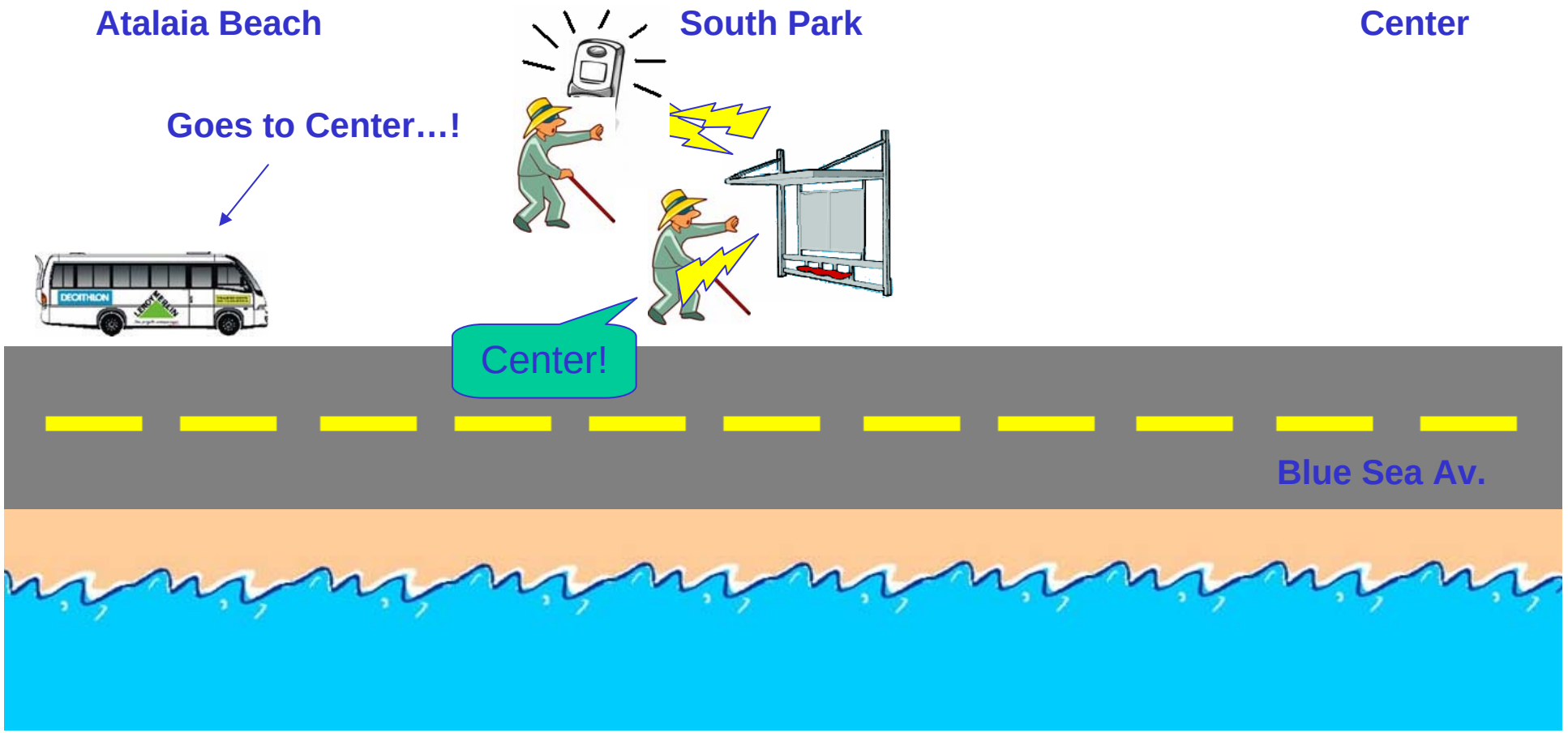
Introduction to Communicating Sequential Process (CSP) (Lecture 10)

Mannheim, September 2007

Contents

- Case Study
  - Ribeiro, Márcio M. (2006)

Blind Helping in Bus stop



The Problem

- To identify buses in a bus stop is hard for blind people.
- The system try to minimize this problem through a mechanism, using mobile phones, that notifies a blind person when a bus has arrived in the bus stop.
- The system has the following behaviour:
  - A blind person in a bus stop accesses his mobile phone and communicates, by voice, his current localization and the target he wants.
  - The mobile phone communicates with the bus stop, sending the celular number.
  - When a bus arrives in the bus stop, the bus sends a signal to the bus stop. The bus stop notifies all mobile phones it has access that the bus has arrived.
  - The mobile phone verify if the bus is the one required by its owner. In affirmative case it vibrates. Otherwise, it continues to wait for another bus.

