Artificial Biochemistry

Combining Stochastic Collectives

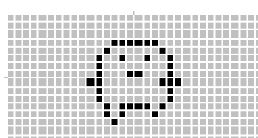
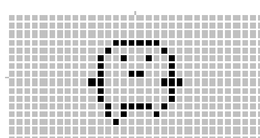
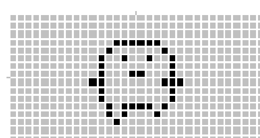
Luca Cardelli

Microsoft Research
MSR-UniTN CC&SB

Trento 2006-04-03

www.luca.demon.co.uk

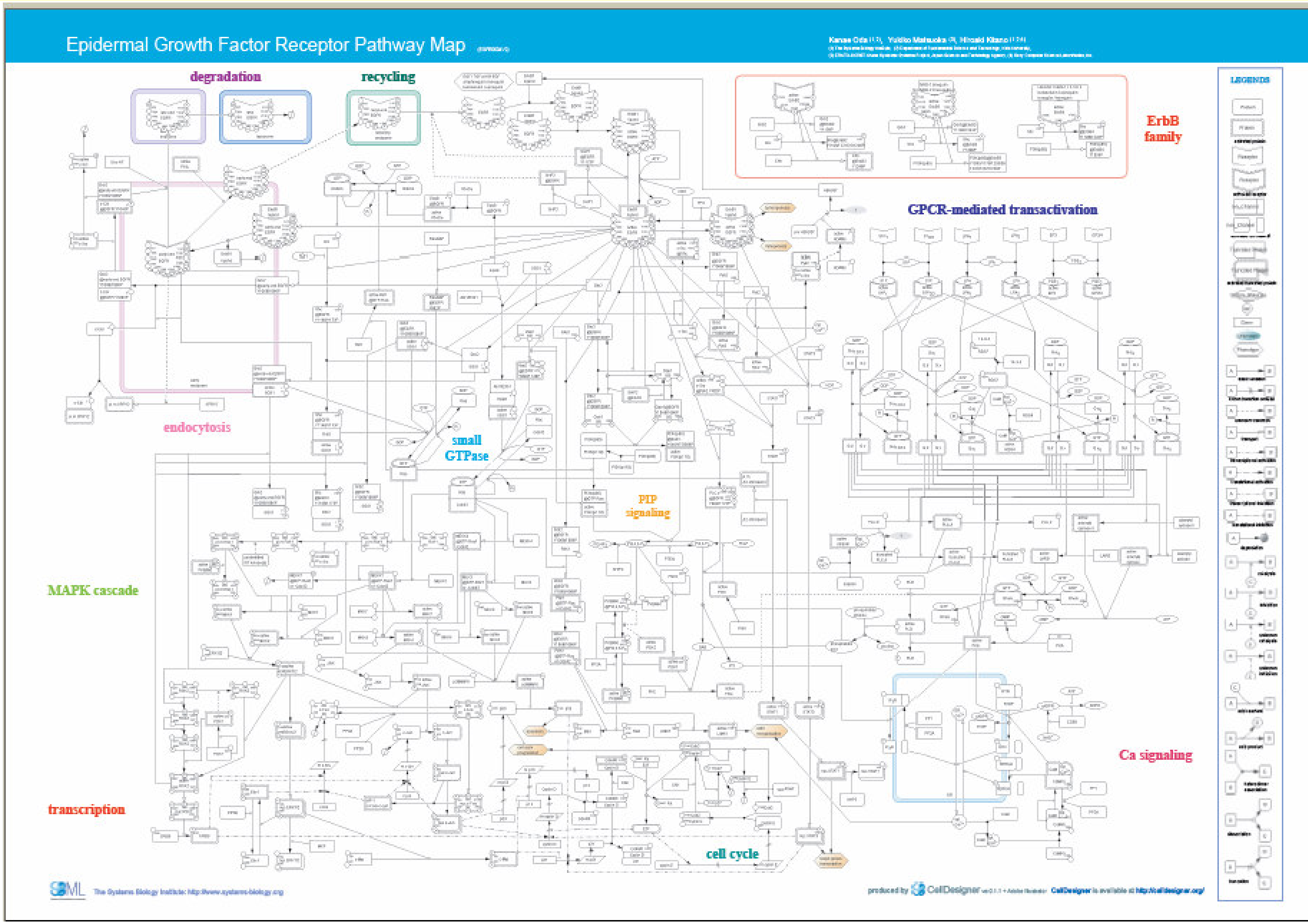
Stochastic Collectives

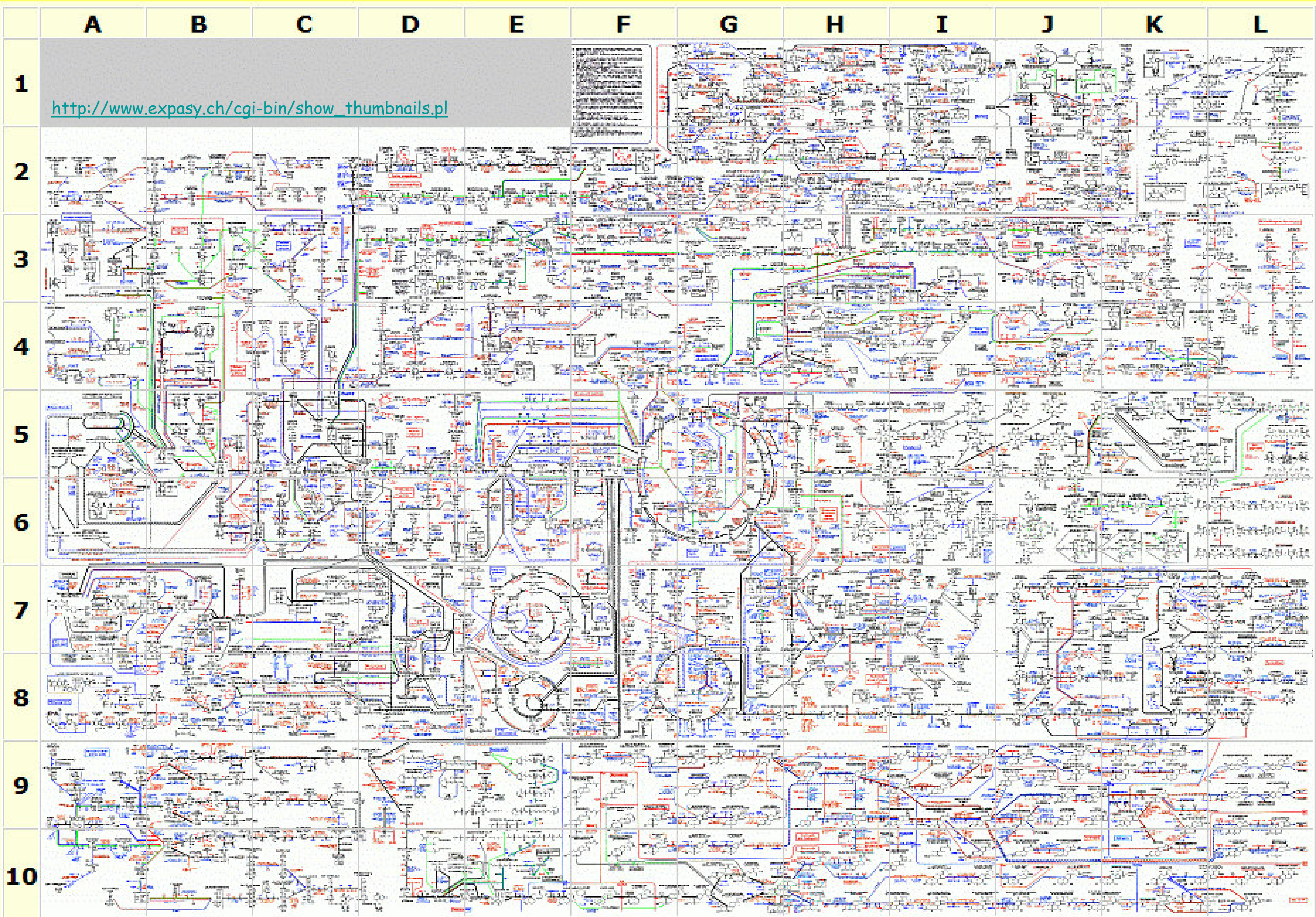
Stochastic Collectives

- "Collective":
  - A large set of interacting finite state automata:
    - Not quite language automata ("large set")
    - Not quite cellular automata ("interacting" but not on a grid)
    - Not quite process algebra ("finite state" and "collective")
    - Not quite calculus (rate of change of "automata"??)
    - Cf. "multi-agent systems" and "swarm intelligence"
- "Stochastic":
  - Interactions have *rates*
- Very much like biochemistry
  - Which is a large set of stochastically interacting molecules/proteins
  - Are proteins finite state and subject to automata-like transitions?
    - Let's say they are, at least because:
    - Much of the knowledge being accumated in Systems Biology is described as state transition diagrams [Kitano].

