UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in	**INF 5510**
Day of exam:	**June 10th, 2011**
Exam hours:	**14:30 – 18:30**

This examination paper consists of 5 pages.

Appendices:	**none**
Permitted materials:	**ANY written material including own notes.**

Teacher:	**Eric Jul**
Teacher at the exam:	**Arne Maus**
Exams collected by:	**Tor Ivar Johansen**

Make sure that your copy of this examination paper is complete before answering.

1 Emerald Conformity

Given the following Emerald program:

```
const BankAccount <- typeobject BankAccount
    operation deposit[Integer]
    operation withdraw[Integer] -> [Integer]
    function fetchBalance[] -> [Integer]
end BankAccount

const BAClass <- class BAClass
    var balance: Integer <-0
    export operation deposit[d: Integer]
       balance <- balance + d
    end deposit
    export operation withdraw[amount: Integer] -> [r:Integer]
       if balance < 0 then
          r <- 0
       elseif amount > balance then
          r <-  balance
       end if
       balance <- balance - r
       r <- amount
    end withdraw
    export function fetchBalance[] -> [r:Integer]
       r <- balance
    end fetchBalance
end BAClass

const Prog1 <- object Prog1
  process
    var ba: BankAccount
    var a: array.of[Any] <- array.of[Any].create[0]

    % Insert extra declaration here, if necessary

    a.addUpper[17]
    ba <- BAClass.create
    ba.deposit[250]
    a.addUpper[ba]
    a.addUpper["Emerald"]
    ba <- BAClass.create
    ba.deposit[300]
    a.addUpper[ba]

    % Insert your code here

  end process
end Prog1
```

1.1 Write Emerald Code for Iterating through Array

In the given program write some code that iterates through the array and for each element prints the index of the element the array and the balance for any element that conforms to `BankAccount`.

Answer Note:

```
%
const BankAccount <- typeobject BankAccount
    operation deposit[Integer]
    operation withdraw[Integer] -> [Integer]
    function fetchBalance[] -> [Integer]
end BankAccount

const BAClass <- class BAClass
    var balance: Integer <-0
    export operation deposit[d: Integer]
       balance <- balance + d
    end deposit
    export operation withdraw[amount: Integer] -> [r:Integer]
       if balance < 0 then
          r <- 0
       elseif amount > balance then
          r <-  balance
       end if
       balance <- balance - r
       r <- amount
    end withdraw

    export function fetchBalance[] -> [r:Integer]
       r <- balance
    end fetchBalance
end BAClass

const Prog1 <- object Prog1

  process
    var home: Node <- locate self
    var ba: BankAccount
    var i: Integer <- 0
    var count: Integer
    var a: array.of[Any] <- array.of[Any].create[0]
    var x: Any
    a.addUpper[17]
    ba <- BAClass.create
    ba.deposit[250]
    a.addUpper[ba]
    a.addUpper["Emerald"]
    ba <- BAClass.create
    ba.deposit[300]
    a.addUpper[ba]

% Insert your code here

    count <- a.upperbound - a.lowerbound + 1
    i <- a.lowerbound
    loop
    exit when (count <= 0)
       home$stdout.PutString["Index " || i.asstring]
       x <- a[i]
       if (typeof x) *> BankAccount then
           home$stdout.PutString[" balance: " ||(view x as
BankAccount).fetchBalance[].asstring]
       end if
       home$stdout.PutString[" " || "\n"]
       i <- i+1
```

```
      count <- count-1
   end loop
  end process
end Prog1
```

1.2 NIL

Will your code above work for NIL?

Answer note:

No.

If yes, explain what you had to do to make it work.

Answer note:

Added a check for NIL

```
       if (x != NIL) and ((typeof x) *> BankAccount) then
```

If no, point out the problem.

Answer note:

NIL does conform but it does not know about what to do.

2 Emerald Distributed Garbage Collection

2.1 Detection of the End of the Mark Phase

Give a short description of why a 2-phase commit algorithm is needed to complete the Mark Phase of the Emerald Distributed Garbage Collector. Illustrate the problem that the 2-phase commit algorithm solves by a simple example. Use either words, or a sequence of simple diagrams showing parts of the object graph including the color of the nodes. If you wish, you may use graphs similar to Figure 6.5 in Eric's Ph.D.

