UNIVERSITY OF OXFORD
SOFTWARE ENGINEERING PROGRAMME

Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Tel +44(0)1865 283525 Fax +44(0)1865 283531
info@softeng.ox.ac.uk www.softeng.ox.ac.uk

Part-time postgraduate study in software engineering

Concurrency and Distributed Systems, CDS

27th November – 1st December 2023
ASSIGNMENT

The purpose of this assignment is to test the extent to which you have achieved the learning objectives of the course. As such, your answer must be substantially your own original work. Where material has been quoted, reproduced, or co-authored, you should take care to identify the extent of that material, and the source or co-author.

Your answers to the questions on this assignment should be submitted using the Software Engineering Programme website — `www.softeng.ox.ac.uk` — following the submission guidelines. When submitting the assignment online, it is important that you formally complete all three assignment submission steps: step 1, read through the declaration; step 2, upload your files; step 3, check your files. **Please ensure your submission is anonymous: do not include your name or any other identifying information on the assignment, nor in accompanying material such as source code, nor within the file names of anything submitted.**

The deadline for submission is 12 noon on Tuesday, 16th January 2024. You are strongly encouraged to submit a version well before the deadline. You may update your submission as often as you like before the deadline, but no submissions or changes will be accepted after the deadline.

We hope to have preliminary results and comments available during the week commencing Monday, 26th February 2024. The final results and comments will be available after the subsequent examiners' meeting.

ANY QUERIES OR REQUESTS FOR CLARIFICATION REGARDING THIS ASSIGNMENT OR PROBLEMS INSTALLING SOFTWARE SHOULD, IN THE FIRST INSTANCE, BE DIRECTED TO THE PROGRAMME OFFICE WITHIN THE NEXT TWO WEEKS.

Concurrency and Distributed Systems
November 2023

Assignment

Please attempt all questions.

Your answers should be submitted as a PDF document; this is likely to be between 10 and 20 pages in length, depending upon the amount of `csp` code included.

You are encouraged to submit also the final versions of the `.csp` scripts that you create. These will not be marked, but may provide additional context for comments and feedback.

Question 1

Consider a system in which there are six events:

```
channel a, b, c, d, e, f
```

and three component processes, defined by

```
P =
  let
    P1 = a -> P2

    P2 = c -> P2
         []
         e -> P1
  within
    P1
```

```
Q =
  let
    Q1 = b -> Q2
         |~|
         c -> Q2

    Q2 = d -> Q1
  within
    Q1
```

```
R =
  let
    R1 = d -> R2
         []
         e -> R2

    R2 = f -> R1
  within
    R1
```

The behaviour of the system is described by the following parallel combination:

```
System =
  ( P
    [| {c} |]
    Q )
  [| {d,e,f} |]
  R
```

(a) Show how the behaviours—the traces, failures, and divergences—of the parallel combination of P and Q, sharing c, can be described by a single sequential process PQ. Your definition of PQ should use a set of equations in which the only operators to the right hand side of each equation are STOP, ->, [], and |~|

```
PQ =
  let
    P1Q1 = ...

      .
      .
      .

  within
    P1Q1
```

Write a pair of refinement checks to show that your definition of PQ produces exactly the same set of behaviours as the process `P [|{c}|] Q`.

(b) Building upon your answer to Part (a), show also how the behaviours of the parallel combination `System` can be described using a single, sequential process.

```
PQR =
  let
    P1Q1R1 = ...

      .
      .
      .

  within
    P1Q1R1
```

Again, write a pair of refinement checks to show that your definition of `PQR` produces exactly the same set of behaviours as the process `System`.

(c) The relationship between the occurrences and the availability of the events `a`, `b`, and `f` is complicated. We can explore this relationship by considering the behaviours of the process

```
System' = System \ {c,d,e}
```

(i) Show how we can use trace refinement to confirm that, in behaviours of `System'`, the event `f` cannot occur unless at least one of `a` and `b` has occurred already: that is, of these three events, the first to occur must be either `a` or `b`.

(ii) Show how we can use failures–divergences refinement to confirm that, whenever `a` or `b` occurs, `f` is guaranteed to be available: that is, immediately after any `a` or `b`, the process `System'` is ready to perform `f` as the next event.

(iii) Show how the behaviours of `System'` can be described by a single sequential process, using a set of equations in which the only operators to the right hand side of each equation are `STOP`, `->`, `[]`, and `|~|`.