State Transitions



Even More State Transitions



Reverse Engineering Nature

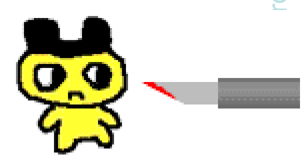
- That's what Systems Biology is up against
  - Exemplified by a technological analogy:

How often do I have to exercise my Tamagotchi?
Every Tamagotchi is different. However we do recommend exercising at least three times a day

- Tamagotchi: a technological organism
  - Has inputs (buttons) and outputs (screen/sound)
  - It has state: happy or needy (or hungry, sick, dead…)
  - Has to be petted at a certain rate (or gets needy)
  - Each one has a slightly different behavior

- Reverse Engineering Tamagotchi
  - Running experiments that elucidate their behavior
  - Building models that explain the experiments

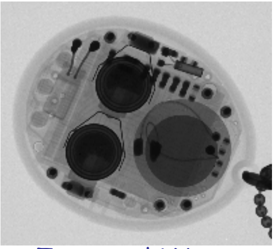
- Applications
  - Engineering: Can we build our own Tamagotchi? (Sadly, no longer made.)
  - Maintenance: Can we fix a broken Tamagotchi?



Understanding T.Nipponensis

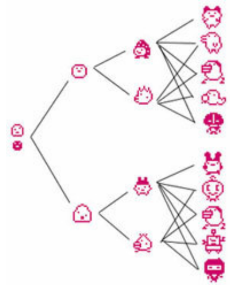
- Tamagotchi Nipponensis: a stochastic interactive automata
  - 40 million sold worldwide; discontinued in 1998
  - Still found "in the wild" in Akihabara

- Traditional scientific investigations fail
  - Design-driven understanding fails
    - We cannot read the manual (Japanese)
    - What does a Tamagotchi "compute"? What is its "purpose"?
    - Why does it have 3 buttons?
  - Mechanistic understanding fails
    - Few moving parts. Removing components mostly ineffective or "lethal"
    - The "tamagotchi folding problem" (sequence of manufacturing steps) is too hard and gives little insight on function
  - Behavioral understanding fails
    - Subjecting to extreme conditions reveals little and may void warranty
    - Does not answer consistently to individual stimuli, nor to sequences of stimuli
    - There are stochastic variations between individuals
  - Ecological understanding fails
    - Difficult to observe in its native environment (kids' hands)
    - Mass produced in little-understood automated factories
    - It evolved by competing with other products in the baffling Japanese market
  - Mathematical understanding fails
    - What differential equations does it obey? (Uh?)



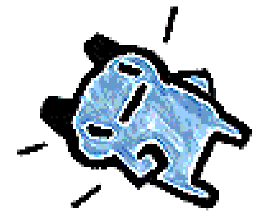
Tamagotchi X-ray

Tamagotchi Surgery
http://necrobones.com/tamasurg/



A New Approach

- "Systems Technology" of T. Nipponensis
  - High-throughput experiments (get all the information you possibly can)
    - Decode the entire software and hardware
    - Take sequences of tamagotchi screen dumps under different conditions
    - Put 300 in a basket and shake them; make statistics of final state



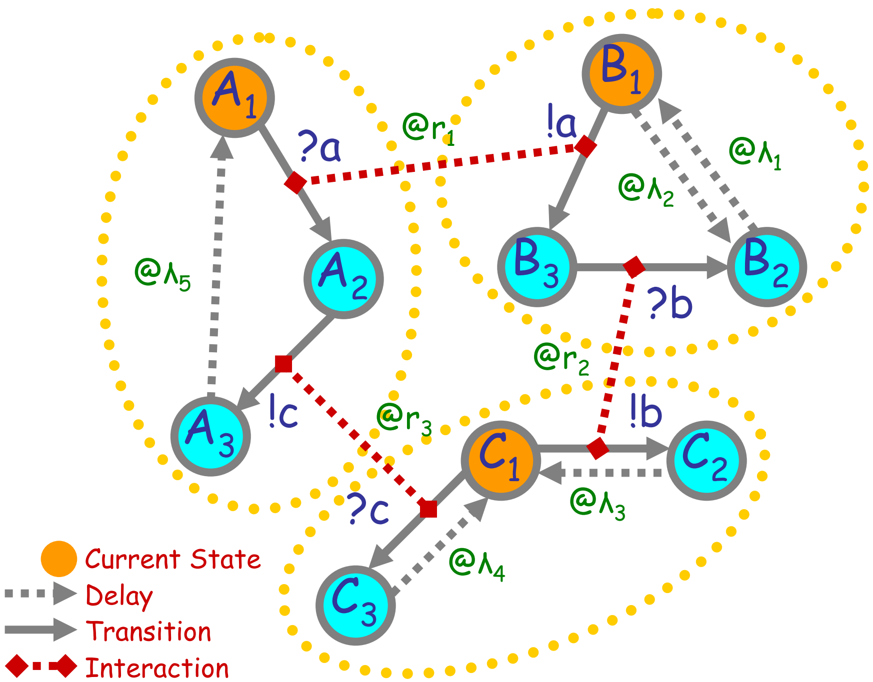
  - Modeling (organize all the information you got)
    - Ignore the "folding" (manufacturing) problem
    - Ignore materials (it's just something with buttons, display, and a *program*.)
    - Abstract until you find a conceptual model (ah-ha: it's a stochastic automata).

- Do we understand what stochastic automata collectives can do?

Communicating Tamagotchi

Automata Collectives

Interacting Automata



new a@r_1
new b@r_2
new c@r_3

Communication channels

A_1 = ?a; A_2
A_2 = !c; A_3
A_3 = @λ_5; A_1

B_1 = @λ_2; B_2 + !a; B_3
B_2 = @λ_1; B_1
B_3 = ?b; B_2

C_1 = !b; C_2 + ?c; C_3
C_2 = @λ_3; C_1
C_3 = @λ_4; C_2

Automata

A_1 | B_1 | C_1

The system and initial state

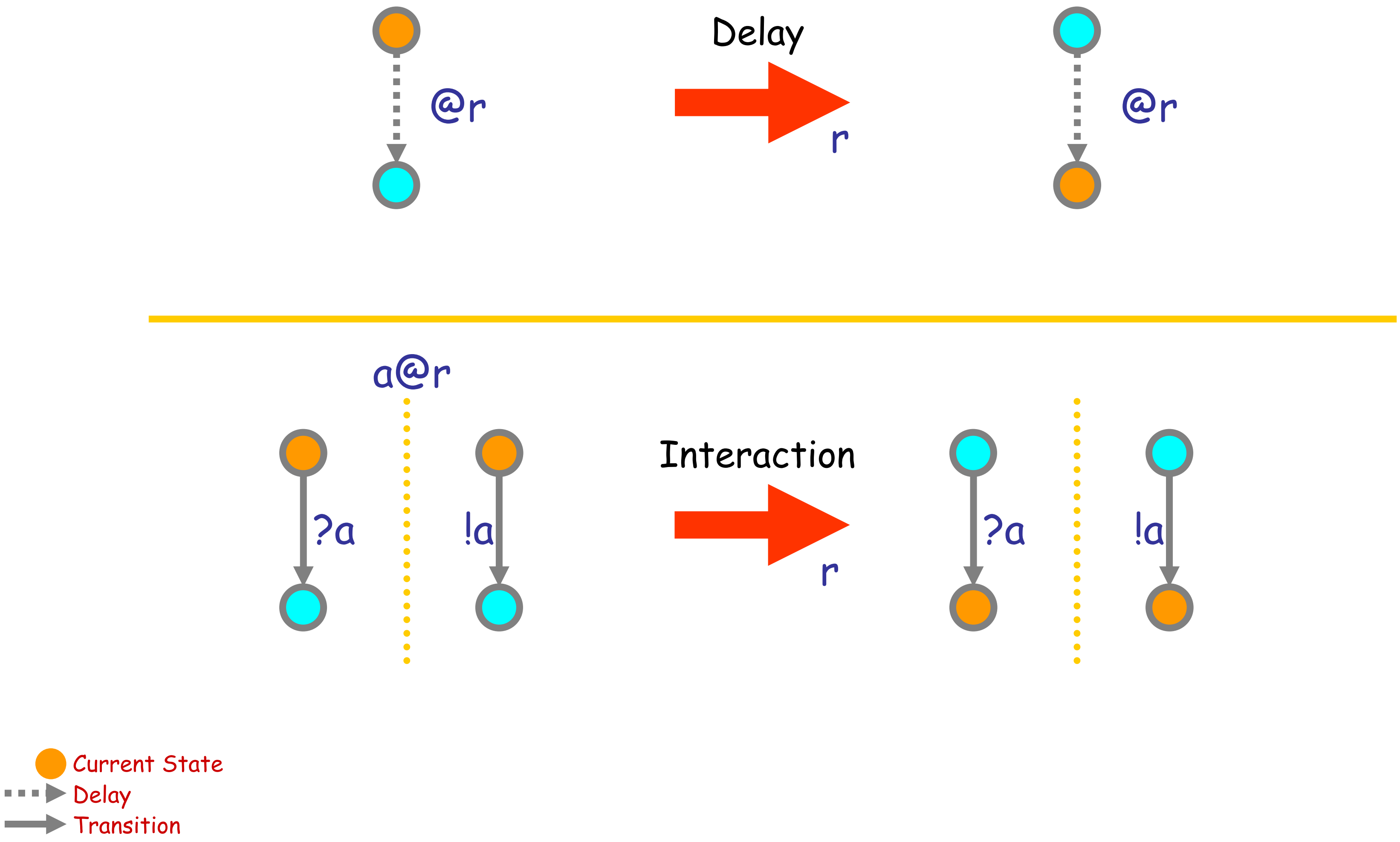
Communicating automata: a graphical FSA-like notation for “finite state restriction-free π-calculus processes”. *Interacting* automata do not even exchange values on communication.

