Tree self-assembly in pi-calculus

Luca Cardelli 2014-06-15

A couple of exercises in stochastic pi-calculus, showing local self-assembly of binary trees from an initial soup of uniform components.

Pi-calculus has very few but very powerful primitives. You can create new communication channels (via 'new') over which you can send (!) and receive (?) messages. The messages you send over channels are … more channels (ok, booleans and integers too, but we don't need those). Communication is binary and synchronous: at least one sender and one receiver must be attempting to use a given channel for a message to pass from exactly one of the senders to exactly one of the receivers. The things creating channels and exchanging messages are called 'processes': think 'agents' or finite state machines, where each communication event can result in a change of state. Processes can be listening simultaneous to multiple channels (via 'or') and communicate on whichever channel has a partner on the other side. Processes run 'in parallel', and can also 'spawn' new processes (not used here). The order of communication events is nondeteministic, but in the stochastic pi-calculus variant there are rates associate with channels that determine how often they can 'fire': the semantics is then a continuous time markov chain, where transitions are communication events, and a Gillespie-style algorithm can be used for simulations.

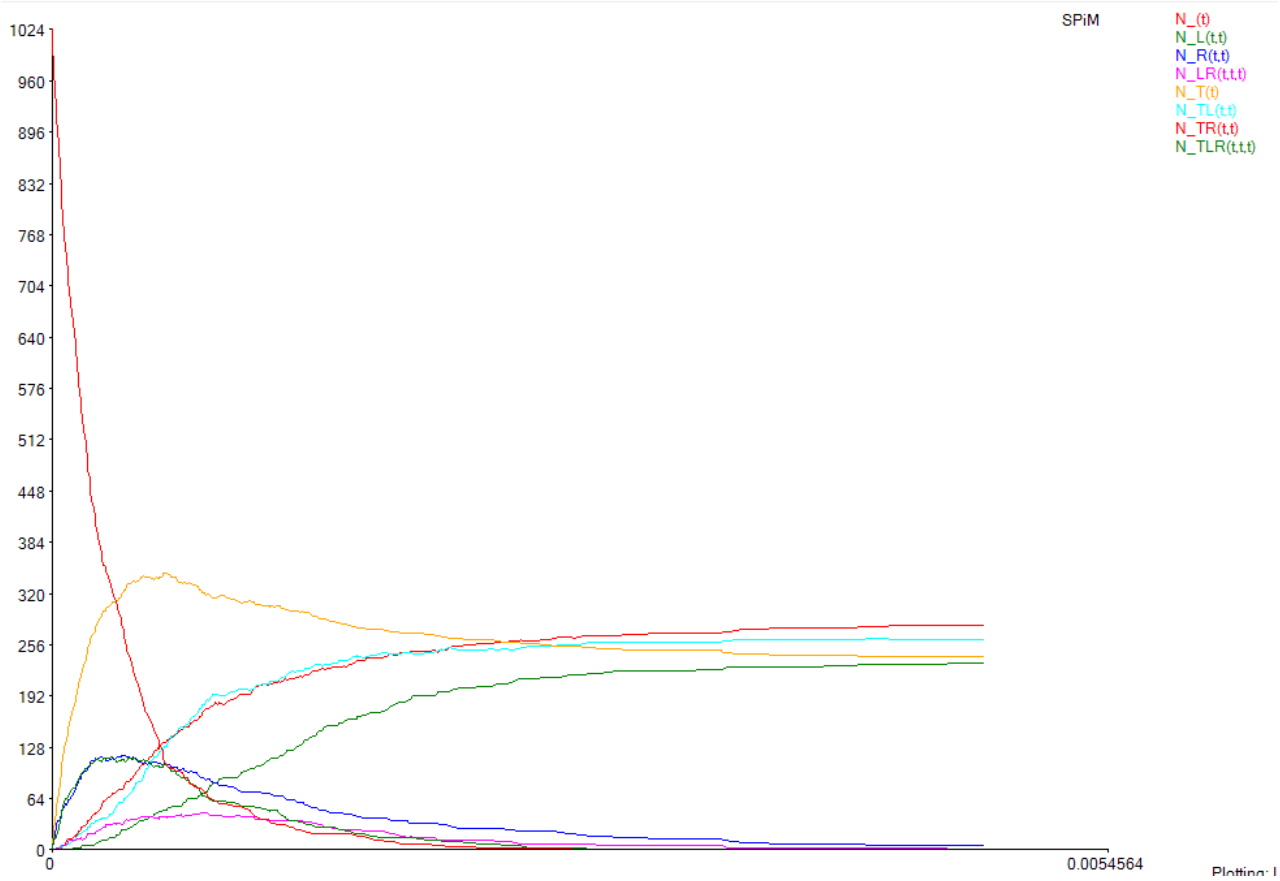
One unusual thing we can do with pi-calculus is to model arbitrary 'complexation' (and polymerization) of molecules (NATURE|VOL 419|26 SEPTEMBER 2002 http://www.wisdom.weizmann.ac.il/~lbn/other_links/aviv_udi.pdf). By dynamically creating channels and passing them around one can effectively create any graph-like dynamical structure of connected molecules. A 'bond' between molecules is represented by them sharing a unique channel, and breaking the bond is represented by both parties agreeing to forget the channel. Moreover, communication is available on this network of 'bond' channels: the communication infrastructure can be used to coordinate actions between interconnected molecules, or to break the bonds.

