Excercise 10

Concurrency Theory

Stephan Spengler Johannes Freiermuth

January 27, 2018

1 Fifo Chanel with π-calculus

a) Control Flow

The control flow of a communicating machine $M = (S, \delta, \rightarrow, s_0)$ can be simulated easily by an CCS: For every state in $s \in S$ a process is created. The outgoing transitions are represented by a sum.

$$s() := \sum_{s \xrightarrow{m} s'} m.s'().$$

$s_0()$ is the CCS' initial state.
This construction is obviously finite and takes all possible control flows into account since all choice are represented by sums.

b) FIFO

To implement FIFOs we implement queues. Each queue has enqueue, dequeue and is-empty functionality, accessible via the names deq, enq and empty. head is the first entry in the queue. tail is the name (pointer) to the queue containing all the other elements in the list. this is the name of this very queue.
The first line (in den code below) just lets other process read this queue's personal access names enq, deq and empty. Then it recreates a unchanged situation.
If deq(head, tail) can be sent, the head of this queue is consumed. tail gives the consumer access to the remaining list. Since this part of the queue now is consumed, there is nothing left to recreate.
Is enq(new) consumed the element new shall be enqueued. Therefore we have to reach the queue's end. By calling !tail(tailEnq, tailDeq, tailEmpty) we get to know the tail's enqueue name, tailEnq. This can by called (sent) together with the element to be added. Finally we have to recreate the current situation. To handle the base case of this recursion we will add another process for the empty queue.

```
Queue(this, head, tail, enq, deq, empty)
  = !this(enq, deq, empty).Queue(this, head, tail, enq, deq, empty)
  + !deq(head,tail).0
  + ?enq(new)
    .?tail(tailEnq, tailDeq, tailEmpty)
    .!tailEnq(new)
    .Queue(this, head, tail, enq, deq, empty)
```

Since the empty queue does not contain any content, head and tail are not needed.
The first line again gives access to the access-function-names.
The empty queue cannot dequeue anything. Instead it sends !empty() to signalize that there is nothing to dequeue.
If the enqueue gets triggered, we can really apply the enqueue here since this is the very end of the queue. We have to enlarge it by an element. This element needs a name (nThis) and new nEnq, nDeq and nEmpty names. Those are restricted.
The new Queue, containing the new element as head, is named this. Thus, viewed from the outside, the new queue silently replaces the former empty-queue. In parallel there is new empty queue with the new names. Since this (nthis) is registered as the new queue's tail, it again got a valid end.

```
EmptyQueue(this, enq, deq, empty)
  = !this(enq, deq, empty).EmptyQueue(this, enq, deq, empty)
  + !empty().EmptyQueue(this, enq, deq, empty)
  + ?enq(new).(v nThis, nEnq, nDeq, nEmpty).(
      Queue(this, new, nThis, enq, deq, empty)
    | EmptyQueue (nThis, nEnq, nDeq, nEmpty)
    )
```

Now we need a managing process, representing a channel. This channel has a queue.
A send is consumed, if some msg was sent via this channel. The message simply gets enqueued.
If get is consumed the channel dequeues and returns the first element of the queue via front, or empty if there is none. Then the Channel is recreated, maybe with another queue.

```
Channel(this, queue, send, get, front, empty)
    = !this(send, get, front, empty)
      .Channel(this, queue, send, get, front, empty)
    + ?send(msg).?queue(qEnq, qDeq, gEmpty).!qEnq(msg)
      .Channel(this, queue, send, get, front, empty)
    + ?get().?queue(qEnq,qDeq,qEmpty).(
        ?qEmpty().!empty().Channel(this, queue, send, get, front, empty)
      + ?qDeq(head, tail).!front(head)
        .Channel(this, tail, send, get, front, empty)
      )
```

There is one more process, giving comfortable access to the queue when consuming its messages. This is the process Sender. As long as the queue is not empty it tries to send the first element till it gets consumed. In any case it simply tries again.

```
Sender(this, channel)
    = ?channel(send,get,empty).!get()(
        ?empty().Sender(this, channel, consume)
      + ?front(msg).!msg.Sender(this, channel, consume)
      )
```

The following process makes the initialising comfortable. We simply need to pass a name for the send functionality. Initially there is an empty queue q used by a channel ch used by a sender s.

```
Init (send)
=(v q, d, e, em, ch, g, f, emp, s) (
    EmptyQueue(q, d, e, em)
  | Channel(ch, q, send, g, f, emp)
  | Sender(s, ch)
)
```

In general ?a remains untouched. !a becomes !send(a).
To run !a.?a.0 asynchronously we can do Init(send)|!send(a).?a.0
It remains to ensure that messages send to, say machines a's channel, cannot be read by machines b. This can be solved by assuming that the sets of message recieved by each machine are disjoint. This might by realised by adding the reciever machine's name as prefix in front of any message.

c) Example

Let us first define the controle flow Processes and replace reciever!a bei !reciever(a) (similar for sender).

```
Send1()=!reciever(Msg0).Send1()
        +?Ack1.Send1()
        +?Ack0.Send2()

Send2()=!reciever(Msg1).Send2()
        +?Ack0.Send2()
        +?Ack1.Send1()

Reciever1()=!sender(Ack1).Reciever1()
            +?Msg1.Reciever1()
            +?Msg0.Reciever2()

Reciever2()=!sender(Ack0).Reciever2()
            +?Msg0.Reciever2()
```

```
                +?Msg1.Reciever1()
```

We simply need to initialize the two channels and run the two inital states:

```
(v send, reciever)(
  Init(sender) | Init(reciever) | Send1() | Reciever1()
)
```