The stochastic version has *rates* on communications, and delays.

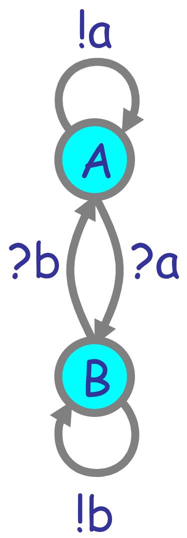
“Finite state” means: no composition or restriction inside recursion.

Analyzable by standard Markovian techniques, by first computing the “product automata” to obtain the underlying finite Markov transition system. [Buchholz]

Interacting Automata Transition Rules



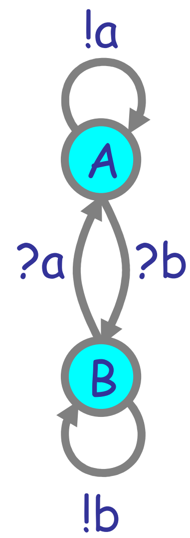
Groupies and Celebrities



Celebrity

(does not want to be like somebody else)

```
directive sample 0.1 1000
directive plot A(); B()

new a@1.0:chan()
new b@1.0:chan()

let A() = do !a; A() or ?a; B()
and B() = do !b; B() or ?b; A()

run 100 of (A() | B())
```

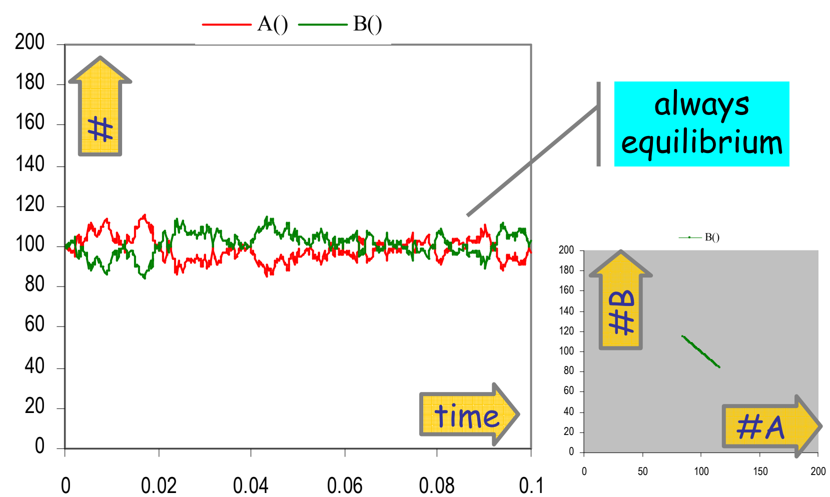


Groupie

(wants to be like somebody different)

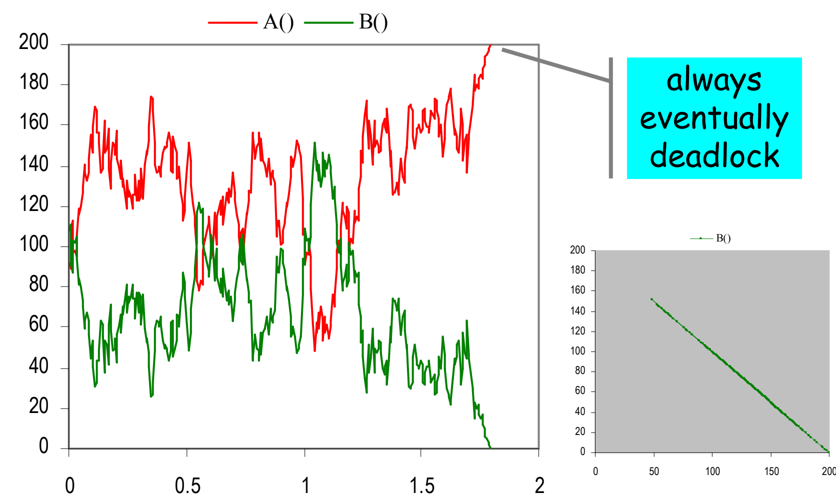
```
directive sample 5.0 1000
directive plot A(); B()

new a@1.0:chan()
new b@1.0:chan()

let A() = do !a; A() or ?b; B()
and B() = do !b; B() or ?a; A()

run 100 of (A() | B())
```

A stochastic collective of celebrities:



Stable because as soon as a A finds itself in the majority, it is more likely to find somebody in the same state, and hence change, so the majority is weakened.

A stochastic collective of groupies:

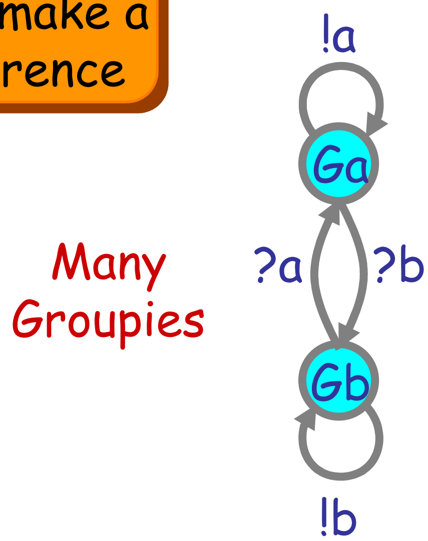


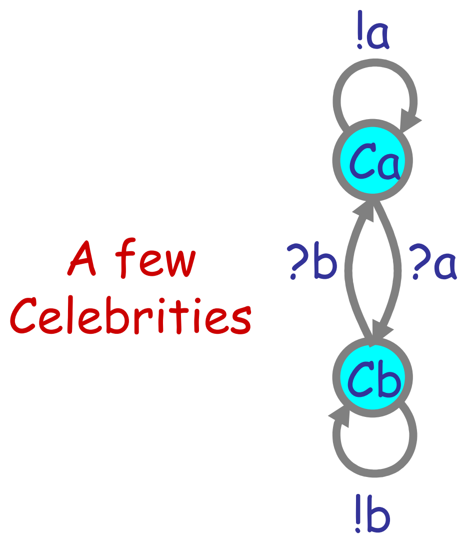
Unstable because within an A majority, an A has difficulty finding a B to emulate, but the few B's have plenty of A's to emulate, so the majority may switch to B. Leads to deadlock when everybody is in the same state and there is nobody different to emulate.

Both Together

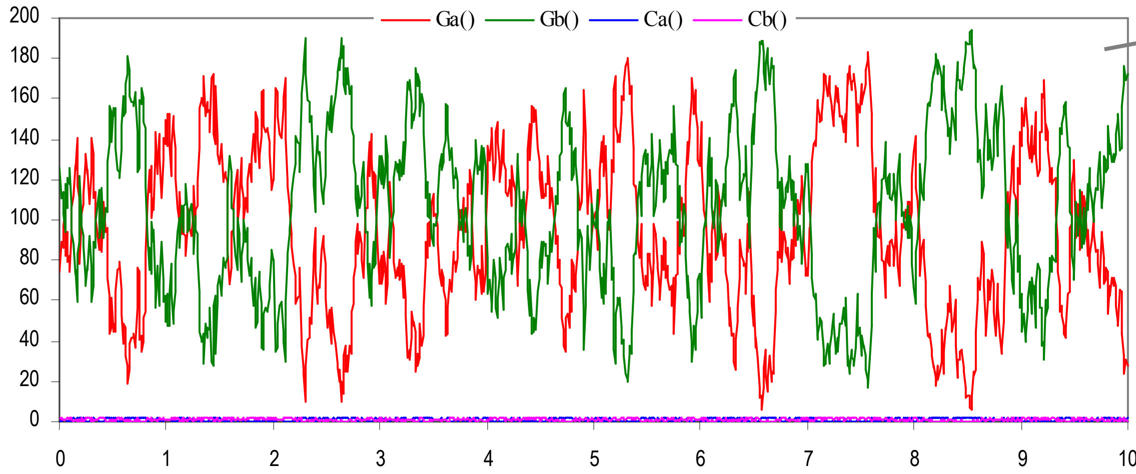
A tiny bit of "noise" can make a huge difference

A way to break the deadlocks: Groupies with just a few Celebrities

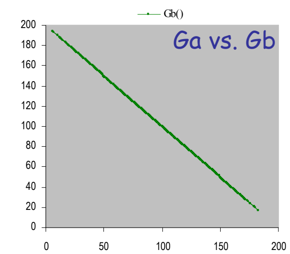





```
directive sample 10.0 1000
directive plot Ga(); Gb(); Ca(); Cb()

new a@1.0:chan()
new b@1.0:chan()

let Ca() = do !a; Ca() or ?a; Cb()
and Cb() = do !b; Cb() or ?b; Ca()

let Ga() = do !a; Ga() or ?b; Gb()
and Gb() = do !b; Gb() or ?a; Ga()

run    1 of (Ca() | Cb())
run 100 of (Ga() | Gb())
```

