Exercises

Concurrency and Distributed Systems
January 2023

Contents

Matchsticks

```
FirstPlayerWins =
  let
    Wins(p) =
      (|~| q : PlayerID @ (|~| n : {0..15} @ look.q.n -> Wins(p)))
      |~|
      (|~| q : PlayerID @ (|~| n : {0..15} @ take.q.n -> Wins(p)))
      |~|
      (|~| q : PlayerID @ (turn.q -> Wins(q)))
      |~|
      win.p -> STOP
  within
    |~| p : {0,1} @ turn.p -> Wins(p)
```

In the previous exercises, we checked that

```
FirstPlayerWins [FD= SmartGame
```

using the above specification process?

Now that we have hiding, can we come up with a simpler specification?

Medical accelerator

A particular radiation therapy machine had two modes: high energy photons, and low energy electrons.

In the first mode, a ‘shield’ was required between the beam source and the patient. This shield had to be removed when the machine was used in the second mode, for otherwise the expected radiation would not reach the patient.

Events

Create a script `exercisesD.csp` and add the following declaration:

```
channel setHigh, setLow, fire, cancel,
  shieldHigh, shieldLow, beamHigh, beamLow
```

We will use: `setHigh` to represent the action of the operator selecting high energy treatment; `shieldHigh` to represent the action of the shield moving into place for that kind of treatment; and `beamHigh` to represent the action of the accelerator moving into high energy beam mode.

We will use: `setLow` to represent the action of the operator selecting low energy treatment; `shieldLow` to represent the action of the shield moving into place for that kind of treatment; and `beamLow` to represent the action of the accelerator moving into low energy beam mode.

We will use `fire` to represent the action of the operator commencing treatment—‘firing’ the beam.

We will use `cancel` to represent the action of the operator cancelling a selected treatment—deciding not to fire the beam.

A key consideration is that the event `fire` should not be possible unless the beam and the shield are both in the correct position for high energy treatment, or both in the correct position for low energy treatment.

In particular, firing a high energy beam without the shield in place may cause serious injury or death.

Design

The behaviour of the machine, with respect to these actions, is described as follows. Add this definition to your script.

```
System =
  SetLow ; Ready

Ready =
  setHigh -> High [] setLow -> Low

High =
  (HighTreatment /\ cancel -> SKIP) ; Ready

HighTreatment =
  SetHigh ; fire -> SKIP ; SetLow
SetHigh =
  beamHigh -> SKIP ||| shieldHigh -> SKIP

SetLow =
  beamLow -> SKIP ||| shieldLow -> SKIP

Low  =
  (LowTreatment /\ cancel -> SKIP) ; Ready

LowTreatment =
  fire -> SKIP
```

Safety

We may formulate a safety check for this design as follows:

```
datatype position = high | low

Safe =
  let
    State(b,s) =
      (b == s) & fire -> State(b,s)
      []
      beamHigh -> State(high,s)
      []
      beamLow -> State(low,s)
      []
      shieldHigh -> State(b,high)
      []
      shieldLow -> State(b,low)
  within
    State(low,low)
```

```
assert Safe [T= System \ { setHigh, setLow, cancel }
```

What does the refinement check tell us? How can we fix this design?

