IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 2015

BEng Honours Degree in Electronic and Information Engineering Part III
MEng Honours Degree in Electronic and Information Engineering Part III
BEng Honours Degree in Mathematics and Computer Science Part III
MEng Honours Degree in Mathematics and Computer Science Part III
MSc in Computing Science
for Internal Students of the Imperial College of Science, Technology and Medicine

This paper is also taken for the relevant examinations for the Associateship of the City and Guilds of London Institute

PAPER C528

CONCURRENT PROGRAMMING

Friday 27 March 2015, 10:00
Duration: 120 minutes

Answer THREE questions

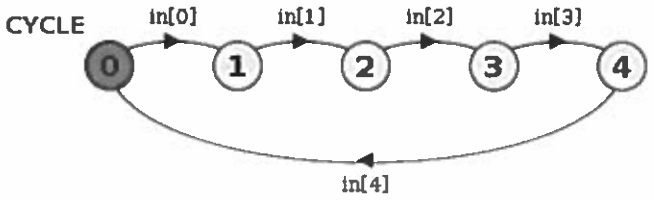
Paper contains 4 questions
Calculators not required

Section A *(Use a separate answer book for this Section)*

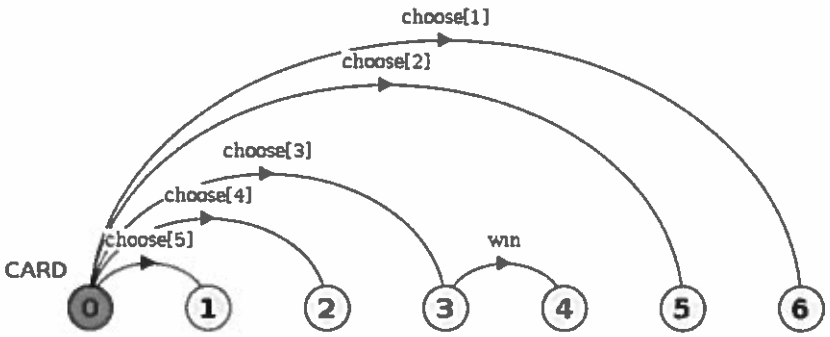
1 a Processes modelled in FSP are said to execute *asynchronously*. Briefly explain what asynchronous execution means in this context.

b For each of the following Labelled Transition Systems (LTS), give an equivalent FSP specification.

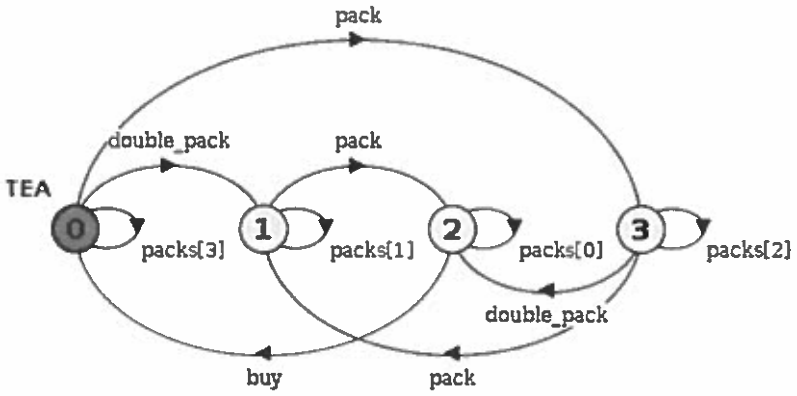
i)



ii)



iii)



c For each of the following specifications, give an equivalent *LTS*. Explain the difference between i) and ii).

i)
```
ON = (sit -> ON | seatBeltOff -> OFF),
OFF = ({sit,loosen} -> OFF | seatBeltOn -> ON).
```

ii)
```
property CAR = ON,
ON = (sit -> ON | seatBeltOff -> OFF),
OFF = ({sit,loosen} -> OFF | seatBeltOn -> ON).
```

The three parts carry, respectively, 20%, 60%, and 20% of the marks.