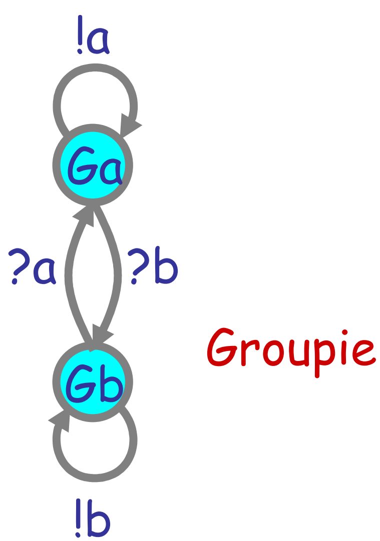


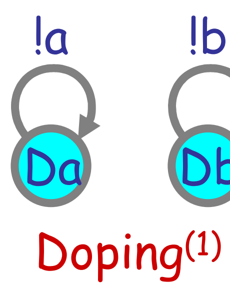
never deadlock



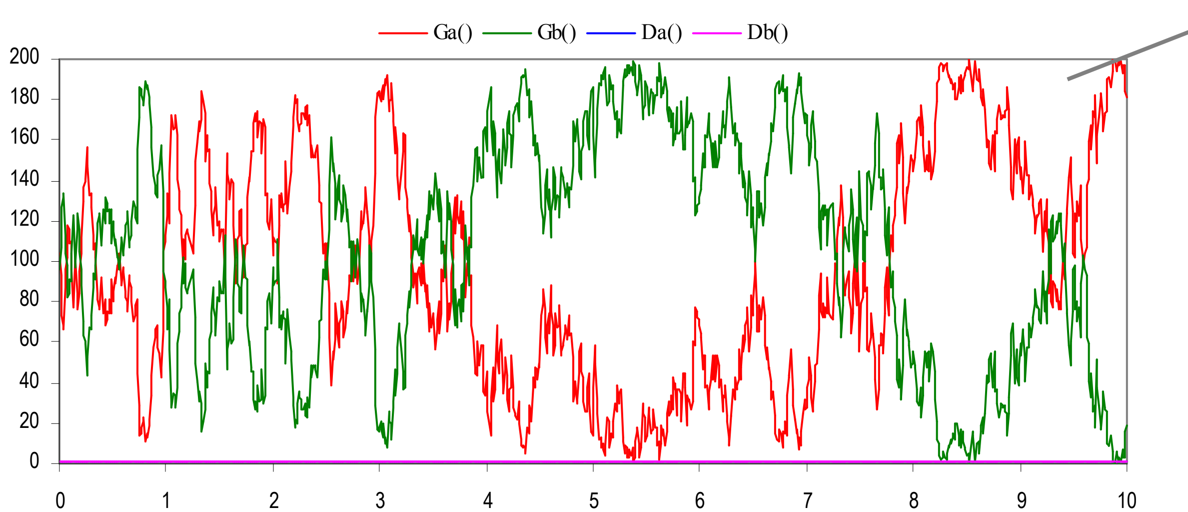
Doped Groupies

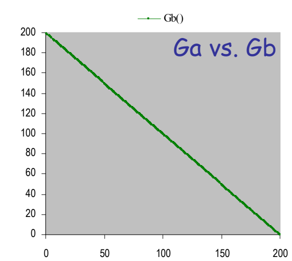
A similar way to break the deadlocks: destabilize the groupies by a small perturbation.






```
directive sample 10.0 1000
directive plot Ga(); Gb(); Da(); Db()

new a@1.0:chan()
new b@1.0:chan()

let Ga() = do !a; Ga() or ?b; Gb()
and Gb() = do !b; Gb() or ?a; Ga()

let Da() = !a; Da()
and Db() = !b; Db()

run   1 of (Da() | Db())
run 100 of (Ga() | Gb())
```

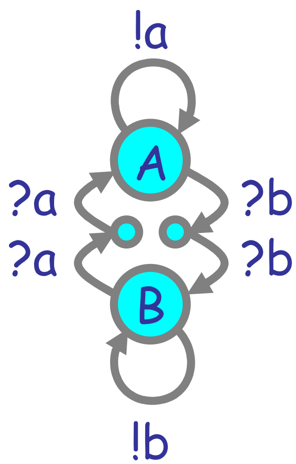


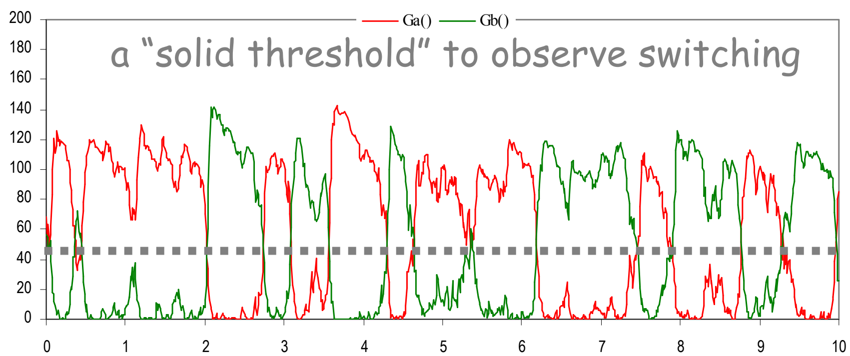


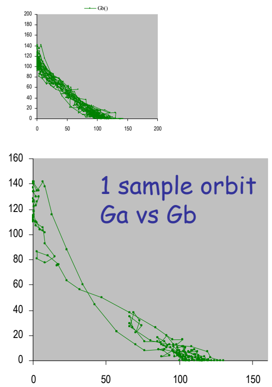
(1)A technical term in microelectronics

Hysteric Groupies

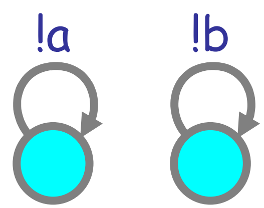
We can get more regular behavior from groupies if they "need more convincing", or "hysteresis" (history-dependence), to switch states.



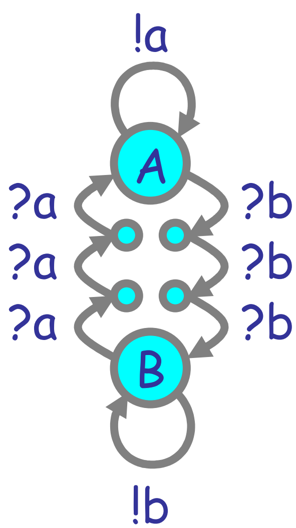


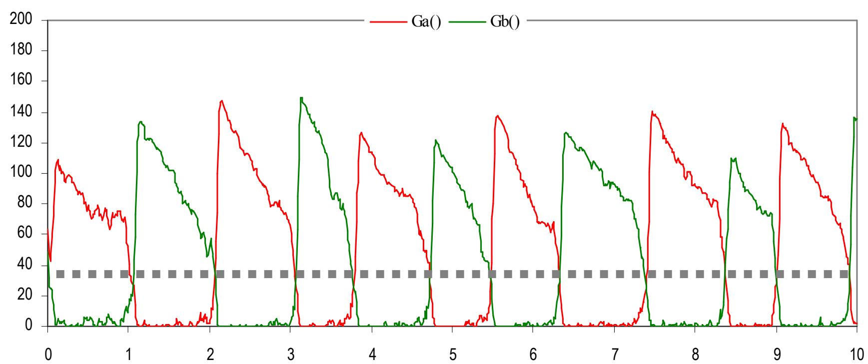


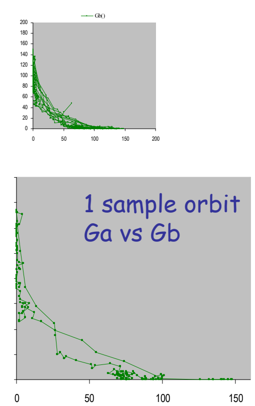

```
directive sample 10.0 1000
directive plot Ga(); Gb()

new a@1.0:chan()
new b@1.0:chan()

let Ga() = do !a; Ga() or ?b; ?b; Gb()
and Gb() = do !b; Gb() or ?a; ?a; Ga()

let Da() = !a; Da()
and Db() = !b; Db()

run 100 of (Ga() | Gb())
run   1 of (Da() | Db())
```



(Still with doping)

