Exercises

Concurrency and Distributed Systems

January 2023

Contents

Matchsticks

Matchsticks is a game for two players.

A pile of 15 matchsticks is placed upon a table. The players then take turns, in each turn removing up to three matchsticks.

A player cannot take more matches than there are left available. A player must take at least one match at each turn.

The player who takes the last match loses the game.

Events

We will use the following event abstraction:

- `look.i.n`: player `i` looks to see that there are `n` matches remaining in the pile;
- `take.i.n`: player `i` removes `n` matches from the pile;
- `turn.i`: the players agree that it is Player `i`'s turn to take some matches;
- `win.i`: player `i` is declared the winner of the game.

Channels

Create a script `exercisesC.csp` with the following declarations:

```
PlayerID = {0,1}

channel look : PlayerID . {0..15}
channel take : PlayerID . {0..15}
channel turn, win : PlayerID

min(i,j) = if i < j then i else j
```

Matches

Add the following process definition:

```
alphaMatches = {| look, take |}

Matches =
  let
    Remaining(n) =
      look?p!n -> Remaining(n)
      []
      (n > 0) & [] m : {1..n} @ take?p!m -> Remaining(n-m)
  within
    Remaining(15)
```

Explore the behaviour of this process using `:probe Matches`.

Player

Complete the following process definition:

```
alphaPlayer(p) = {| turn, look.p, take.p, win.p |}

Player(p) =
  let
    MyTurn =
      turn.p -> Look(p)

    Look(p) =
      look!p?n -> Looked(n)

    Looked(n) =
      (n > 0) & Act(n)
      []
```

```
    (n == 0) & win.p -> STOP

  Act(n) =
    |~| k : {1..min(3,n)} @ take!p!k -> YourTurn

  YourTurn =
    turn.(p+1)%2 -> MyTurn
within
  ...
```

(% is the modulus operator, so `(p+1)%2` is 0 if `p` is 1 and 1 if `p` is 0.)

Explore the behaviour of an instance of this process using
`:probe Player(0).`

Game

```
alphaPlayers = {| turn, look, take, win |}

Players =
  || p : PlayerID @ [alphaPlayer(p)] Player(p)

Game =
  Matches [ alphaMatches || alphaPlayers ] Players
```

Explore the behaviour of this process. Check to see that it is deadlock free (you should see a trace ending in a `win`).

Smart player

Define a process `SmartPlayer` to describe a player that behaves as `Player` except that, at each turn, they will attempt to leave their opponent with `4k + 1` matches, for some positive integer `k`.

(And if they can't do that, they will take just one match.)

Specification

Add the following specification to your script:

```
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)
```

Refinement checks

If we were to define

```
SmartPlayers =
  || p : PlayerID @ [alphaPlayer(p)] SmartPlayer(p)

SmartGame =
  Matches [ alphaMatches || alphaPlayers ] SmartPlayers
```

What would the following refinement checks tell us?

- `assert FirstPlayerWins [T= SmartGame`
- `assert FirstPlayerWins [FD= SmartGame`

Revolving Door

A revolving door has four sections. At any one time, one of these sections is accessible from the outside of the building, another is accessible from the inside, and the other two are turning in and turning out.

A person may `enter` or `leave` either of the sections that are inside or outside.

A person in one of the other two sections must wait until the door has rotated a further 90 degrees, upon which their section will be facing inside or outside, and they can step out.

Events

Add the following declarations to your script:

```
PersonID = {0, 1}

datatype Sides = inside | outside

channel enter, leave, detect : Sides . PersonID
channel rotate
```

We will use the event `rotate` to represent the action of the door rotating 90 degrees.

There is a motion detector on either side of the door. We will use the event `detect.s.p` to represent the action of a person `p` being detected on side `s`.

Door

Add the following definition to your script:

```
aDoor = {| enter, leave, rotate |}

Door =
  let
    State(Outside,RotatingIn,Inside,RotatingOut) =
      empty(Outside) &
        enter.outside?p ->
          State({p},RotatingIn,Inside,RotatingOut)
      []
      ( [] p : Outside @
        leave.outside.p ->
          State({},RotatingIn,Inside,RotatingOut) )
      []
```

```
      empty(Inside) &
        enter.inside?p ->
          State(Outside,RotatingIn,{p},RotatingOut)
      []
      ( [] p : Inside @
        leave.inside.p ->
          State(Outside,RotatingIn,{},RotatingOut) )
      []
      rotate ->
        State(RotatingOut,Outside,RotatingIn,Inside)
  within
    State({},{},{},{})
```

Door Controller

Add and complete the following definition:

```
aDoorController = {| detect, rotate |}

DoorController =
  let
    ...
  within
    Rotates(0)
```

The door controller process should allow exactly four `rotate` events after the most recent `detect` event.

It should always be ready to accept another `detect` event, from either side of the door.

Assumption

Add the following definition to your script:

```
aConstraint(p) = { enter.s.p, leave.s.p, detect.s.p | s <- Sid

Constraint(p) =
  let
    InTheDoor =
      leave?s!p -> NotInTheDoor

    NotInTheDoor =
      detect?s!p ->
        enter!s!p -> InTheDoor
  within
    NotInTheDoor
```

```
aAssumption = {| enter, leave, detect |}

Assumption =
  || p : PersonID @ [aConstraint(p)] Constraint(p)
```

What is this assumption that we are making? Is it reasonable?

Deadlock

Add the following, and check the assertion:

```
aDoorSystem = {| enter, leave, rotate, detect |}

DoorSystem =
  Door [ aDoor || aDoorController ] DoorController

DoorSystemWithAssumption =
  DoorSystem [ aDoorSystem || aAssumption ] Assumption

assert DoorSystemWithAssumption :[deadlock free]
```

What is going on? Would it make a difference if `PersonID` was `{1,2,3}`? How would you fix this problem in real life?