Specification

- Type declarations
- Constant declarations
- Auxiliary functions
- Channel declarations
- Main Processes
- Synchronization Events
- System

Specification

- Verification using FDR
  - Deadlock
  - Livelock
  - Determinism
- Refinements

Specification

- Types

```
-- Celular Number
id: {0..2} -- FDR does not deal with infinite type
-- names of the bus stop
datatype stopNames = AtalaiaBeach | CashewtPark | Center
```

- Constants

```
mobilesPerBusStop = 3
placesList = <AtalaiaBeach, CashewPark, Center>
(the bus route folllows this sequence of place)
```

Specification

- Auxiliary Functions (to be defined later)
- Channels

  -- to communicate the identifier, the localization and the target by the blind person

  channel blindData: id.stopNames.stopNames

  -- to communicate the blind data to the bus stop

  channel infoMobile: stopNames.id.stopNames

Specification

- Channels

  -- to communicate the bus stop that the bus has arrived. The bus localization and target are sent.

  channel busArrived: stopNames.stopNames

  -- to notify the mobile phones the arrival of a bus. It synchronises with the mobile identifier and sends the bus information.

  channel notifyMobiles: id.stopNames.stopNames

Specification

- Channels

  -- to communicate the blind give up return home. Its is necessary to maintain a consistent list of mobiles in each bus stop.

  channel blindGiveUp: id

  -- to communicate the bus to move (synchronises with the environment)

  channel move: stopNames

Specification

- Channels

  -- to vibrate the mobile

  channel vibrateMobile: id

  -- to communicate the bus it is at the last stop

  channel lastStop: stopNames

  -- to communicate the traffic is good and the bus can move (synchronises with the environment)

  channel goodTraffic: stopNames

  -- to communicate the traffic is bad and the bus must stay at the bus stop

  channel badTraffic: stopNames

Specification

- Channels

  -- to remove a mobile phone from the bus stop list.

  channel removeMobileBusStop: stopNames.id

Specification

- Processes
  - MOBILE_PHONE
  - BUS_STOP
  - BUS

Specification

- MOBILE_PHONE
  - Actions:
    - Mobiles try to register at a bus stop. If there is space available, the register is performed and the blind wait for the bus.
    - The blind can give up and notify this using the mobile.
    - The mobile vibrates if the bus that arrives satisfies the blind needs. Otherwise the blind stays waiting for another bus.

Specification

```
MOBILE_PHONE(id) =
        blindData.id?place?blindTarget ->
                if (place == blindTarget) then MOBILE_PHONE(id)
                else infoMobile.place!id!blindTarget ->
                        MOBILE_WAITING(id, place, blindTarget)


MOBILE_WAITING(id, place, blindTarget) =
        notifyMobiles.id.place?busTarget ->
                if (BusTarget == blindTarget) then
                    vibrateMobile.id -> removeMobileBusStop.place.id -> MOBILE_PHONE(id)
                else MOBILE_WAITING(id, place, blindTarget)
        []
        blindGiveUp.id -> removeMobileBusStop.place.id -> MOBILE_PHONE(id)
```

Specification

- BUS_STOP
  - Actions:
    - Receives data from the mobile phone
    - Receives data from the bus and immediately notifies the mobile phones
    - Removes mobile numbers from the list it maintains.

Specification

```
BUS_STOP(place, mobiles) =
        (infoMobile.place?mobile?blindTarget ->
                if ((#mobiles > mobilePerBusStop) or (elem(mobile, mobiles)))
                then  BUS_STOP(place, mobiles)
                else   BUS_STOP(place, <mobile>^mobiles)

         []

         busArrived.place?busTarget ->
                (NOTIFY_MOBILES(place, mobiles, busTarget) ;
                                                        BUS_STOP(place, mobiles))
         []
         removeMobileBusStop.place?id ->
                BUS_STOP(place, removeMobile(id, mobiles))
```

Specification

```
NOTIFY_MOBILES(place, mobiles, busTarget) =
        if (length(mobiles) == 0)
        then SKIP
        else notifyMobiles.head(mobiles).place!busTarget ->
                NOTIFY_MOBILES(place, tail(mobiles),busTarget)
```

Specification

- BUS
  - Actions:
    - Each bus has a localization, a route to perform and a target. The route can be in the same order of placeList or in reverse order. At each bus stop the route is updated to remove the place already visited.
    - The route is fixed for a given line bus.
    - The bus moves if the traffic is good, otherwise stays at the bus stop.
    - The bus notifies the bus stop when it arrives.