Question 2

Three friends are planning to meet up to play a board game. The game requires a minimum of three players: it doesn't work with only one or two. There are two places that they could meet, `a` and `b`.

```
datatype Place = a | b
```

These two places are far apart, and the friends will need to reach agreement before setting out. Each friend will decide independently which place they would prefer, and then propose that place to each of the others. Each will abide by the view of the majority: for example, if two or more of them propose `a`, then all three will go to `a`.

We will use numbers between 1 and 4 to refer to the friends: there's another friend coming up later in the question.

```
Friend = {1..4}
```

We will use the event `prefer.i.p` to represent the action of friend `i` deciding that they prefer `p`

```
channel prefer : Friend . Place
```

and the event `propose.i.j.p` to represent the action of friend `i` proposing to friend `j` that they should meet at place `p`:

```
channel propose : Friend . Friend . Place
```

We will use the event `goto.i.p` to represent the action of friend `i` going to place `p`.

```
channel goto : Friend . Place
```

and we will start with just three friends:

```
Three = {1..3}
```

The following definitions represent an attempt at describing the behaviour of the friends:

```
Good(i) =
  let
    Others = diff(Three,{i})

    Start =
      prefer.i.a -> Propose(a,Others,1,0)
      |~|
      prefer.i.b -> Propose(b,Others,0,1)

    Propose(p,S,A,B) =
      ( [] j : S @ propose.i.j.p ->
          Propose(p,diff(S,{j}),A,B) )
      []
      ( A + B <= 3 & [] j : Others @ propose.j.i.a ->
          Propose(p,S,A+1,B) )
      []
      ( A + B <= 3 & [] j : Others @ propose.j.i.b ->
          Propose(p,S,A,B+1) )
      []
      A >= 2 & goto.i.a -> STOP
      []
      B >= 2 & goto.i.b -> STOP
  within
    Start

proposals(i) =
  union ( {| propose.i |}, {| propose.j.i | j <- Friend |} )

Alpha(i) =
  union ( proposals(i), {| prefer.i, goto.i |} )

Evening =
  || i : Three @ [ Alpha(i) ] Good(i)
```

(a) Explain the role of each of the four parameters in the definition of the subprocess `Propose(p,S,A,B)`.

Explain why the constraint `A + B <= 3` has been included in two places. Does it have to be exactly this constraint, or would a similar constraint have the same effect?

Does it matter that the event `propose.i.i` is included in the alphabet of `Good(i)`? Explain your answer.

Using only `goto` events, write a process that describes the two acceptable, successful outcomes to the evening

```
Success = ...
```

Your process should be as nondeterministic as possible.

Using `Success` as a specification, write a failures–divergences refinement check that can be used to confirm that the evening is guaranteed to be successful: that is, that everyone goes to `a`, or everyone goes to `b`. Explain why this check fails.

(b) Write a new version of the good friend process, `Good2(i)`, that will guarantee that the evening is successful. Your new process should be similar to `Good(i)`, differing only in terms of the constraints or guards preceding two of the options. Explain your answer.

```
Good2(i) = ...
```

Write a new version of the evening process, `Evening2`, to make use of `Good2(i)`

```
Evening2 = ...
```

and write a refinement check that can be used to confirm that show that your definitions of `Good2(i)` and `Evening` guarantee a successful outcome.

(c) Suppose now that there are four friends:

```
Four = {1..4}
```

Write a process `Good3(i)` that is identical to `Good2(i)`, except that there are four rather than three friends to consider. You will need to change the definition of `Others` and make some minor modifications to the constraints in `Propose(p,S,A,B)`.

```
Good3(i) = ...
```

Write a process `Evening3` that uses `Good3(i)` and a new version of the specification that takes account of the fact that there are four friends

```
Evening3 = ...

Success3 = ...
```

and write a refinement check that can be used to confirm that either all four friends go to `a`, or all four go to `b`. Explain why this check fails.

(d) In Part (c), if the decision process has been unsuccessful, then each of the four friends will be in the same state. We can fix the problem by extending the definition of `Process(p,S,A,B)` to allow for a new course of action: if they find themselves in this state, Friend `1` will declare their own preference, and all four friends will go there. We will use `declare.i.j.p` to represent the action of `i` declaring to `j` that their preference is for `p`.

Introduce the new channel `declare`, write a new version of the friend process,

```
Good4(i) = ...
```

that includes the new course of action, and new versions of the alphabet and parallel combination, `Alpha4(i)` and `Evening4`, to take account of declarations. Explain your answer, and write a refinement check to confirm that success is guaranteed.

(e) Now suppose that one of the friends may fail to make all of their proposals. In all other respects, they will behave as a good friend: they will be ready to receive proposals from others, and to receive declarations from Friend `1`; they will also go with the majority preference, or the preference of Friend `1` if necessary.

