one condition variable.

```
public class Gift_Storage {
    int g = G;
    int counter = 0;


    public synchronized void put() throws InterruptedException {
        while(counter==g) wait();
        ++count;
        notifyAll();
    }

    public synchronized void transport() throws InterruptedException {
        while(count<g) wait();
        count = 0;
```

(Continued on page 8.)

```
            notifyAll();
        }
    }
```

1f Signaling Disciplines (weight 3)

What is the difference between the Signal and Continue discipline and the Signal and Wait discipline? Does your monitor implementation from exercise 1d work correctly with the Signal and Wait discipline? Explain briefly.

Solution:

Signal and Continue: The signaler continues and the signaled process execute at some later point.

Signal and Wait: The signaler waits until some later time and the signaled process executes now.

Fig 5.1 in page 209 from Andrews book.

The proposed implementation from 1d does not work correctly with SW because it uses signal_all(cv) and this operation is not well-formed for the SW discipline.

Problem 2 Asynchronous Communication: Santa's Factory (weight 40)

We here consider asynchronous message passing using the language with **send** and **await** statements, and the following variation of the snow-globe production from Problem 1.
In this case, we have a total of four agents in the system:

- One `MaterialCollector` agent,
- One `GiftAssembler` agent,
- One `BoxMaker` agent, and
- One `GiftWrapper` agent.

The agents follow the precedence graph from Figure 1 in page 2, which means: The `MaterialCollector` notifies the `GiftAssembler` agent every time it finishes to collect a group of materials, the `GiftAssembler` agent notifies the `GiftWrapper` agent every time it finishes to hand-make a snow-globe, and the `BoxMaker` agent also notifies the `GiftWrapper` agent every time it finishes to hand-make a box.

(Continued on page 9.)

Consider the following implementation of the `GiftAssembler` agents:

```
X : Agent;  // assumed initialized to the MaterialCollector
Y : Agent;  // assumed initialized to the GiftWrapper

while true do
  await X:materialIsCollected;
  // ''hand-make a snow-globe''
  send Y:giftIsAssembled;
od
```

You may assume that no communication occurs during the snow-globe production.

2a Loop invariant (weight 4)

Consider a `GiftAssembler` agent A implemented as above.
The function *isBeingAssembled* is defined over the local history of A to calculate the number of snow-globes which are in production but not complete.

$$isBeingAssembled(\varepsilon) = 0$$
$$isBeingAssembled(h; X \downarrow A : materialIsCollected) = isBeingAssembled(h) + 1$$
$$isBeingAssembled(h; A \uparrow Y : giftIsAssembled) = isBeingAssembled(h) - 1$$

Formulate a loop invariant of A using the *isBeingAssembled* function.

Solution: $isBeingAssembled(h) = 0$

2b Program Analysis (weight 13)

Use Hoare Logic to verify your loop invariant from exercise 2a (In the Appendix you can find a list of rules for program verification), and explain briefly why this is a suitable loop invariant.

Solution:

while *true* **do** $\{isBeingAssembled(h) = 0\}$
$\{isBeingAssembled(h) + 1 - 1 = 0\}$
$\{isBeingAssembled(h; X \downarrow A : materialIsCollected) - 1 = 0\}$
await $X : materialIsCollected$;
$\{isBeingAssembled(h) - 1 = 0\}$
`// ''hand-make a snow-globe''`
$\{isBeingAssembled(h; A \uparrow Y : giftIsAssembled) = 0\}$
send $Y : giftIsAssembled$;
$\{isBeingAssembled(h) = 0\}$
od

(Continued on page 10.)

We are then left with the trivial verification condition:

$$\{isBeingAssembled(h) = 0\} \Rightarrow \{isBeingAssembled(h) + 1 - 1 = 0\}$$

2c History Invariant (weight 7)

Using the *isBeingAssembled* function formulate an invariant for A over its local history h, which always holds, and show formally or informally that this invariant holds after each interaction point in the implementation.

Solution: $0 \leq isBeingAssembled(h) \leq 1$
For each iteraction statement, we can do by proving $Q \Rightarrow I_h$, where I_h is the history invariant and Q is postcondition for the interaction statement in the proof outline above.

For **await** X : $materialIsCollected$, we have:

$$\{isBeingAssembled(h) - 1 = 0\} \Rightarrow \{0 \leq isBeingAssembled(h) \leq 1\}$$

For **send** Y : $giftIsAssembled$, we have:

$$\{isBeingAssembled(h) = 0\} \Rightarrow \{0 \leq isBeingAssembled(h) \leq 1\}$$

2d Implementation (weight 12)

Provide an implementation of the `GiftWrapper`. This agent should repeatedly try to wrap the snow-globes, but it must wait for the arrival of one snow-globe and one box before wrapping it. The implementation should allow `GiftWrapper` to receive a snow-globe and a box in a random order. Extend the program sketch below. Fill in code for the dots where needed.

```
while true do
  ...
  //  ‘‘wrap a snow-globe’’
  ...
od
```

Solution:

```
X : Agent;  // assumed initialized to the GiftAssembler
Y : Agent;  // assumed initialized to the BoxMaker

while true do
  (await X:giftIsAssembled;
    await Y:boxIsMade)
```

(Continued on page 11.)

```
    []
  (await Y:boxIsMade;
    await X:giftIsAssembled)

  //  ‘‘wrap a snow-globe’’
od
```

2e Local Histories (weight 4)

Give two examples of possible local histories of the `GiftWrapper` agent.

