Modeling a problem in π

January 19, 2004

Abstract

In this session, we propose you to model a problem in π-calculus and use the ABC to solve some instances of the problem.

1 Original Problem (*parallel game*)

n players (with $n \geq 3$) are arranged in a circle. Among them $n-1$ coins are distributed. At each stage in the game those players with more than one coin give one to each neighbour. Show that the game terminates.

2 Studied Problem (*sequential game*)

n players (with $n \geq 3$) are arranged in a circle. Among them $n-1$ coins are distributed. At each stage in the game, a randomly chosen player with more than one coin gives one coin to each of his neighbour. Show that the game terminates.

We represent a configuration of the game with a tuple of n integers $(c_1, \ldots, c_n)$ where c_i is the number of coins the player i owns.

example: Here follows a possible trace of the sequential game with initial configuration $(0, 2, 2, 0, 1)$.

$$(0, 2, 2, 0, 1) \to (1, 0, 3, 0, 1) \to (1, 1, 0, 1, 1)$$

3 Modeling the problem

We want to model the problem in π-calculus. The idea is to represent each player with one agent `Player` and to link them together in order to represent a game.

3.1 Hardcoding the coins

To simplify, we first decide to encode the number of coins into the name of the agents. So, instead of having only one agent `Player`, we have a family of agents $\texttt{Player}_n$ where n is the number of coins the player has at current time.

Each agent communicates on two channels: `left` and `right`. A player uses the channel `left` to communicate with the player on his left and the channel `right` to communicate with the player on his right.

Here is the ABC code for $\texttt{Player}_n$ when $n = 0$ or $n = 1$:

```
agent Player_0(left,right) = left.Player_1(left,right) \
                           + right.Player_1(left,right)
agent Player_1(left,right) = left.Player_2(left,right) \
                           + right.Player_2(left,right)
```

Could you give the code for $\texttt{Player}_2$? More generally, what is the body of $\texttt{Player}_n$ when $n \geq 2$?

What is the problem with hardcoding the number of coins in the name of the agents?

3.2 Church encoding

Now we have seen the general scheme for encoding a player, we want to model fully the behaviour of a player. To achieve this goal, we represent a player by an integer (Church encoded) and an agent that manages this integer.

First, we start from the standard unary encoding of integers in the π-calculus.

```
agent Zero(l) = l(z,s).'z.0
agent Succ(l,n) = l(z,s).'s<n>.0
```

You may have noticed that these integers are not *persistent*. Try to change your definitions in order to have almost persistent integers (without using unguarded recursion). The idea is to see the integer 0 located at l as a server that listen on the channel l and if it receives two names z and s, then he sends a signal to z and loop. Try to adapt this idea to the case of the successor and change your definitions in ABC.

Now you have defined these two base agents, define an agent `Ge2(n,true,false)` that tests whether the integer located at `n` is greater than or equal to 2 or not. If the integer located at `n` is greater than or equal to 2, then your agent should send on the channel `true` the location of the predecessor of the predecessor of this integer and otherwise, it should send a signal on the channel `false`.

```
agent Ge2(n,true,false) = ??
```

Now that all the material is ready, define the agent `Player(left,right,n)` parametrised by the two channels `left` and `right` and the location `n` of the integer that encodes the number of coins the player owns. Here is the start of your answer:

```
agent Player(left,right,n) = left.(??)
                           + right.(??)
                           + (^caseTrue,caseFalse) \
                               ( Ge2(n,caseTrue,caseFalse) \
                               | ( caseFalse.(??) \
                                 + caseTrue(n).(??)))
```

3.3 A game

Now that the agent that represents players is defined, give an agent representing the configuration $(2, 0, 0)$ (to ease this, define an agent `One(l)` and an agent `Two(l)`)

```
agent One(l) = (^k)(Succ(l,k) | Zero(k))
agent Two(l) = (^k)(Succ(l,k) | One(k))
```

What is the general form of a configuration of the game?

```
agent Game = ...
```

3.4 Termination

3.4.1 Infinite computation

Give a closed process `Inf` that behaves like:
$\texttt{Inf} \xrightarrow{\tau} \texttt{Inf} \xrightarrow{\tau} \cdots \xrightarrow{\tau} \texttt{Inf} \xrightarrow{\tau} \texttt{Inf} \xrightarrow{\tau} \cdots$

3.4.2 Solving a particular game

The system that represents a game configuration is a closed process. Thus, it can only evolve by performing internal transitions.

How would you model the fact that all reductions of the (sequential) game terminate? (think of the meaning to be bisimilar to `Inf`, to simulate `Inf` or to be simulated by `Inf`)

Try your guess with ABC with a particular initial configuration.

3.4.3 Fixing a problem in players

A possible problem with the definition of the agent `Player` is that even if a player has less than 2 coins, it can perform some internal computation (test if he has more or less than 2 coins) and loops on itself (do nothing since he has less than 2 coins).

The idea to fix this "problem" is to have 2 agents to represent a player. One is a *normal* player, and one is a *lazy* player. The normal player acts exactly as above but instead of looping on himself when he has less than 2 coins, he becomes a lazy player. Thus, a lazy player should represent only a player with less than 2 coins and then he only listen to his neighbours for receiving a coin but do not test if he has less or more than 2 coins.

Modify your agent `Player` and define an agent `LazyPlayer`.

```
agent Player(left,right,n) = left.(??) \
                           + right.(??) \
                           + (^caseTrue,caseFalse) \
                                ( Ge2(n,caseTrue,caseFalse) \
                                | ( caseFalse.LazyPlayer(left,right,n) \
                                  + caseTrue(n).(??)))

agent LazyPlayer(left,right,n) =  left.(??) \
                                + right.(??)
```

Retry to solve question 3.4.2.