Answer note:

The 2-phase commit algorithm is necessary because a given node may not have any gray objects, but LATER it may receive shade request thus generating new gray objects. So the invariant once-empty-always-empty does not hold for the node local graysets. It does hold for the union of all gray sets. The 2-phase commit algorithm detects that there has been a time period where all gray sets have been empty at the same time, which then means that the mark phase is complete because no node can generate new shade requests.

See a hand drawn example on the second to last page.

3 Storage Layout

Given the following class and a variable declaration:

```
const Semaphore <- monitor class Semaphore [initial : Integer]
  class export operation create -> [r : Semaphore]
    r <- Semaphore.create[1]
  end create
  var count : Integer <- initial
  var waiters : Condition <- Condition.create
  export operation P
    count <- count - 1
    if count < 0 then
      wait waiters
    end if
  end P
  export operation V
    count <- count + 1
    if count <= 0 then
      signal waiters
    end if
  end V
end Semaphore

var s: Semaphore <- Semaphore.create[]

move s to locate self
```

3.1 Draw Storage Layout for an Emerald Object

Given the following piece of Emerald code, show the storage layout of the object reference by the variable s, in a diagram similar to Figure 4.1 in Eric's Ph.D. but also including actual numbers for virtual addresses and Object IDs. Include the Object Table, Object Descriptors, and Object Data Areas. Assign the objects an Object ID starting with 100. Assign objects virtual Memory addresses starting with 500. Note: You must assume that due to the move statement, the object referenced by s is a global object.

Answer note:

Hand drawing of diagram can be seen on the last page of this paper.

4 Immutability

Given the following piece of Emerald Code:

```
const ICoordinateClass <- immutable class ICoordinateClass ...
   ...
end ICoordinateClass

const CoordinateClass <- class CoordinateClass
    var x: Real <- 0.0
    var y: Real <- 0.0
    export operation setX[newX: Real]
       x <- newX
```

```
    end setX
    export operation setY[newY: Real]
       y <- newY
    end setY
    export operation getX[] -> [r: Real]
       r <- x
    end getX
    export operation getY[] -> [r: Real]
       r <- y
    end getY
    export operation getImmutable ...
        ...
    end getImmutable
end CoordinateClass


const Prog1 <- object Prog1
  process
     var c: CoordinateClass
     var ic: ICoordinateClass
     c <- CoordinateClass.create[]
     ic <- c.getImmutable[]
  end process
end Prog1
```

4.1 Write Emerald Code for Generating an Immutable Copy

Replace the "`...`" in the code piece by Emerald Code so that an immutable copy of the `CoordinateClass` is generated and returned from the operation `getImmutable`.

Answer note:

Here is the code:

```
const ICoordinateClass <- immutable class ICoordinateClass[x: Real, y: Real]
    export operation getX[] -> [r: Real]
       r <- x
    end getX
    export operation getY[] -> [r: Real]
       r <- y
    end getY
end ICoordinateClass

const CoordinateClass <- class CoordinateClass
    var x: Real <- 0.0
    var y: Real <- 0.0
    export operation setX[newX: Real]
       x <- newX
    end setX
    export operation setY[newY: Real]
       y <- newY
    end setY
    export operation getX[] -> [r: Real]
       r <- x
    end getX
    export operation getY[] -> [r: Real]
       r <- y
    end getY
```

```
    export operation getImmutable[] -> [r: ICoordinateClass]
       r <- ICoordinateClass.create[x,y]
    end getImmutable
end CoordinateClass


const Prog1 <- object Prog1
  process
     var c: CoordinateClass
     var ic: ICoordinateClass
     c <- CoordinateClass.create[]
     ic <- c.getImmutable[]
  end process
end Prog1
```

5 Emerald Concurrency: Rendezvous

5.1 Write Emerald Code for Rendezvous

Write a monitored Emerald Class that has a `Rendezvous` operation that allows two processes to meet up: When a process calls `Rendezvous` , it will wait for another process to call `Rendezvous`, thereafter both processes will proceed.