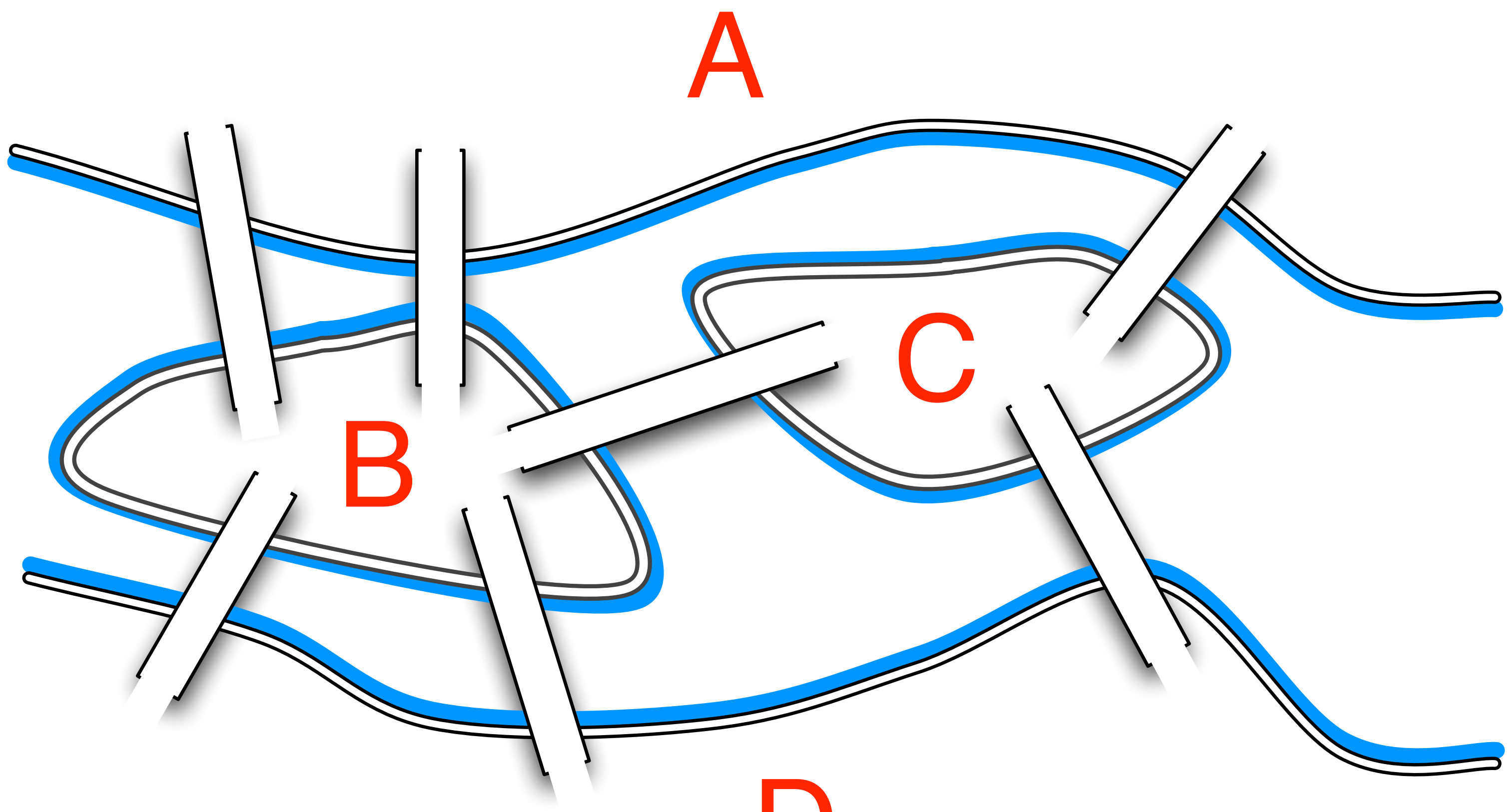
Bridges

The city of Königsberg once had seven bridges, connecting the two sides of the Pregel River by way of a pair of large islands.

One of the bridges ran between the two islands; the others connected the islands to the banks of the river.

In 1735, the mathematician Euler showed that it was impossible to find a route, a walk around the city, that crossed each bridge exactly once.

In the diagram on the next slide, the two sides of the river are labelled A and D, and the two islands are labelled B and C.


A
B
C
D

Events

Add the following declarations to your script:

```
datatype LOC = A | B | C | D

channel start, depart, arrive : LOC
```

We will use: `start.l` to represent the action of starting our walk at location `l`; `arrive.l` to represent the action of arriving at `l`, via one of the bridges; `depart.l` to represent the action of leaving `l`, again via one of the bridges.

Locations

Add the following declarations to your script:

```
Location(l) =
  let
    Here =
      depart.l -> NotHere

    NotHere =
      start.l -> Here
      []
      arrive.l -> Here
  within
    NotHere
```

```
Locations = ||| l : LOC @ Location(l)
```

Could we have defined `Locations` using alphabet parallel, without also using renaming?

Bridges

Add the following:

```
channel done

Bridge(l,m) =
  depart.l -> arrive.m -> done -> STOP
  []
  depart.m -> arrive.l -> done -> STOP
```

This process represents a bridge between locations `l` and `m` that can be used in either direction, but can only be used once overall.

After it has been used, it is ready to engage in the event `done`.

Add and complete the following declarations:

```
bridge(1) = Bridge(A,B)
bridge(2) = Bridge(A,B)
bridge(3) = Bridge(A,C)
...

Bridges = [| {done} |] b : {1..7} @ bridge(b)
```

Could we have defined `Bridges` using alphabet parallel, without also using renaming?

Routes

Add and complete the following declarations:

```
City =
  Locations [| {| arrive,depart |} |] Bridges

Start =
  ..

Routes =
  City [| {| start |} |] Start
```

The process `Start` should insist that `start` happens only once.

Refinement

Add and explain the following assertions:

```
assert
  STOP [T= Routes \ {| start, arrive, depart |}

assert
  Routes \ {| start, arrive, depart |} [T= done -> STOP
```

What do they tell us?

Adding a bridge

Show that the addition of one more bridge—let's call it `bridge(8)`—makes it possible to find a walk around the city in which each bridge is crossed exactly once.

(You will need to modify the definitions of `bridge` and `Bridges`, but not those of `Bridge`, `City`, or any other process.)

Completing a tour: optional

Add the following declaration to your script:

```
channel finish : LOC
```

Modify your definition of `Location(l)` so that it allows the event `finish.l` when the walker is at that location.

Modify your definition of `Start` so that it allows also `done` and `finish` events, insisting that they can occur only in that order.

Make the necessary modifications to the alphabets and synchronisation sets.

Use refinement checks to see whether or not it is possible to complete a walk around the city, crossing every one of the (now eight) bridges, and ending up back where you started.