2a A thread that executes a `wait()` in Java releases the lock on the object whose method it is executing. Explain why releasing the lock by `wait()` is important when programming monitors in Java.

b For each of the following FSP specifications, give an equivalent LTS.

i)
```
MOTHER = (sleep -> eat -> clean -> MOTHER |
  eat -> wash -> watch_tv -> MOTHER |
  sleep -> clean -> {wash,watch_tv,eat} -> MOTHER).
```

ii)
```
const MAX = 4
POINTS = POINTS[0],
POINTS[i:0..MAX] =
   (when (i<MAX) buy[j:1..MAX-i] -> POINTS[j+i] |
    when (i>1) use[j:1..i] ->  POINTS[i-j] |
   point[i] -> POINTS[i]).
```

c i) Explain the meaning of Process Sharing (`{a1,a2,..,an}::P`) and Process Labelling (`a:P`) in the Finite State Process (FSP) notations.

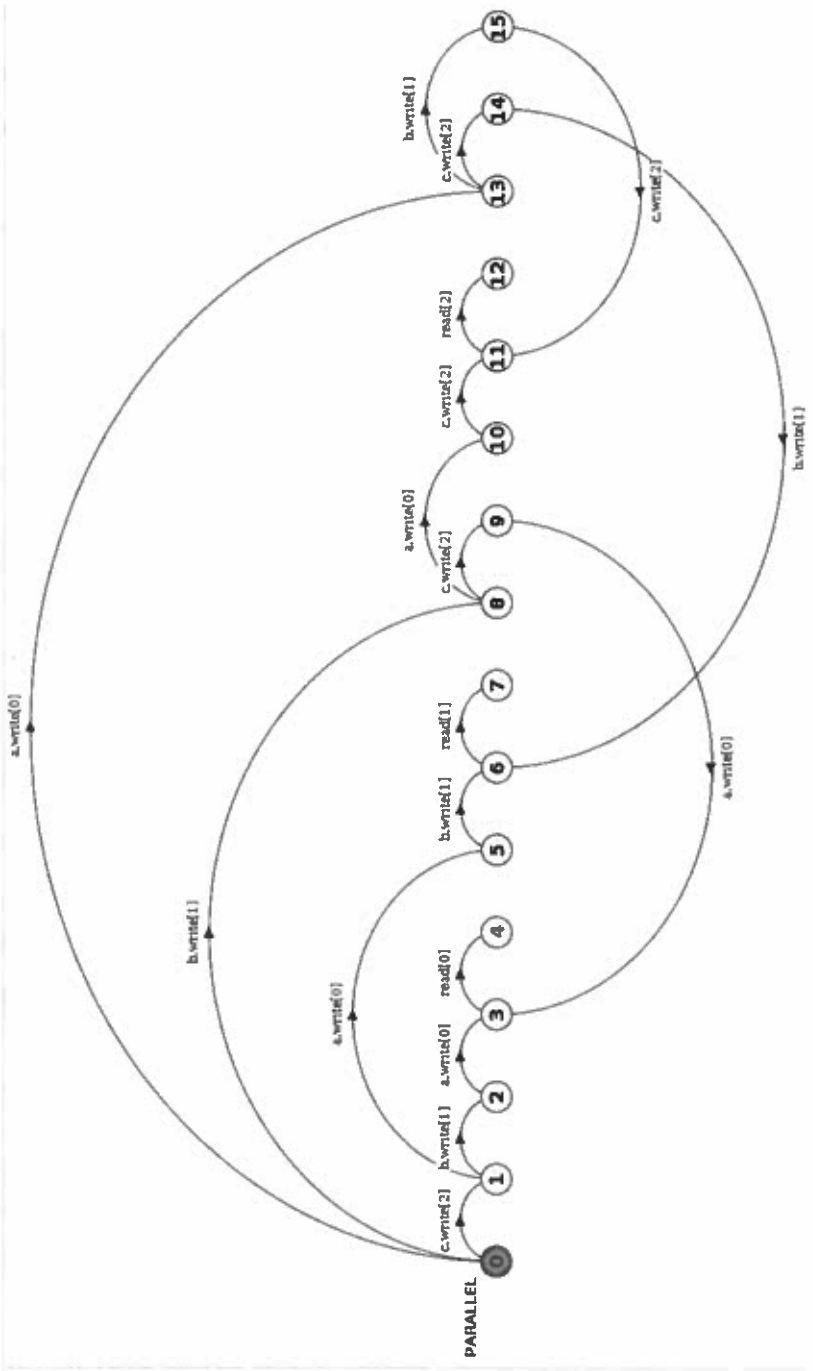
ii) Given the following FSP specifications, define the FSP specification for `PARALLEL` using Process Sharing and Process Labelling, i.e. fill `???` below.

```
const MAX = 2
VAR = VAR[0],
VAR[i:0..MAX] = (read[i] -> VAR[i] | write[j:0..MAX] -> VAR[j]).

THREAD(N=1) = (write[N]->read[i:0..MAX]->STOP)+ {write[i:0..MAX]}.

||PARALLEL = ???
```

The equivalent LTS of `PARALLEL` is given in the next page.



