Understanding Asynchronous Product Construction

Q: Draw the state space of this system of processes given by p1 and p2:

```plaintext
bit x=false, y=false, z=false;
active proctype p1()
{do
  :: x -> y=true -> z=true
  :: x=true
od}
active proctype p2()
{ z; y=false }

/***** let's annotate the control locations via comments, to get this: *****/
bit x=false, y=false, z=false;
active proctype p1()
{do /*l1*/
  :: x -> /*l2*/ y=true -> /*l3*/ z=true
  :: x=true
od}
active proctype p2()
{/*m1*/ z; /*m2*/ y=false; /*m3*/ }

These two processes together describe an interesting state space:

The state vector is: <pc of p1, pc of p2, x, y, z>
Initial state: <l1,m1,0,0,0>
I use '::' to indicate the do-od defined states
I use '||' to indicate the "branchings" due to parallel processes interleaving
I use "1:x" etc to indicate which process does what

<l1,m1,0,0,0> -- x=1 --> <l1,m1,1,0,0>
  :: <l1,m1,1,0,0> -- x=1 --> <l1,m1,1,0,0>
  :: <l1,m1,1,0,0> -- x --> <l2,m1,1,0,0>
<l2,m1,1,0,0> -- y=1 --> <l3,m1,1,1,0>
<l3,m1,1,1,0> -- z=1 --> <l1,m1,1,1,1>
  :: <l1,m1,1,1,1> -- 1:x --> <l1,m1,1,1,1>
  :: <l1,m1,1,1,1> -- 1:x=1 --> <l2,m1,1,1,1>
  ||
  <l1,m1,1,1,1> -- 2:y=0 --> <l1,m2,1,0,1>

<l2,m1,1,1,1> -- 1:y=1 --> <l3,m1,1,1,1>
  ||
<l2,m1,1,1,1> -- 2:y=0 --> <l2,m2,1,0,1>
<l3,m1,1,1,1> -- 1:z=1 --> <l1,m1,1,1,1>
  ||
<l3,m1,1,1,1> -- 2:y=0 --> <l3,m2,1,0,1>

... you can finish this up ...
```
Understanding Synchronous Product Construction

bit x=false, y=false;
proc p1() {
  do
    :: x=!x
    :: y=!y
  od
/*-- want: []<>(x&!y). Negated: <>[]x<y) --*/
never{
  do
    :: skip
    :: (x|y) -> break
  od;
  accept_lab: (x|y) -> goto accept_lab
}
State: x, y, turn (one of p1 for system, n for never). << >> is accept state

<0,0,n> --n:skip--> <0,0,p1>
<0,0,p1> --p1:--> <1,0,n>
  :: <1,0,n> --n:skip--> <1,0,p1>
  :: <1,0,n> --n:break--> <1,0,p1>
  :: <1,0,p1> --p1:--> <1,0,p1>
  :: <1,1,n> --n:go to --> <1,1,p1>
  :: <1,1,p1> --p1:--> <1,0,n>
  :: <1,0,n> --n:go to --> <1,0,p1>
Fairness Basics

- Justice (weak fairness): <>[]a --> []<>b
- Spin’s default weak fairness (process fairness): If a process is eventually henceforth enabled in a trace, then it is infinite often taken in that trace. (You can’t have a reachable cycle where a process is continuously enabled along the cycle but is never taken.)
- Compassion (strong fairness): []<>a --> []<>b


Tips for debugging SPIN models

- Run syntax check
- Put in the proctypes and no never (or a degenerate never that allows all moves)
- Keep a low depth-bound of say 500 steps
- Check safety verification
- Hit “run” and get a feel for the state-space
- Introduce a never, but make sure that it is total, i.e., it must have a move defined for all system states (else it will restrict the product-machine moves). Make sure that the liveness checking flag is on.
- If you are peeking into a process, make sure that the partial-order reduction message is not indicating “trouble” (sometimes it says that the run is unsound because POR has to be off; then do so and rerun)
- Run property verification; when you get an error indication (e.g., errors: 1) in the xspin window, try to reduce the DFS depth to the smallest value that still generates an error
- Do an error-trail simulation, setting all the items for observation.