Exercise 1: Floppy trees

The program below describes a 'soup' of initially identical molecules; we program them to gradually interconnect into a global tree-like structure. Each molecule acts autonomously and the only knowledge it has of its environment is encoded in the channels it knows: either global channels (here the single 'soup' through which it can bump into other molecules) or local channels 'binding' it to other molecules. Each molecule has three potential attachment points: top,left,right to interconnect into a tree structure. In the first (easier) solution below, we model 'floppy' trees: each molecule is prevented from attach to itself (e.g. left to top) by pi-calculus rules, but a tree made of multiple molecules can 'twist' and have one of its leafs attaching to its root. That happens if we start from a completely symmetric state where any bottom can attach to any other top: this ability persists even when molecules are already attached to each other. So this solution will (most likely) result in assembling multiple trees, each with one of its leaves attached to its root.

Since there are 3 attachment point per molecule, and each can be free or bound to another molecule, each molecule has 8 possible states. The program below is therefore organized in 8 recursive process definitions (plus N_Gen to generate new molecules/nodes). For example N_TL(t,l) is the state of a Node (N_TL) already attached at the Top and Left to other molecules, via the t and l channels respectively; the right attachment point is still free. The t and l channels are each uniquely shared with one other molecule (this is an invariant of the program below, not of pi-calculus). This particular N_TL node has nothing left to do but listen to the soup waiting for some other molecule who might want to attach to it on the available right binding site. Using a stochastic simulator, we can for example plot the number of molecules over time that are in the N_TL(x,y) state for any x,y.

```
-----------------------------------------------------------------
(* Run with http://research.microsoft.com/en-us/downloads/992f59a0-c8c2-40bc-ab25-34516cf132c9/default.aspx *)
directive sample 0.005 1000
directive plot N_(t); N_L(t,t); N_R(t,t); N_LR(t,t,t); N_T(t); N_TL(t,t); N_TR(t,t); N_TLR(t,t,t)

type Bond = chan                        (* a bond ('channel') between nodes *)
type Soup = chan(Bond)                          (* a chemical soup (another channel) through which bonds are exchanged *)

new c@1.0:Soup                          (* create the 'new' unique soup 'c', which has interaction rate 1.0 *)

let N_Gen() =                                   (* Generate a new node *)
  (new t@1.0:Bond run N_(t))            (* create a new node N_(t) with a new (unique) top bond 't' that it can use to bind to other nodes *)

and N_(t:Bond) =                        (* A free node N_ with a top bond 't' available but still unattached *)
  do !c(t); N_T(t)                      (* Give (!) the 't' bond to somebody in the soup, then (;) transition to a node bound at the top N_T *)
  or ?c(l); N_L(t,l)                    (* Or receive (?) somebody else's top bond as 'l' and transition to a node bound at the left N_L *)
  or ?c(r); N_R(t,r)                    (* Or receive somebody's top bond as 'r' and transition to a node bound at the right N_R *)

and N_L(t:Bond,l:Bond) =                (* A node bound only at the left (via 'l') with bond 't' still available *)
  do !c(t); N_TL(t,l)                   (* Give the 't' bond to somebody, then transition to a node bound at the top and left N_TL *)
  or ?c(r); N_LR(t,l,r)                 (* Or receive somebody's top bond as 'r' and transition to a node bound at the left and right N_LR *)

and N_R(t:Bond,r:Bond) =                (* A node bound only at the right (via 'r') with bond 't' still available *)
  do !c(t); N_TR(t,r)                   (* Give the 't' bond to somebody, then transition to a node bound at the top and right N_TR *)
  or ?c(l); N_LR(t,l,r)                 (* Or receive somebody's top bond as 'l' and transition to a node bound at the left and right N_LR *)

and N_LR(t:Bond,l:Bond,r:Bond) =        (* A node bound at the left and right with bond 't' still available *)
  !c(t); N_TLR(t,l,r)                   (* Give the 't' bond to somebody, then transition to a node bound at the top,left,right N_TLR *)

and N_T(t:Bond) =                       (* A node bound only at the top (via 't') *)
  do ?c(l); N_TL(t,l)                   (* Receive somebody's top bond as 'l' and transition to a node bound at the top and left N_TL *)
  or ?c(r); N_TR(t,r)                   (* Receive somebody's top bond as 'r' and transition to a node bound at the top and right N_TR *)

and N_TL(t:Bond,l:Bond) =               (* A node bound at the top and left *)
  ?c(r); N_TLR(t,l,r)                   (* Receive somebody's top bond as 'r' and transition to a node bound at the top,left,right N_TLR *)

and N_TR(t:Bond,r:Bond) =               (* A node bound at the top and right *)
 ?c(l); N_TLR(t,l,r)                    (* Receive somebody's top bond as 'l' and transition to a node bound at the top,left,right N_TLR *)

and N_TLR(t:Bond,l:Bond,r:Bond) =       (* A node bound at the top,left,right *)
  ()                                            (* Nothing to do *)

run 1024 of N_Gen()                     (* Create 1024 copies of nodes in the N_ initial state and let the run *)
-----------------------------------------------------------------
```