Answer note:

Below is the code for the requested class – and some code that illustrates it (not required).

```
const RendezvousC <- monitor class RendezvousC
  var count : Integer <- 0
  var waiters : Condition <- Condition.create
  export operation rendezvous
    if count == 0 then
      count <- 1
      wait waiters
    else
      count <- 0
      signal waiters
    end if
  end rendezvous
end RendezvousC

const Prog1 <- object Prog1
  process
    var i: Integer
    var worker: Any
    var home: Node <-locate self
    var r: RendezvousC <- RendezvousC.create
    for (i <- 1: i <= 2 : i <- i+1)
      worker <- object W
        process
          home$stdout.PutString[home.gettimeofday.asstring || ": "]
          home$stdout.PutString["Worker no " || i.asString || " starting\n"]
          (locate self).Delay[Time.create[10*i, 100]]
          home$stdout.PutString[home.gettimeofday.asstring || ": "]
          home$stdout.PutString["Worker no " || i.asString || " rendevousing\n"]
```

```
          r.rendezvous[]
          home$stdout.PutString[home.gettimeofday.asstring || ": "]
          home$stdout.PutString["Worker no " || i.asString || " back from
rendezvous\n"]
        end process
      end W
    end for
  end process
end Prog1
```

6 Emerald Mobility

6.1 Run-time Costs of Attachment

What is the run-time overhead in connection with assignment of an attached variable? Compare to an assignment to a non-attached variable.

Answer note:

There is no extra run-time overhead compared to an assignment to a non-attached variable; only a bit in a compiler generated table.

6.2 Layout of the Template for an Object

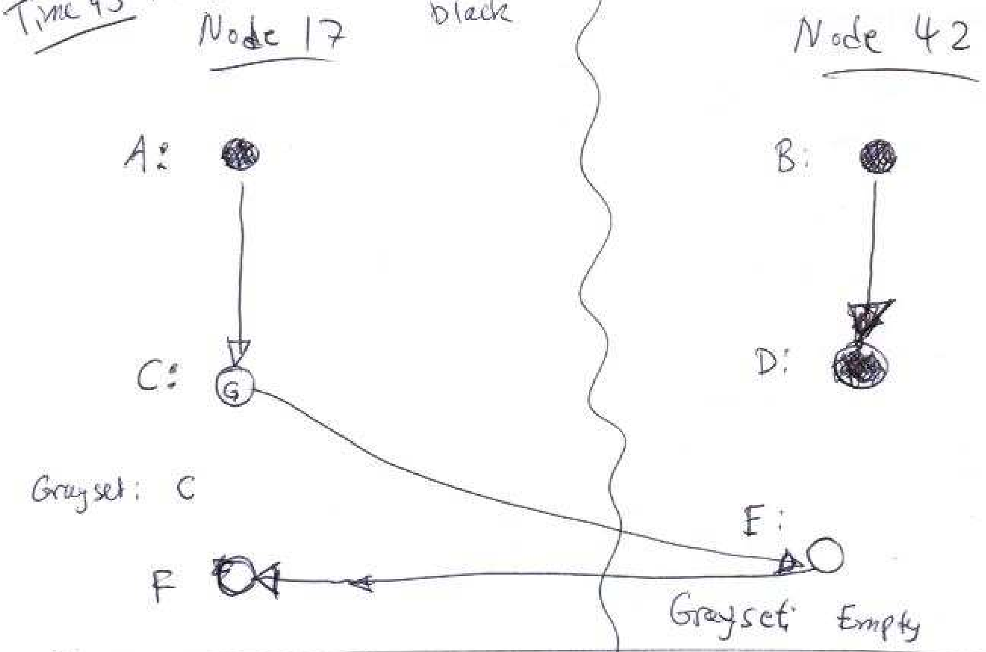
Describe the content of the template for the objects created by the `Semaphore` class shown in 3.1 above. Make your diagram similar to Figure 4.5 in Eric's Ph.D.