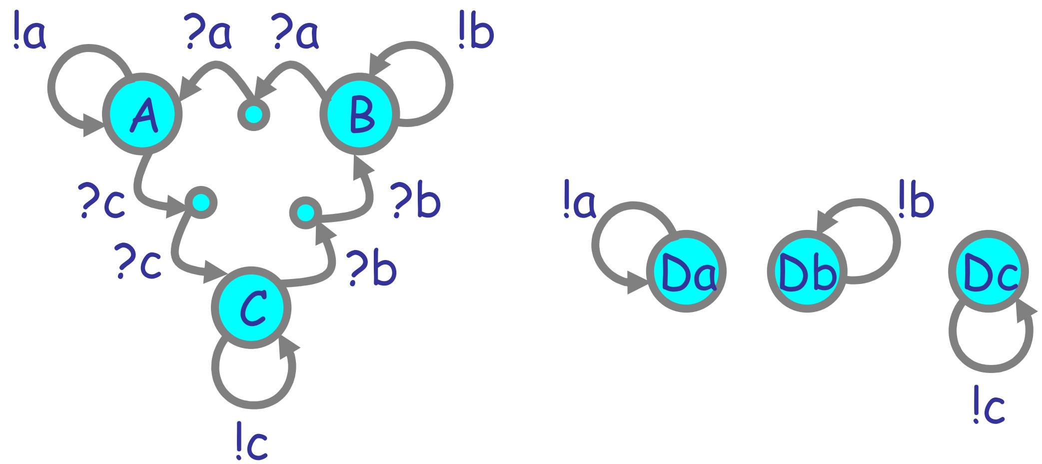




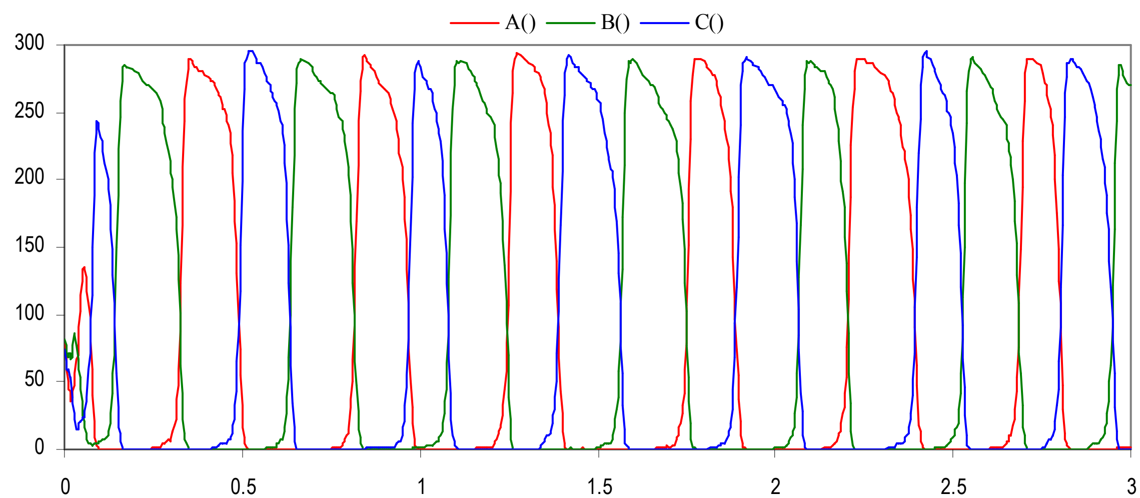


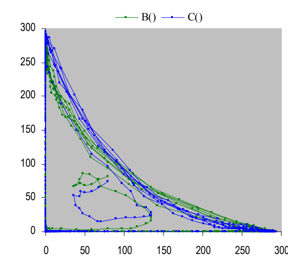

```
directive sample 10.0 1000
directive plot Ga(); Gb()

new a@1.0:chan()
new b@1.0:chan()

let Ga() = do !a; Ga() or ?b; ?b; ?b; Gb()
and Gb() = do !b; Gb() or ?a; ?a; ?a; Ga()

let Da() = !a; Da()
and Db() = !b; Db()

run 100 of (Ga() | Gb())
run   1 of (Da() | Db())
```

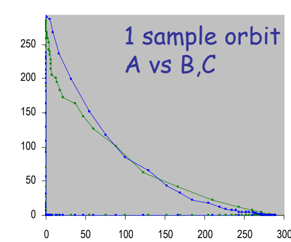
Hysteric 3-Way Groupies



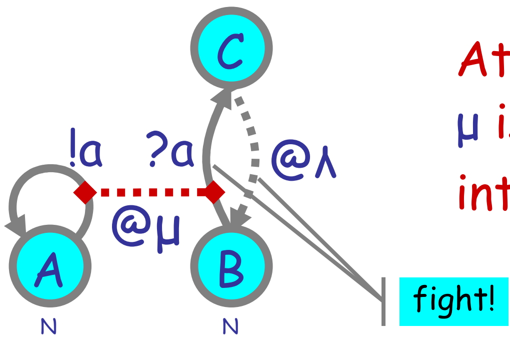

```
directive sample 3.0 1000
directive plot A(); B(); C()

new a@1.0:chan()
new b@1.0:chan()
new c@1.0:chan()

let A() = do !a; A() or ?c; ?c; C()
and B() = do !b; B() or ?a; ?a; A()
and C() = do !c; C() or ?b; ?b; B()

let Da() = !a; Da()
and Db() = !b; Db()
and Dc() = !c; Dc()

run 100 of (A() | B() | C())
run 1 of (Da() | Db() | Dc())
```







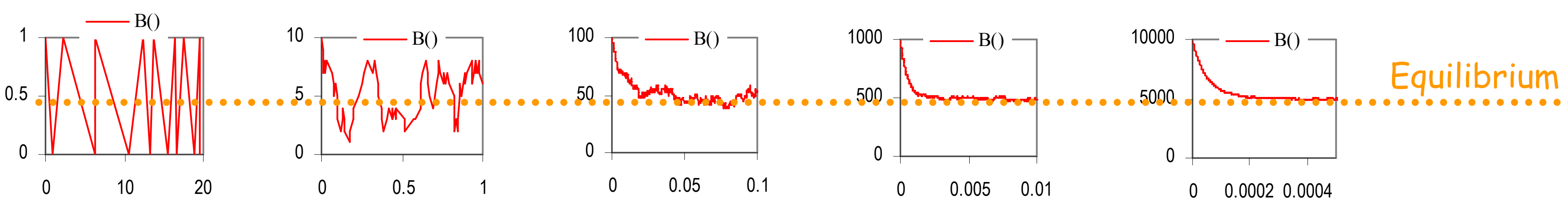
The Strength of Populations



At size 2N, on a shared channel,
μ is N times stronger than λ:
interaction easily wins over delay.

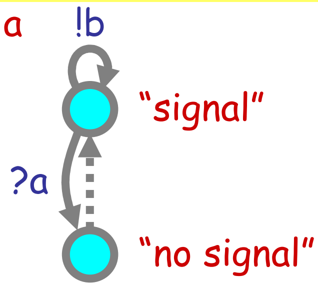
```
directive sample 0.01 1000
directive plot B()

val lam = 1000.0
val mu = 1.0

new a@mu:chan
let A() = !a; A()
and B() = ?a; C()
and C() = delay@lam; B()

run 1000 of (A() | B())
```

N=1	N=10	N=100	N=1000	N=10000
λ=1	λ=10	λ=100	λ=1000	λ=10000
μ=1	μ=1	μ=1	μ=1	μ=1

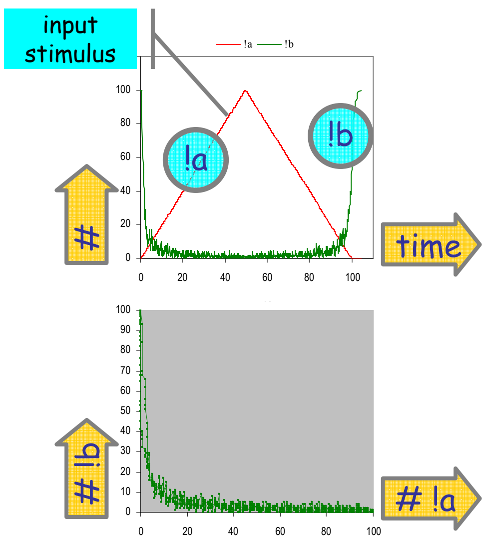


Boolean Inverter Collectives

b = not a

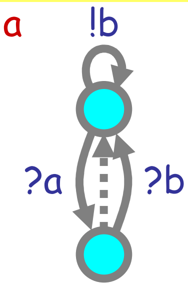


in presence of a, b goes low
in absence of a, b goes high

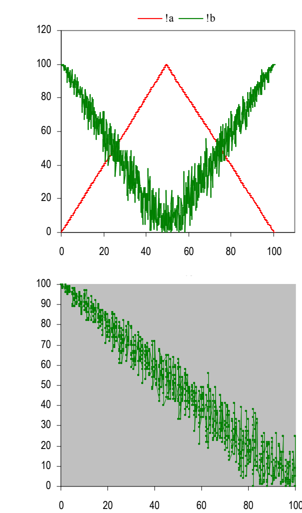



```
directive sample 110.0 1000
directive plot !a; !b

new a@1.0:chan new b@1.0:chan

let Inv_hi(a:chan, b:chan) =
  do !b; Inv_hi(a,b)
  or ?a; Inv_lo(a,b)
and Inv_lo(a:chan, b:chan) =
  delay@1.0; Inv_hi(a,b)

run 100 of Inv_hi(a,b)

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))
let S1(a:chan, tock:chan) =  do !a; S1(a,tock) or ?tock; ()
let SN(n:int, t:float, a:chan, tick:chan, tock:chan) =
  if n=0 then clock(t, tock) else ?tick; (S1(a,tock) | SN(n-1,t,a,tick,tock))
let raisingfalling(a:chan, n:int, t:float) =
  (new tick:chan new tock:chan
   run (clock(t,tick) | SN(n,t,a,tick,tock)))

run raisingfalling(a,100,0.5)
```