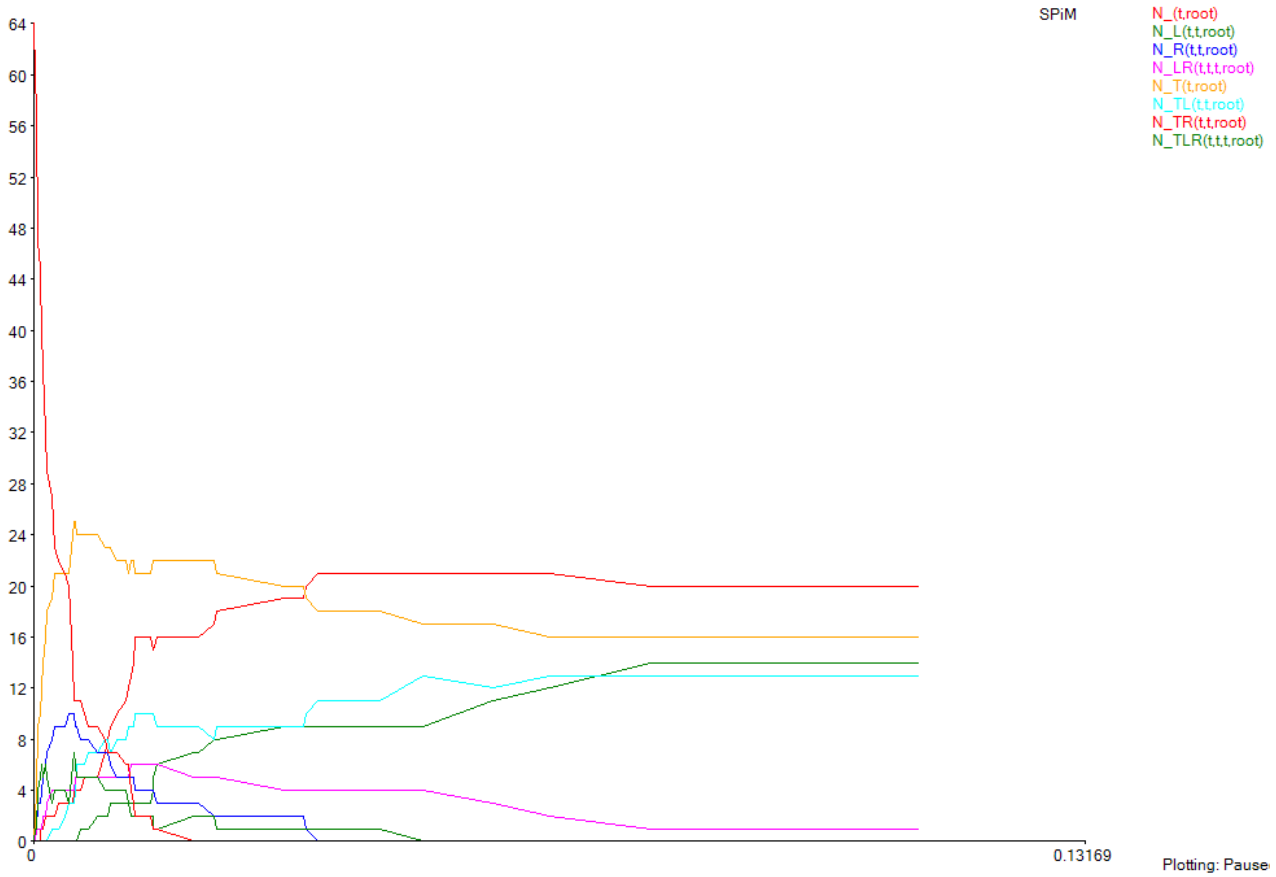
In this simulation 1024 nodes self-assemble; the bottom 4 traces are the 4 different 'root' states, and the 4 top traces are the 4 different 'non-root' states. The root states go to zero because of cycles being formed; the population appears to converge to a roughly equal mixture of the 4 non-root states. (Why?)