The three parts carry, respectively, 20%, 40%, and 40% of the marks.

Section B *(Use a separate answer book for this Section)*

3 The following is an FSP model of the Dining Philosophers problem that can reach a deadlocked execution.

```
PHIL = (sitdown -> right.get -> left.get ->
        eat -> left.put -> right.put ->
        arise -> PHIL).

FORK = (get -> put -> FORK).

/* For N=5, there are 5 philosophers: phil[0..4].
 * The index expression ((i-1)+N)%N is modulo N to
 * place a fork between, e.g., phil[0] and phil[4]. */
||DINERS(N=5) =
  forall [i:0..N-1](phil[i]:PHIL
    || {phil[i].left, phil[((i-1)+N)%N].right}::FORK).
```

a Give a trace to a deadlocked state for the DINERS process.

b In each of the following three cases, extend or modify the above Dining Philosophers model to avoid deadlocks by following the suggested approach. The answers to each case are independent.

i) Specify a WAITER process as a central arbitrator that permits only $N-1$ philosophers to sit down at the same time.

```
WAITER(N=5) = ...
  // N=5 specifies 5 philosophers (as in DINERS)
```

When composed with the original DINERS process, WAITER should permit a maximum of $N-1$ philosophers to engage in the sitdown action before an arise action occurs.

ii) Specify a modification of the original PHIL process to be parameterised as

```
PHIL(L=0, R=1) = ...
  // 0 and 1 placeholder default values
```

where L and R are integers that impart an ordering on the two forks. Have each philosopher get the fork with the lower index first.

Specify a modification of the DINERS composition to use the new PHIL process.

(Hint: it is not necessary to change FORK.)

iii) Specify a modification of the original `PHIL` process that gives the philosopher the choice to perform a `yield` action after acquiring the first fork. `yield` should be an alternative choice alongside acquiring the second fork. After yielding, the philosopher should release the first fork, arise and return to its initial state.

c i) For your answer in part (b) (iii), state whether your solution to avoiding deadlocks has introduced any progress problem. If so, state for which actions progress is violated.

ii) In a gathering of polite philosophers, the philosophers assign higher priority to yielding the first fork as follows:

```
||POLITEDINERS = DINERS << {phil[0..4].yield}.
```

(Here, `DINERS` is as the original `DINERS` but using your `PHIL` from (b) (iii).)

* State whether the `POLITEDINERS` system is deadlock-free.

* State whether progress violated for any actions? If so, state for which actions progress is violated

The three parts carry, respectively, 15%, 65%, and 20% of the marks.

4a Concurrent threads can interact by operating on shared state.

i) Explain the problem of unintended *interference* between such threads.

Write a small Java example to illustrate your explanation: give code for a passive object encapsulating mutable state, and code for active threads sharing and operating on this state. Outline how the interference problem can occur in an execution of your example.

ii) Explain the main mechanism supported in Java to avoid the interference problem and its usage.

Describe or show how your example in (i) should be modified to avoid interference.

b Write a Java class that implements a monitor satisfying the following specification.

The purpose of the monitor is to allow a group of N threads to synchronize all together repeatedly. Additionally, the monitor determines, in each round of synchronizations, which thread was the first to initiate its synchronization action.

- N is specified as a constructor argument when the monitor object is created. Assume $N > 1$.
- The monitor has a `public` method `checkFirst` that returns a `boolean`.
- The `checkFirst` method blocks the calling thread until all N threads have called `checkFirst`. That is, `checkFirst` ensures all N threads have called `checkFirst` before any of them can proceed.
- `checkFirst` returns (after all N threads have called this method) to each of the N threads: `true` for the thread whose call to `checkFirst` was the first to execute in this round, and `false` otherwise.
- An instance of the monitor should be reusable by the threads to repeat this synchronization activity multiple times.

Usage example. The following describes the intended behaviour of the monitor for an example scenario. You do not have to implement this use case.

Assume $N = 3$, with threads `t1`, `t2` and `t3`, calling `checkFirst` on a single instance of the monitor. If the `checkFirst` call by `t2` was the first to execute, then `checkFirst` will return `true` to `t2` and `false` to `t1` and `t3`.

The two parts carry equal marks.