Answer note:

Hand drawing figure can been see on last page of this paper

2.1 Example Gray Sets concerning 2-phase commit

Time 43: After marking A, B, D black



Time 44 — Node 42 reports its grayset empty

Time 45 — Node 17 asks #42 to shade E

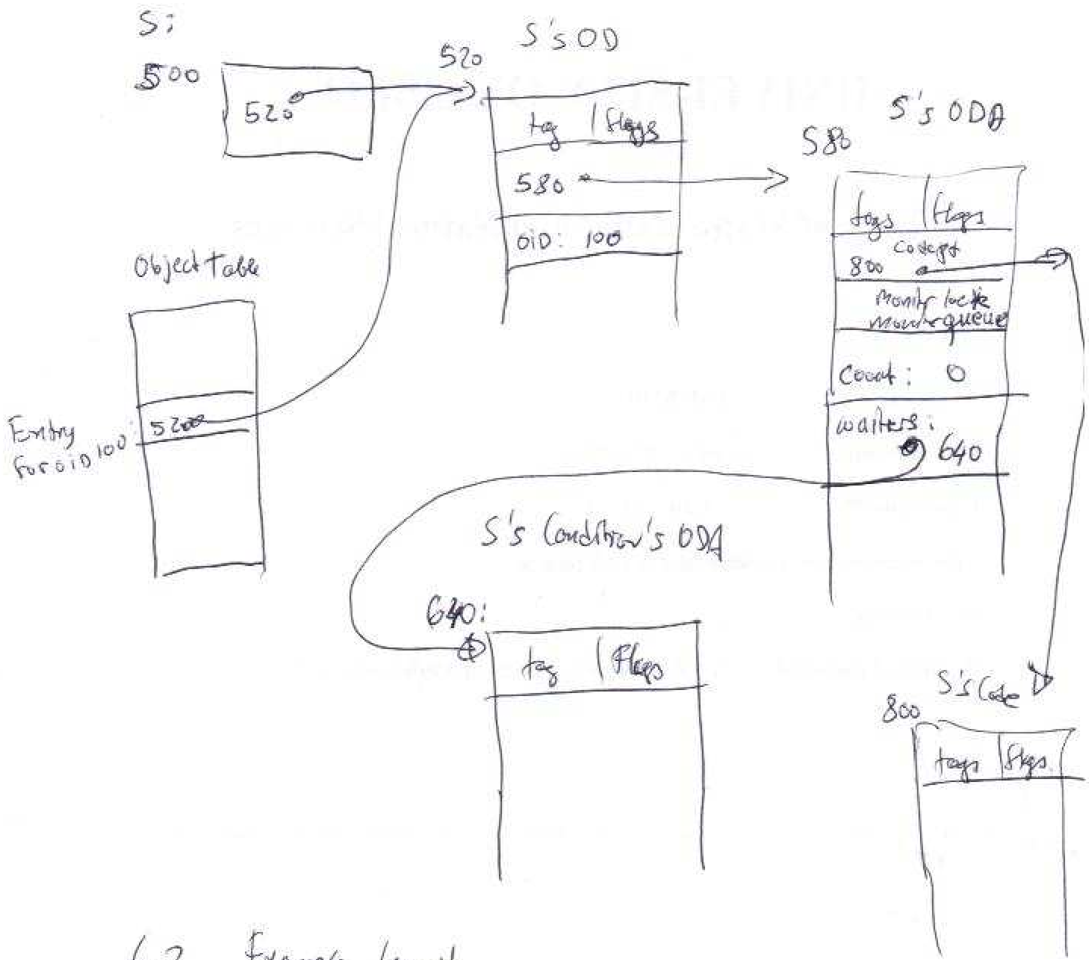
Time 46 — Node 42: shades E gray & reports back to node 17

Time 47 — Node 17 gets shade reply & marks C black

Time 48 — Node 17 reports its grayset empty

Clearly asking 42 at time 44 then asking 17 at time 48 will falsely report no grays. 2-phase necessary

3.1 Example layout



6.2 Example layout

Template for Semaphore Objects.

type	#	Attached
monitor	1	0
data	1	0
pointer	1	0

↑ this part is missing in Fig 4.5 so OK if you leave it out