b = not a

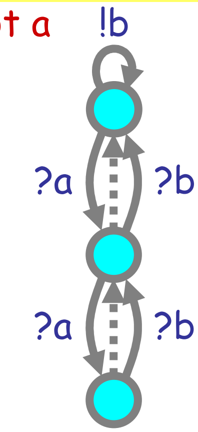


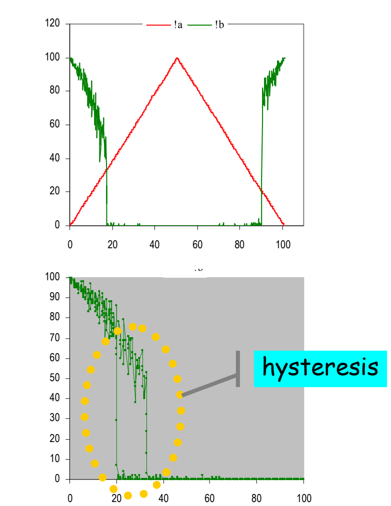
the high b state reinforces itself (as a population)



```
directive sample 110.0 1000
directive plot !a; !b

new a@1.0:chan new b@1.0:chan

let Inv_hi(a:chan, b:chan) =
  do !b; Inv_hi(a,b)
  or ?a; Inv_lo(a,b)
and Inv_lo(a:chan, b:chan) =
  do ?b; Inv_hi(a,b)
  or delay@1.0; Inv_hi(a,b)

run 100 of Inv_hi(a,b)

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))
let S1(a:chan, tock:chan) =  do !a; S1(a,tock) or ?tock; ()
let SN(n:int, t:float, a:chan, tick:chan, tock:chan) =
  if n=0 then clock(t, tock) else ?tick; (S1(a,tock) | SN(n-1,t,a,tick,tock))
let raisingfalling(a:chan, n:int, t:float) =
  (new tick:chan new tock:chan
   run (clock(t,tick) | SN(n,t,a,tick,tock)))

run raisingfalling(a,100,0.5)
```

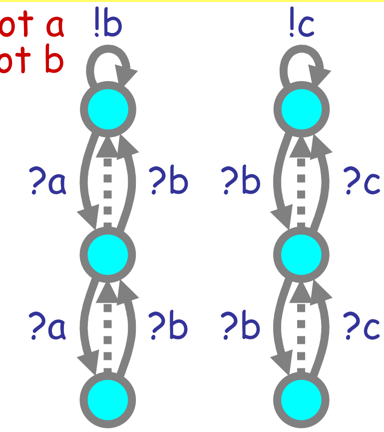
b = not a

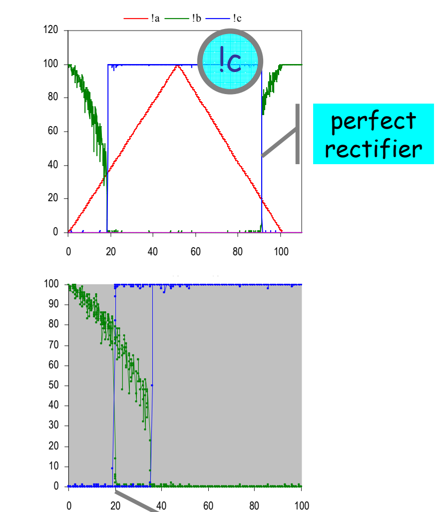





```
directive sample 110.0 1000
directive plot !a; !b

new a@1.0:chan new b@1.0:chan

let Inv2_hi(a:chan, b:chan) =
  do !b; Inv2_hi(a,b) or ?a; Inv2_mi(a,b)
and Inv2_mi(a:chan, b:chan) =
  do ?b; Inv2_hi(a,b) or delay@1.0; Inv2_hi(a,b)
  or ?a; Inv2_lo(a,b)
and Inv2_lo(a:chan, b:chan) =
  do ?b; Inv2_mi(a,b) or delay@1.0; Inv2_mi(a,b)

run 100 of Inv2_hi(a,b)

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))
let S1(a:chan, tock:chan) =  do !a; S1(a,tock) or ?tock; ()
let SN(n:int, t:float, a:chan, tick:chan, tock:chan) =
  if n=0 then clock(t, tock) else ?tick; (S1(a,tock) | SN(n-1,t,a,tick,tock))
let raisingfalling(a:chan, n:int, t:float) =
  (new tick:chan new tock:chan
   run (clock(t,tick) | SN(n,t,a,tick,tock)))

run raisingfalling(a,100,0.5)
```

b = not a
c = not b



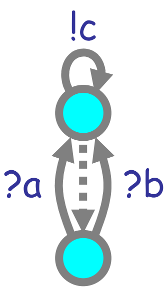


zero-point noise resistant

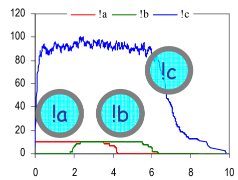
```
directive sample 110.0 1000
directive plot !a; !b; !c; !d

new a@1.0:chan new b@1.0:chan new c@1.0:chan

let Inv2_hi(a:chan, b:chan) =
  do !b; Inv2_hi(a,b) or ?a; Inv2_mi(a,b)
and Inv2_mi(a:chan, b:chan) =
  do ?b; Inv2_hi(a,b) or delay@1.0; Inv2_hi(a,b)
  or ?a; Inv2_lo(a,b)
and Inv2_lo(a:chan, b:chan) =
  do ?b; Inv2_mi(a,b) or delay@1.0; Inv2_mi(a,b)

run 100 of (Inv2_hi(a,b) | Inv2_lo(b,c))

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))
let S1(a:chan, tock:chan) =  do !a; S1(a,tock) or ?tock; ()
let SN(n:int, t:float, a:chan, tick:chan, tock:chan) =
  if n=0 then clock(t, tock) else ?tick; (S1(a,tock) | SN(n-1,t,a,tick,tock))
let raisingfalling(a:chan, n:int, t:float) =
  (new tick:chan new tock:chan
   run (clock(t,tick) | SN(n,t,a,tick,tock)))

run raisingfalling(a,100,0.5)
```

Boolean Gate Collectives

c = a or b

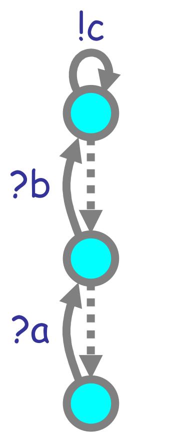


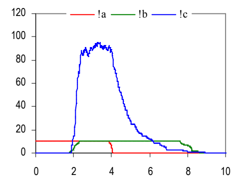
Inputs:
10 !a for 4t
2t; 10 !b for 4t




```
directive sample 10.0 1000
directive plot !a; !b; !c

new a@1.0:chan new b@1.0:chan new c@1.0:chan
val del = 1.0

let Or_hi(a:chan, b:chan, c:chan) =
  do !c; Or_hi(a,b,c) or delay@del; Or_lo(a,b,c)
and Or_lo(a:chan, b:chan, c:chan) =
  do ?a; Or_hi(a,b,c) or ?b; Or_hi(a,b,c)

run 100 of Or_lo(a,b,c)

let clock(t:float, tick:chan) =      (* sends a tick every t time *)
  (val ti = t/200.0 val d = 1.0/ti
   let step(n:int) =
     if n<=0 then !tick; clock(t, tick) else delay@d; step(n-1)
   run step(200))

let S_a(tick:chan) = do !a; S_a(tick) or ?tick; ()
let S_b(tick:chan) = ?tick; S_b1(tick)
and S_b1(tick:chan) = do !b; S_b1(tick) or ?tick; S_b2(tick)
and S_b2(tick:chan) = do !b; S_b2(tick) or ?tick; ()

run 10 of (new tick:chan run (clock(4.0,tick) | S_a(tick)))
run 10 of (new tick:chan run (clock(2.0,tick) | S_b(tick)))
```

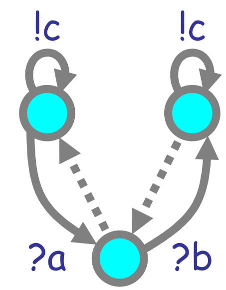
c = a and b

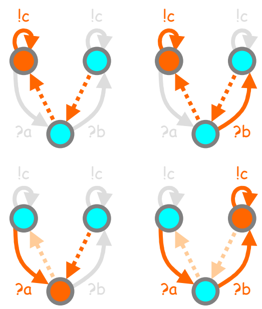


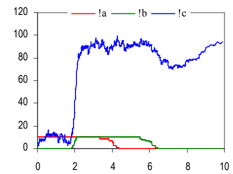



```
directive sample 10.0 1000
directive plot !a; !b; !c

new a@1.0:chan new b@1.0:chan new c@1.0:chan
val del = 1.0

let And_hi(a:chan, b:chan, c:chan) =
  do !c; And_hi(a,b,c) or delay@del; And_lo_a(a,b,c)
and And_lo_a(a:chan, b:chan, c:chan) =
  do ?a; And_hi(a,b,c) or delay@del; And_lo_b(a,b,c)
and And_lo_b(a:chan, b:chan, c:chan) =
  ?b; And_lo_a(a,b,c)

run 100 of And_lo_b(a,b,c)

let clock(t:float, tick:chan) =      (* sends a tick every t time *)
  (val ti = t/200.0 val d = 1.0/ti
   let step(n:int) =
     if n<=0 then !tick; clock(t, tick) else delay@d; step(n-1)
   run step(200))

let S_a(tick:chan) = do !a; S_a(tick) or ?tick; ()
let S_b(tick:chan) = ?tick; S_b1(tick)
and S_b1(tick:chan) = do !b; S_b1(tick) or ?tick; S_b2(tick)
and S_b2(tick:chan) = do !b; S_b2(tick) or ?tick; S_b3(tick)
and S_b3(tick:chan) = do !b; S_b3(tick) or ?tick; ()

run 10 of (new tick:chan run (clock(4.0,tick) | S_a(tick)))
run 10 of (new tick:chan run (clock(2.0,tick) | S_b(tick)))
```