Solution:

$X \downarrow A : boxIsMade;\ Y \downarrow A : giftIsAssembled$

$Y \downarrow A : giftIsAssembled;\ X \downarrow A : boxIsMade$

Problem 3 ABS Analysis (weight 20)

Consider the following ABS implementation of a class `StorageRoom`, providing the services for elves to store and remove gifts:

```
class StorageRoom {
  Int counter;

  // the number of gifts
  {counter := 0}

  Void store(Int x) { counter := counter + x }

  Int remove(Int x) { await counter >= x;
                      counter := counter - x; return x }
}
```

We here consider the following class invariant I for class `StorageRoom`:

$$I \triangleq (\texttt{counter} = total(\mathcal{H})) \wedge (\texttt{counter} >= 0)$$

where $\mathcal{H}$ is the local history variable of `StorageRoom` and *total* is a history function (see below).

3a History Function (weight 5)

Define inductively the history function $total : Seq[Ev] \rightarrow Int$ which computes the number of current gifts in the storage room. **Hint:** The total function should be defined over the events shown in Figure 2.

(Continued on page 12.)

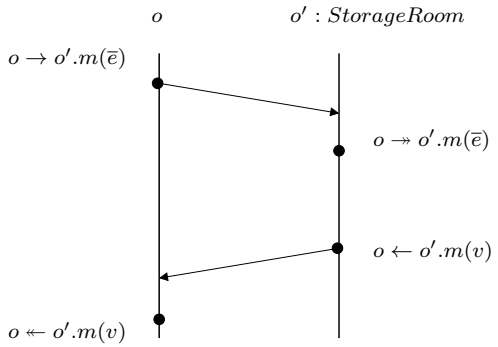

Figure 2: A method call cycle, where object o calls a method m on object o'. The arrows indicate message passing, and the bullets indicates **events**. Remark the events on the left hand side are visible to o, whereas the events on the right hand side are visible to o'.

Solution:

$$\begin{array}{ll} \mathit{total}(\epsilon) & \triangleq 0 \\ \mathit{total}(h \vdash \_ \leftarrow \texttt{this.store(x)}) & \triangleq \mathit{total}(h) + x \\ \mathit{total}(h \vdash \_ \leftarrow \texttt{this.remove(x)}) & \triangleq \mathit{total}(h) - x \\ \mathit{total}(h \vdash \textbf{others}) & \triangleq \mathit{total}(h) \end{array}$$

3b Initialization (weight 5)

Argue informally that the invariant I is established by the initialization code of the class `StorageRoom`.

Solution:

Ignoring the initial object creation event on $\mathcal{H}$, we have $\mathcal{H} = \epsilon$, since `empty` is initialized to `true`, we have

$$(0 = \mathit{total}(\epsilon)) \wedge (0 >= 0) = \texttt{true}$$

Formally, we would prove the triple:

$$\{\mathcal{H} = \langle \mathit{parent}(\texttt{this}) \twoheadrightarrow \texttt{this.}\textbf{new}\ \texttt{StorageRoom}\rangle\}\ \texttt{counter} := 0\ \{I\}$$

this reduces to proving $\mathit{total}(\mathit{parent}(\texttt{this}) \twoheadrightarrow \texttt{this.}\textbf{new}\texttt{StorageRoom})$, which is the same as $\mathit{total}(\epsilon)$.

3c Analysis (weight 10)

Formulate the verification condition for proving that `remove` maintains the invariant I, and verify this condition. (You do not have to verify the `store` method.)

(Continued on page 13.)

Hint: You may use the following Hoare axiom when reasoning about an **await** statement with boolean guard `b` and **return** statement with expression `e`, postcondition Q, and auxiliary variable return:

$$\{I\}\ \textbf{await}\ \texttt{b}\ \{I \wedge \texttt{b}\}$$

$$\{Q_e^{\mathsf{return}}\}\ \textbf{return}\ \texttt{e}\ \{Q\}$$

Hint: In order to establish that some method `m`, defined by `m(x)`$\{$`Body`$\}$, maintains the invariant I, it suffices to verify the Hoare triple:

$$\{I\}\ \texttt{Body}\ \{I^{\mathcal{H}}_{\mathcal{H}\vdash\texttt{caller}\leftarrow\texttt{this.m(return)}}\}$$

Solution:

Verification condition:

$\{I\}$
await `counter >= x; counter := counter - x; return x`
$\{I^{\mathcal{H}}_{\mathcal{H}\vdash\texttt{caller}\leftarrow\texttt{this.remove(return)}}\}$

Also a *wlp* condition is ok:

$$I \Rightarrow wlp(\langle\langle counter()B\rangle\rangle, I)$$

where B is the three statements in the method body.