Write a process to describe the behaviour of an okay friend

```
Okay5(i) = ...
```

and a new version of the parallel combination, `Evening5`, to include three good friends `1` to `3` and one okay friend `4`. Write a refinement check that could be used to determine whether success is guaranteed, and explain why this check fails.

(f) With a fourth friend who is just okay, a different approach is required. Rather than base their actions upon purely upon the replies received, the friends will also keep an eye on the time. At six o'clock, each friend will consider the replies that they have received, and see whether there is a place with three or more votes.

If there is, then they will wait until seven o'clock, and then go to that place. If there is not, then what they do depends upon which friend they are: if they are Friend `1`, they will declare their preference to each of the others, before seven o'clock; if they are one of the others, they will ask Friend `1` for their preference, again before seven o'clock, and then wait for the answer before going to the place that Friend `1` prefers.

While Friend `1` is waiting, or declaring, they will be ready to be asked for their preference by any of the others. While the others are waiting, they will need to be ready to hear Friend `1` declare a preference. Introducing new channels as necessary, write new definitions for the good and okay friend processes,

```
Good6(i) = ...

Okay6(i) = ...
```

that guarantee a successful outcome, and write a check that can be used to confirm that this is the case.

You may have come up with a solution that will result in the four friends going to the place that is preferred by Friend `1`, regardless of the preferences of the others. Write one or more checks that can be used to determine whether or not this is the case. If it is, try to come up with another solution in which other outcomes are possible.

Explain your answer.

Question 3

A customer who enters the Visa Centre will be given a numbered ticket by a security guard. The numbers on the tickets range from 0 to maxnum - 1:

```
Number = {0..maxnum - 1}
```

Once the numbers have reached maxnum - 1, they start again from zero.

There are two desks, each labelled with a letter:

```
datatype Letter = A | B
```

Each of the desks is equipped with a button. A clerk can press this button to summon the next customer. The next number will then be displayed on a screen, along with the letter of the desk that the clerk is sitting at. After pressing the button, the clerk will look at the screen to see which number they should expect.

When a customer sees their number on the screen, they should go to the desk indicated and present their ticket. If the number is the one that the clerk is expecting, they will serve the customer. If not, then they will return the ticket to the customer, who will need to continue waiting. In either case, once they have served the customer, or returned the ticket, the clerk will be ready to press the button again, to summon the next customer.

We will suppose that there are three customers:

```
datatype Customer = P | Q | R
```

A customer cannot leave without being served; they must continue waiting. A customer who has been served must leave immediately.
We will model the behaviours of the visa centre and its customers using the following channels:

```
channel enter, leave : Name
channel ticket : Number
channel next : Letter
channel see : Number . Letter
channel present, return : Letter . Name . Number
channel serve : Letter . Name
```

Note that the channel see represents two kinds of actions that can be performed independently: a clerk at a desk seeing something on the display, and a customer seeing something on a display. If we are to use just one channel, and not record (in the event) who is doing the seeing, then we will need to interleave, rather than share, this channel when we put processes together.
We can easily write a process to describe the contribution of the security guard, who gives out tickets in numerical order:

```
Ticket =
  let
    State(n) =
      ticket!n -> State((n+1)%maxnum)
  within
    State(0)
```

The modulo operator % is useful here: we write m%n to mean the value of number m modulo n; for example, 7%4 = 3.

(a) Choose a value for maxnum:

```
maxnum = ...
```

Explain your choice.

(b) Write a process to describe the contribution of the display screen. This is blank until the very first time that a `next` event occurs; after that point, it is always ready for anyone to see the latest combination of a number and a letter, corresponding to a ticket and a desk. This combination is updated whenever a `next` event occurs.

```
Display = ...
```

Briefly explain your answer.

(c) Write a process to describe the contribution of a clerk at either of the desks:

```
Desk(l) = ...
```

A clerk should begin by pressing the `next` button at their desk. They should then see what number is being displayed, and be ready for a customer to present a ticket, as explained above.

(d) Write a process to describe the contribution of a customer who won't leave without being served:

```
Customer(c) = ...
```

Briefly explain your answer.

(e) Write a process to describe the combination of all of these components acting together, using shared parallel `[| |]` and interleaving `|||`.

```
System = ...
```

Briefly explain your answer.

(f) If a customer enters the centre, can we guarantee that they will be able to leave? Write one or more refinement checks that can be used to answer this question. Depending upon how you approach this, you may need to exclude certain behaviours from consideration by redesigning your customer process. Explain your answer.

Your investigation may have revealed a shortcoming in the design of the centre. If it has, explain what it is, and suggest at least one way in which it might be addressed.