c = a imply b

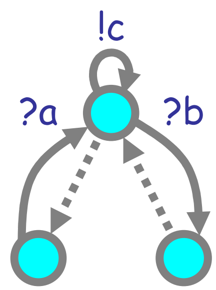


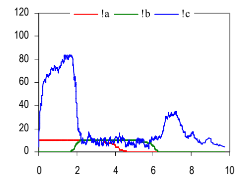





```
directive sample 10.0 1000
directive plot !a; !b; !c

new a@1.0:chan new b@1.0:chan new c@1.0:chan
val del = 1.0

let Imply_hi_a(a:chan, b:chan, c:chan) =
  do !c; Imply_hi_a(a,b,c) or ?a; Imply_lo(a,b,c)
and Imply_hi_b(a:chan, b:chan, c:chan) =
  do !c; Imply_hi_b(a,b,c) or delay@del; Imply_lo(a,b,c)
and Imply_lo(a:chan, b:chan, c:chan) =
  do ?b; Imply_hi_b(a,b,c) or delay@del; Imply_hi_a(a,b,c)

run 100 of Imply_lo(a,b,c)

let clock(t:float, tick:chan) =      (* sends a tick every t time *)
  (val ti = t/200.0 val d = 1.0/ti
   let step(n:int) =
     if n<=0 then !tick; clock(t, tick) else delay@d; step(n-1)
   run step(200))

let S_a(tick:chan) = do !a; S_a(tick) or ?tick; ()
let S_b(tick:chan) = ?tick; S_b1(tick)
and S_b1(tick:chan) = do !b; S_b1(tick) or ?tick; S_b2(tick)
and S_b2(tick:chan) = do !b; S_b2(tick) or ?tick; ()

run 10 of (new tick:chan run (clock(4.0,tick) | S_a(tick)))
run 10 of (new tick:chan run (clock(2.0,tick) | S_b(tick)))
```

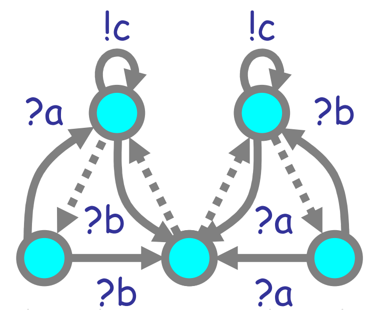
c = a unless b

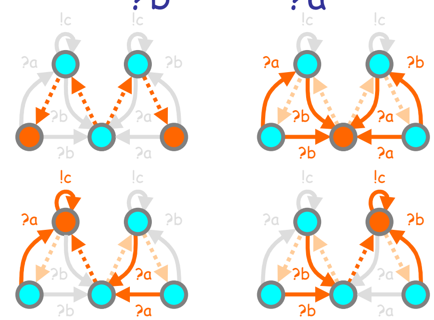


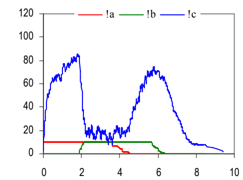



```
directive sample 10.0 1000
directive plot !a; !b; !c

new a@1.0:chan new b@1.0:chan new c@1.0:chan
val del = 1.0

let OOlO_hi(a:chan, b:chan, c:chan) =
  do !c; OOlO_hi(a,b,c) or delay@del; OOlO_lo_a(a,b,c) or ?b;
OOlO_lo_b(a,b,c)
and OOlO_lo_a(a:chan, b:chan, c:chan) =
  ?a; OOlO_hi(a,b,c)
and OOlO_lo_b(a:chan, b:chan, c:chan) =
  delay@del; OOlO_hi(a,b,c)

run 50 of (OOlO_lo_a(a,b,c) | OOlO_lo_b(a,b,c))

let clock(t:float, tick:chan) =      (* sends a tick every t time *)
  (val ti = t/200.0 val d = 1.0/ti
   let step(n:int) =
     if n<=0 then !tick; clock(t, tick) else delay@d; step(n-1)
   run step(200))

let S_a(tick:chan) = do !a; S_a(tick) or ?tick; ()
let S_b(tick:chan) = ?tick; S_b1(tick)
and S_b1(tick:chan) = do !b; S_b1(tick) or ?tick; S_b2(tick)
and S_b2(tick:chan) = do !b; S_b2(tick) or ?tick; ()

run 10 of (new tick:chan run (clock(4.0,tick) | S_a(tick)))
run 10 of (new tick:chan run (clock(2.0,tick) | S_b(tick)))
```

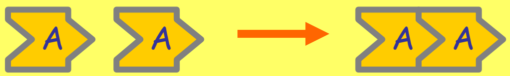
c = a xor b







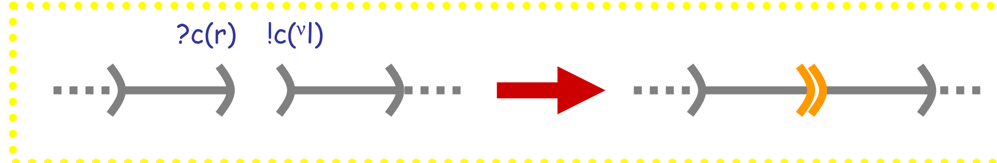

```
directive sample 10.0 1000
directive plot !a; !b; !c

new a@1.0:chan new b@1.0:chan new c@1.0:chan

let Xor_hi_a(a:chan, b:chan, c:chan) =
  do !c; Xor_hi_a(a,b,c) or ?b; Xor_lo_ab(a,b,c) or delay@1.0; Xor_lo_a(a,b,c)
and Xor_hi_b(a:chan, b:chan, c:chan) =
  do !c; Xor_hi_b(a,b,c) or ?a; Xor_lo_ab(a,b,c) or delay@1.0; Xor_lo_b(a,b,c)
and Xor_lo_a(a:chan, b:chan, c:chan) =
  do ?a; Xor_hi_a(a,b,c) or ?b; Xor_lo_ab(a,b,c)
and Xor_lo_b(a:chan, b:chan, c:chan) =
  do ?b; Xor_hi_b(a,b,c) or ?a; Xor_lo_ab(a,b,c)
and Xor_lo_ab(a:chan, b:chan, c:chan) =
  do delay@1.0; Xor_hi_a(a,b,c) or delay@1.0; Xor_hi_b(a,b,c)

run 50 of (Xor_lo_a(a,b,c) | Xor_lo_b(a,b,c))

let clock(t:float, tick:chan) =      (* sends a tick every t time *)
  (val ti = t/200.0 val d = 1.0/ti
   let step(n:int) =
     if n<=0 then !tick; clock(t, tick) else delay@d; step(n-1)
   run step(200))

let S_a(tick:chan) = do !a; S_a(tick) or ?tick; ()
let S_b(tick:chan) = ?tick; S_b1(tick)
and S_b1(tick:chan) = do !b; S_b1(tick) or ?tick; S_b2(tick)
and S_b2(tick:chan) = do !b; S_b2(tick) or ?tick; ()

run 10 of (new tick:chan run (clock(4.0,tick) | S_a(tick)))
run 10 of (new tick:chan run (clock(2.0,tick) | S_b(tick)))
```



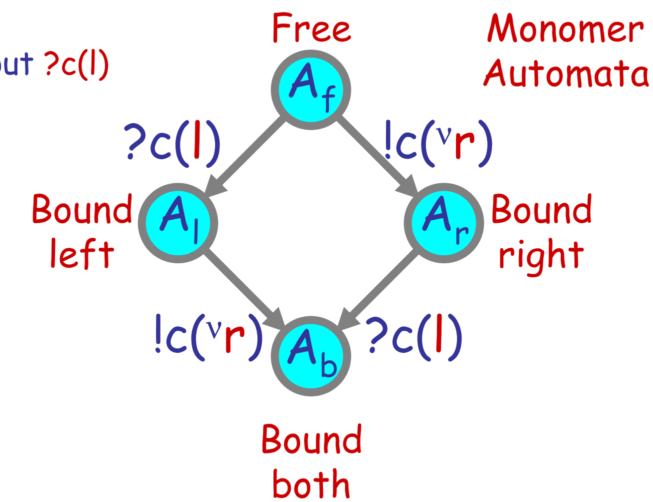
Bidirectional

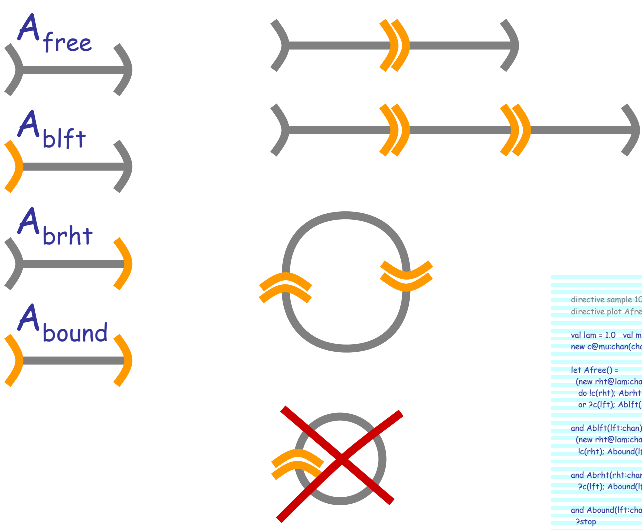
Polymerization

```
new c@μ  new stop@1.0

A_free =
    (new rht@λ; !c(rht); A_brht(rht))
    + ?c(lft); A_blft(lft)
A_blft(lft) =
    (new rht@λ; !c(rht); A_bound(lft,rht))
A_brht(rht) =
    ?c(lft); A_bound(lft,rht)
A_bound(lft,rht) = ?stop
```