Exercise 2: Rigid trees

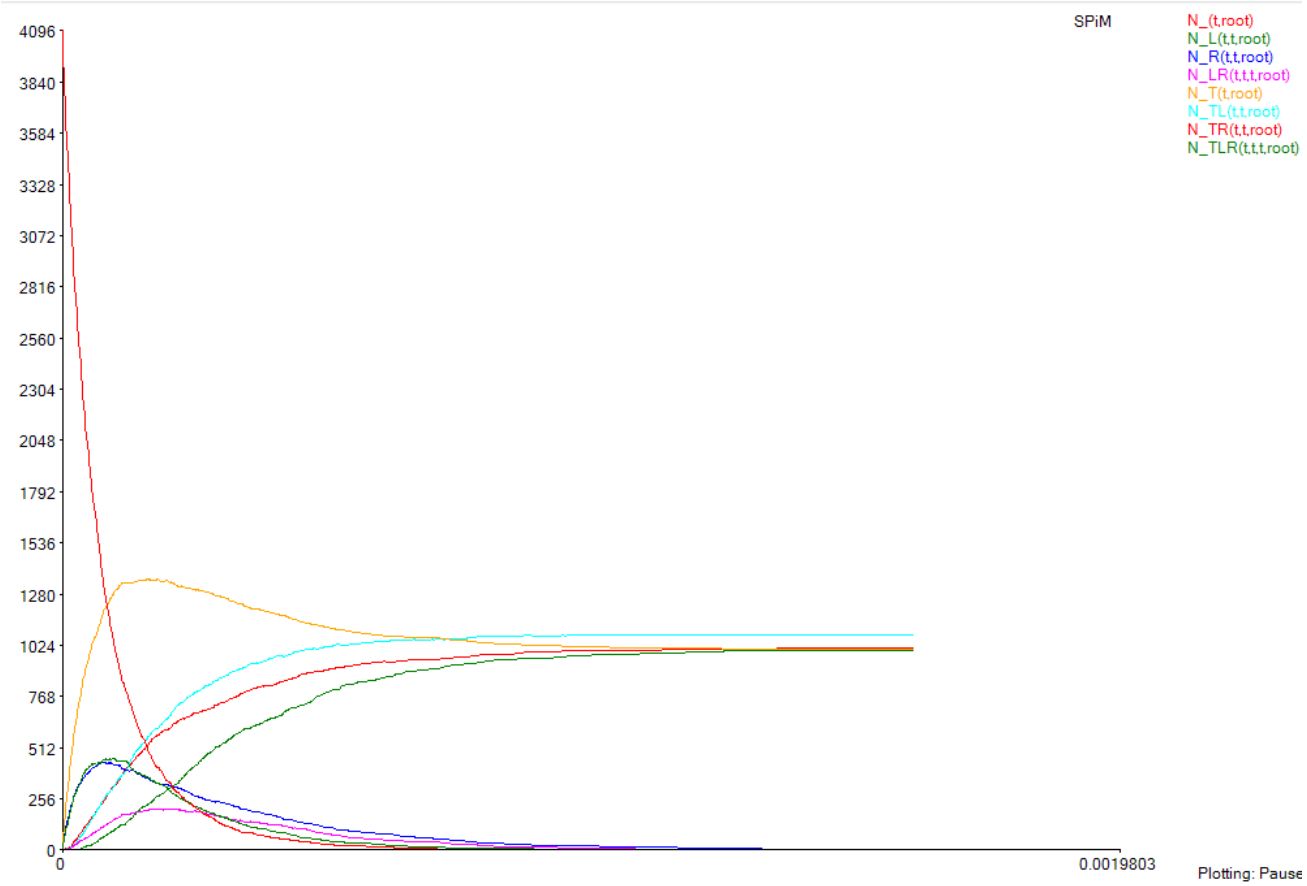
We now want to model 'rigid' trees where the bottoms do not attach to their own tops. If this is was to be true 'in reality', it would probably be because of mechanical constraints, like rigid tiles. Here we are going to enforce rigidity in an artificial way, which involves locally storing and communicating O(log n) information. We are going to use more extensively the communication infrastructure, running communication algorithms over the trees while they are assembling. We use a trick of the simulator to run these protocols at 'infinite speed' w.r.t. the speed of assembly (otherwise we would need more complicated protocols). This is dangerous because if the infinite-speed protocols were to loop, they would throw the simulator in an infinite loop with no visible output. Infinite speed is expressed by not specifying a rate at all for a channel.

