Exercises

Concurrency and Distributed Systems

January 2023

Contents

Lifts

Consider a building with two floors

```
FLOOR = {0,1}
```

used by two different people

```
datatype PERSON = aa | bb
```

A person may enter or leave the building. They may also enter or leave a lift, or call the lift, at a particular floor.

```
channel enterBuilding, exitBuilding : PERSON
channel enterLift, exitLift, call : FLOOR . PERSON
```

Floors

```
Floor(f) =
  let
    Holding(P) =
      call!f?p:P -> Holding(P)
      []
      enterLift!f?p:P -> Holding(diff(P,{p}))
      []
      exitLift!f?p -> Holding(union(P,{p}))
      []
      f == 0 & ( enterBuilding?p -> Holding(union(P,{p}))
                 []
                 exitBuilding?p:P -> Holding(diff(P,{p})) )
  within
    Holding({})
```

```
Floors =
||| f : FLOOR @ Floor(f)

aFloors = {| call, enterLift, exitLift,
             enterBuilding, exitBuilding |}
```

Once in the lift, a person may request a specific floor, the lift may go to a specific floor, and the lift doors may open or close.

```
channel request : FLOOR . PERSON
channel goto : FLOOR
channel close, open
```

The lift could check whether a person is authorised to use it—the channel `check` is also in its alphabet—but this feature hasn't been enabled as yet.

```
aLift = {| request, enterLift, exitLift,
           open, close, goto, check |}

Lift =
  let
    Open(f,P) =
      request?g?p:P -> Open(f,P)
      []
      enterLift!f?p -> Open(f,union(P,{p}))
      []
      exitLift!f?p:P -> Open(f,diff(P,{p}))
      []
      close -> Closed(f,P)
```

```
  Closed(f,P) =
    request?g?p:P -> Closed(f,P)
    []
    goto?g -> Closed(g,P)
    []
    open -> Open(f,P)
within
  Open(0,{})
```

Controller

The control system accepts calls and requests, and controls the movement of the lift.

The first floor is a secure area, and only authorised people are allowed to visit. Authorisation may be granted; it may also be revoked.

The system maintains a list of which people are authorised to visit the first floor, and will allow `call`s and `request`s only from people who are on this list. It will also confirm whether a given person is on the list, or not.

```
channel authorise, deauthorise : PERSON
datatype AUTHORISED = yes | no
channel check : PERSON . AUTHORISED

aController = {| goto, request, call,
                 authorise, deauthorise, check |}
```

```
Controller =
  let
    State(s,A) =
      s != <> & goto!head(s) -> State(tail(s),A)
      []
      length(s) < 3 & ( request?f?p:A -> State(s^<f>,A)
                        []
                        call?f?p:A -> State(s^<f>,A) )
      []
      authorise?p -> State(s,union(A,{p}))
      []
      deauthorise?p -> State(s,diff(A,{p}))
      []
      check?p:A!yes -> State(s,A)
      []
      check?p:diff(PERSON,A)!no -> State(s,A)
  within
    State(<>,{})
```

Building

These processes may be combined to produce an account of the sequences of actions that are possible within the building:

```
Building =
  (Lift [aLift || aController] Controller)
  [ union(aLift,aController) || aFloors ]
  Floors
```

Following the experience of the revolving door, we might be reassured to learn that our building is free from deadlocks

```
assert Building :[deadlock free]
```

It is not, however, secure. If we define

```
Secure =
  let
    Authorised(A) =
      ([] e : {| enterBuilding, exitBuilding, enterLift,
         exitLift.0, goto, call, request, open,
           close, check |} @ e -> Authorised(A))
      []
      authorise?p -> Authorised(union(A,{p}))
      []
      deauthorise?p -> Authorised(diff(A,{p}))
      []
      exitLift.1?p:A -> Authorised(A)
  within
    Authorised({})
```

then the following check fails:

```
assert Secure [T= Building
```

What is the problem? And how might we fix it?

Hint

As well as 'enhancing' the `Lift` process, you may need to decide that we should not consider situations in which e.g. an authorised person were to be deauthorised while on the secure floor.

You can write an assumption process to prevent such a thing from happening, and thus rule the corresponding behaviours out of consideration.

Your solution could look something like this:

```
SmartBuilding =
  (SmartLift [aLift || aController] Controller)
  [ union(aLift,aController) || aFloors ] Floors

assert Secure [T= ( Building
                    [aBuilding || aAssumptions]
                    Assumptions )
```