Specification

```
BUS(busTarget, place, route) =
    move.busTarget -> BUS_MOVING(busTarget, place, route)

  BUS_MOVING(busTarget, place, route) =
    goodTraffic.busTarget ->
            if (busTarget == head(route)) then
                    lastStop.busTarget ->
                            BUS(setTarget(busTarget), busTarget, setRoute(busTarget))
            else
                    busArrived.head(route)!busTarget ->
                            BUS_MOVING(busTarget, head(route), tail(route))

    |~|

    badTraffic.busTarget -> BUS_MOVING(busTarget, place, route)
```

Specification

- Synchronization events

  Events_Mobile_BusStop =

  {|infoMobile, notifyMobiles,

  blindGiveUp, removeMobileBusStop|}

  Events_BusStop_Bus = {|busArrived|}

Specification

- System

MOBILES = ||| i: id @ MOBILE(i)
BUS_STOPS = ||| k: stopNames @ BUS_STOP(k, <>)
BUSES = ||| d: stopNames @ BUS(d,
head(placeList), placeList))

SISTEMA = MOBILES [| Events_Mobile_BusStop |]
(BUS_STOPS [| {| busArrived |} |] BUSES)

Specification

- Refinement

  Change |~| by [] in BUS and prove that the new process BUS_MODIFIED refines the original one

  assert BUSES [FD=BUSES_MODIFIED

Specification

- Refinement . The concept of last stop does not concern to a TAXI.

```
BUS(busTarget, place, route) =
        move.busTarget -> BUS_MOVING(busTarget, place, route)

    BUS_MOVING(busTarget, place, route) =
      goodTraffic.busTarget ->
                if (busTarget == head(route)) then
                        lastStop.busTarget ->
                                BUS(setTarget(busTarget), busTarget, setRoute(busTarget))
                else
                        busArrived.head(route)!busTarget ->
                                BUS_MOVING(busTarget, head(route), tail(route))

      |~|

      badTraffic.busTarget -> BUS_MOVING(busTarget, place, route)
```

Specification

```
TAXI(busTarget, place, route) =
        move.busTarget -> TAXI_MOVING(busTarget, place, route)

TAXI_MOVING(busTarget, place, route) =
        goodTraffic.busTarget ->
                if (busTarget == head(route)) then
                        TAXI(setTarget(busTarget), busTarget, setRoute(busTarget))
                else
                        busArrived.head(route)!busTarget ->
                                TAXI_MOVING(busTarget, head(route),
    tail(route))

        |~|

        badTraffic.busTarget -> TAXI_MOVING(busTarget, place, route)
```

Specification

- Refinement

  TAXIS = ||| d: stopNames @ TAXI(d, head(placeList), placeList))

  assert BUSES \{| lastStop |} [FD= TAXIS

Specification

- Refinement. Suppose now that the traffic has no interest to motos.

```
TAXI(busTarget, place, route) =
        move.busTarget -> TAXI_MOVING(busTarget, place, route)

TAXI_MOVING(busTarget, place, route) =
        goodTraffic.busTarget ->
                if (busTarget == head(route)) then
                        TAXI(setTarget(busTarget), busTarget, setRoute(busTarget))
                else
                        busArrived.head(route)!busTarget ->
                                TAXI_MOVING(busTarget, head(route),
    tail(route))

        |~|

         badTraffic.busTarget -> TAXI_MOVING(busTarget place, route)
```

Specification

```
MOTO(busTarget, place, route) =
        move.busTarget -> MOTO_MOVING(busTarget, place, route)

MOTO_MOVING(busTarget, place, route) =
        if (busTarget == head(route)) then
                MOTO(setTarget(busTarget), busTarget, setRoute(busTarget))
        else
                busArrived.head(route)!busTarget ->
                        MOTO_MOVING(busTarget, head(route), tail(route))

MOTOS = ||| d: stopNames @ MOTO(d, head(placeList), placeList))

assert TAXIS \{| goodTraffic, badTraffic |} [T = MOTOS
assert BUSES \{| goodTraffic, badTraffic, lastStop |} [T = MOTOS
        (by transitivity)
```

Auxiliary Functions

- There are pre defined functions common to any functional language: head, tail, length, elem…
- The functions are defined as in functional programming, using pattern matching

```
removeMobile(id, <>) = <>
removeMobile(id, mobiles) =
    if (id == head(mobiles)) then tail(mobiles)
    else <head(mobiles)> ^ removeMobile(id, tail(mobiles))

setTarget(name) =
    if (name == head(placeList) then head(reverse(placeList))
    else name

setRoute(name) =
    if name == head(placeList) then placeList
   else reverse(placeList)

reverse(lista) = reverse(tail(lista))^<head(lista)>
```