The solution below uses the same basic program structure as above, but each node in a partially assembled tree also 'knows' who is the root of its tree. When two molecules are about to connect, they first exchange information about their roots, and then check to see if their roots are the same, in which case they refuse to connect. If they do connect, the new root information is propagated through the rest of the newly assembled tree at 'infinite speed', over the same channels that interconnect the molecules.

```
-----------------------------------------------------------------------
(* Run with http://research.microsoft.com/en-us/downloads/992f59a0-c8c2-40bc-ab25-34516cf132c9/default.aspx *)
directive sample 1.0 10000
directive plot N_(t,root); N_L(t,t,root); N_R(t,t,root); N_LR(t,t,t,root); N_T(t,root); N_TL(t,t,root); N_TR(t,t,root); N_TLR(t,t,t,root)

type Root = chan                    (* not for communication *)
type Bond = chan(Root)              (* infinite speed communication *)
type Soup = chan(Bond)              (* finite speed communication *)

new c@1.0:Soup                      (* at rate 1.0 *)

let N_Gen() =                       (* create a new node N_(t,root) with a unique top bond 't' that it can use to bind to other nodes *)
  (new t:Bond new root:Root run N_(t,root)) (* and a unique 'root' token for itself and its children *)

and N_(t:Bond, root:Root) =
  do !c(t); ?t(uproot); !t(root); N_T(t, uproot)                                    (* exchange root information *)
  or ?c(l); !l(root); ?l(dnroot); N_L(t,l, root)
  or ?c(r); !r(root); ?r(dnroot); N_R(t,r, root)

and N_L(t:Bond,l:Bond, root:Root) =
  do !c(t); ?t(uproot); !t(root); if uproot = root then N_L(t,l, root) else !l(uproot); N_TL(t,l, uproot)     (* send a root update left *)
  or ?c(r); !r(root); ?r(dnroot); if dnroot = root then N_L(t,l, root) else N_LR(t,l,r, root)

and N_R(t:Bond,r:Bond, root:Root) =
  do !c(t); ?t(uproot); !t(root); if uproot = root then N_R(t,r, root) else !r(uproot); N_TR(t,r, uproot)   (* send a root update right *)
  or ?c(l); !l(root); ?l(dnroot); if dnroot = root then N_R(t,r, root) else N_LR(t,l,r, root)

and N_LR(t:Bond,l:Bond,r:Bond, root:Root) =
  !c(t); ?t(uproot); !t(root); if uproot = root then N_LR(t,l,r, root) else !l(uproot); !r(uproot); N_TLR(t,l,r, uproot)   (* send root updates *)

and N_T(t:Bond, root:Root) =
  do ?c(l); !l(root); ?l(dnroot); if dnroot = root then N_T(t, root) else N_TL(t,l, root)
  or ?c(r); !r(root); ?r(dnroot); if dnroot = root then N_T(t, root) else N_TR(t,r, root)
  or ?t(uproot); N_T(t, uproot)                                    (* receive a root update *)

and N_TL(t:Bond,l:Bond, root:Root) =
  do ?c(r); !r(root); ?r(dnroot); if dnroot = root then N_TL(t,l, root) else N_TLR(t,l,r, root)
  or ?t(uproot); !l(uproot); N_TL(t,l, uproot)                (* receive and resend a root update left *)

and N_TR(t:Bond,r:Bond, root:Root) =
  do ?c(l); !l(root); ?l(dnroot); if dnroot = root then N_TR(t,r, root) else N_TLR(t,l,r, root)
  or ?t(uproot); !r(uproot); N_TR(t,r, uproot)                (* receive and resend a root update right *)

and N_TLR(t:Bond,l:Bond,r:Bond, root:Root) =
  ?t(uproot); !l(uproot); !r(uproot); N_TLR(t,l,r, uproot)  (* receive and resend root updates *)

run 64 of N_Gen()
-----------------------------------------------------------------------
```



Here we simulate 64 nodes assembling into a single tree: in the end, there is a single copy of a root node N_LR.



Same, with 4096 nodes (there is a single copy of a root node left at the bottom, which happens to be N_R). When you change the number of nodes, you should readjust the sampling parameters in “directive sample x y” so that x is close to the interval you mean to plot, and y is e.g. 10000 (number of sample points), or you may not have enough sample points in your interval.

Exercise 3: Light-up the left spine

Given the solution in Exercise 2, it is now a relatively simple task for any root node to send an “on” signal down its left subtree and an “off” signal down its right subtree. Any other node that receives the signal, propagates it unchanged to its left subtree and sends “off” to its right subtree. This can be done while the tree is assembling, if done repeatedly.