The postcondition $I^{\mathcal{H}}_{\mathcal{H}\vdash\texttt{caller}\leftarrow\texttt{this.remove(return)}}$ can be simplified by:

$$\begin{array}{ll} & ((\texttt{counter} = total(\mathcal{H})) \wedge (\texttt{counter} >= 0))^{\mathcal{H}}_{\mathcal{H}\vdash\texttt{caller}\leftarrow\texttt{this.remove(return)}} \\ = & (\texttt{counter} = total(\mathcal{H}) - \texttt{return}) \wedge (\texttt{counter} >= 0) \end{array}$$

By the **await** rule, we have

$$\{I\}\ \textbf{await}\ \texttt{counter} >= \texttt{x}\ \{I \wedge \texttt{counter} >= x\}$$

It then remains to prove:
$\{I \wedge \texttt{counter} >= x\}$
`counter := counter - x; return x`
$\{(\texttt{counter} = total(\mathcal{H}) - \texttt{return}) \wedge (\texttt{counter} >= 0)\}$
By backward construction, we arrive at the verification condition:

$$I \wedge \texttt{counter} >= x \Rightarrow \{(\texttt{counter} = total(\mathcal{H}) - \texttt{return}) \wedge (\texttt{counter} >= 0)\}^{\texttt{return,counter}}_{x,\texttt{counter}-x}$$

which is satisfied since

$$I \wedge \texttt{counter} >= x \Rightarrow (\texttt{counter} - x = total(\mathcal{H}) - x) \wedge (\texttt{counter} - x >= 0)$$

(Continued on page 14.)

Problem 4 Extra points (optional question): Program Verification (weight 7)

IMPORTANT NOTE:
This question is optional, you can solve this question if you wish to get extra points added on top of your final score.

The Factorial function:

In mathematics, the factorial of a non-negative integer **n**, denoted by **n!**, is the product of all positive integers less than or equal to **n**. For example,

5! == 5 * 4 * 3 * 2 * 1 == 120.

Let n be a positive integer, fac(n) calculates the factorial of n and is implemented as follows:

```
x = n;  y = 1;  z = 0;

while (z  !=  x) {
  z = z + 1;  y = y * z;
}
```

Use program logic (In the Appendix you can find a list of rules for program verification) to prove the following triple:

$$\{\texttt{True}\}\ \texttt{fac(n)}\ \{\texttt{y = x!}\}$$

Use **y = z!** as loop invariant.

Addtionally, you may assume the following properties for the factorial:

$$0! == 1$$

$$(n + 1)! == (n)! * (n+1)$$

Solution:

```
{True}
   x = n;  y = 1;  z = 0;
{x == n and  y == 1 and  z == 0}
{y == z!}
   while (z  !=  x) { {y == z! and z != x}
     {y * (z + 1) == (z + 1)!}
     z = z + 1;  {y * z == z!}     y = y * z;
     {y == z!}
```

(Continued on page 15.)

```
   }
{y == z! and z == x}
{y == x!}
```

Proof obligations:

```
x == n and  y == 1 and  z == 0  =>  y == z!    trivial because 0! = 1

y == z! and z != x => y * (z + 1) == (z + 1)!
True because y == z!  => y * (z + 1) == (z + 1)!

y == z! and z == x  => y == x!  trivial
```

Appendix: Inference Rules

Assignment

$$\{P_{x \leftarrow e}\}\texttt{x = e}\{P\}$$

Composition

$$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1\texttt{;}\ S_2\{R\}}$$

While

$$\frac{\{I \wedge B\}\texttt{S}\{I\}}{\{I\}\texttt{while (B) S;}\{I \wedge \neg B\}}$$

Consequence

$$\frac{(P' \Rightarrow P) \quad \{P\}\texttt{S}\{Q\} \quad (Q \Rightarrow Q')}{\{P'\}\texttt{S}\{Q'\}}$$

Non-deterministic choice ([])

$$\frac{\{P_1\}\ \texttt{S}_1\ \{Q\} \qquad \{P_2\}\ \texttt{S}_2\ \{Q\}}{\{P_1 \wedge P_2\}\ (\texttt{S}_1 [] \texttt{S}_2)\ \{Q\}}$$

Send

$$\{Q_{h \leftarrow h; A \uparrow B : m}\}\ \textbf{send}\ \mathtt{B} : \mathtt{m}\ \{Q\}$$

Receive

$$\{Q_{h \leftarrow h; B \downarrow A : m}\}\ \textbf{await}\ \mathtt{B} : \mathtt{m}\ \{Q\}$$

(Continued on page 17.)