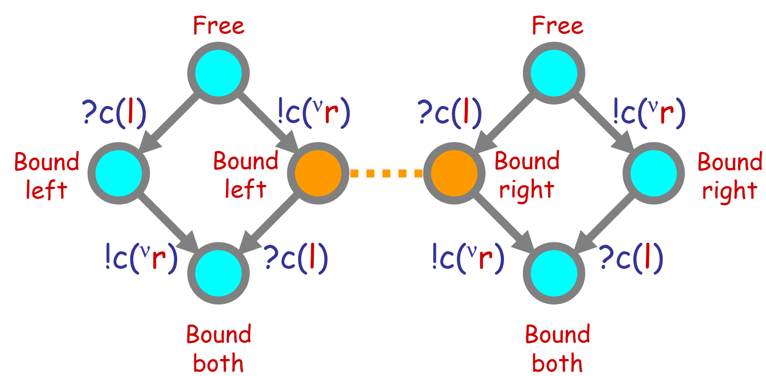


Communicating Automata
Bound output !c(νr) and input ?c(l) on automata transitions to model complexation






```
directive sample 10000.0
directive plot Afree(); Ablft(); Abrht(); Abound()

val lam = 1.0   val mu = 1.0
new c@mu:chan(chan)  new stop@1.0:chan

let Afree() =
  (new rht@lam:chan run
   do !c(rht); Abrht(rht)
   or ?c(lft); Ablft(lft))

and Ablft(lft:chan) =
  (new rht@lam:chan run
   !c(rht); Abound(lft,rht))

and Abrht(rht:chan) =
   ?c(lft); Abound(lft,rht)

and Abound(lft:chan, rht:chan) =
  ?stop

run (2 of Afree())
```

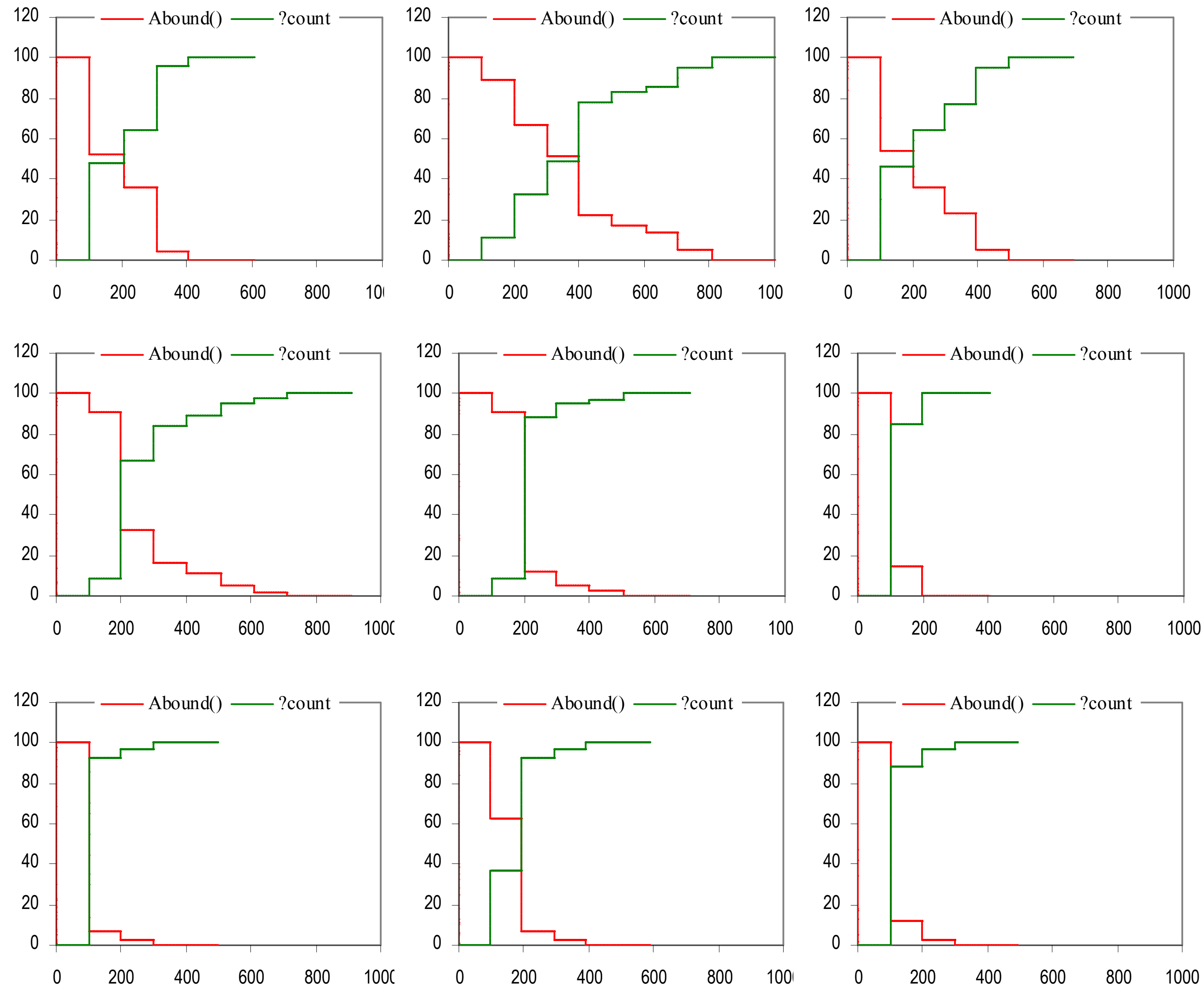


Bidirectional Polymerization

Circular Polymer Lengths

Scanning and counting the size of the circular polymers (by a cheap trick).

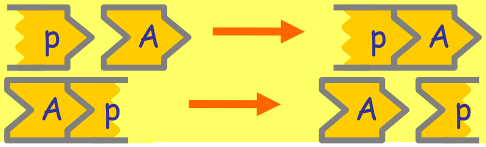
Polymer formation is complete within 10t; then a different polymer is scanned every 100t.




```
directive sample 1000.0
directive plot Abound(); ?count

type Link = chan(chan)
type Barb = chan

val lam = 1000.0 (* set high for better counting *)
val mu = 1.0
new c@mu:chan(Link)
new enter@lam:chan(Barb)
new count@lam:Barb

let Afree() =
  (new rht@lam:Link run
   do !c(rht); Abrht(rht)
   or ?c(lft); Ablft(lft))

and Ablft(lft:Link) =
  (new rht@lam:Link run
   !c(rht); Abound(lft,rht))

and Abrht(rht:Link) =
   ?c(lft); Abound(lft,rht)

and Abound(lft:Link, rht:Link) =
  do ?enter(barb); (?barb | !rht(barb))
  or ?lft(barb); (?barb | !rht(barb))
(* each Abound waits for a barb, exhibits it, and passes it to
   the right so we can plot number of Abound in a ring *)

let clock(t:float, tick:chan) =       (* sends a tick every t time *)
  (val ti = t/1000.0 val d = 1.0/ti
   let step(n:int) =
     if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(1000))

new tick:chan
let Scan() = ?tick; !enter(count); Scan()

run 100 of Afree()
run (clock(100.0, tick) | Scan())
```

$100 \times A_{free}$, initially.

The height of each rising step is the size of a separate circular polymer. (Unbiased sample of nine consecutive runs.)



Actin-like

Poly/Depolymerization

new c@μ

A_{free} =

  (new lft@λ; !c(lft); A_{blft}(lft)) +

  ?c(rht); A_{brht}(rht)

A_{blft}(lft) =

  !lft; A_{free} +

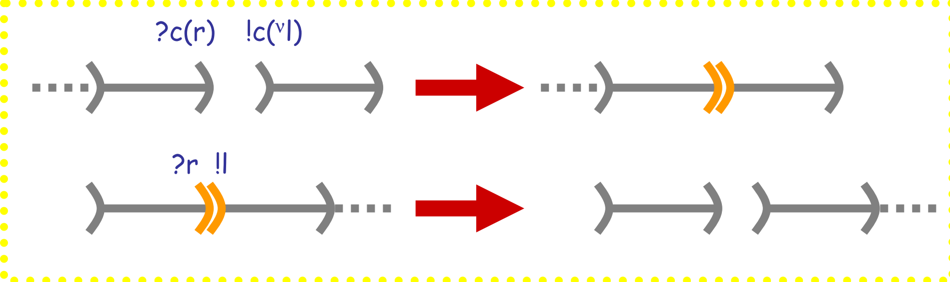
  ?c(rht); A_{bound}(lft,rht)

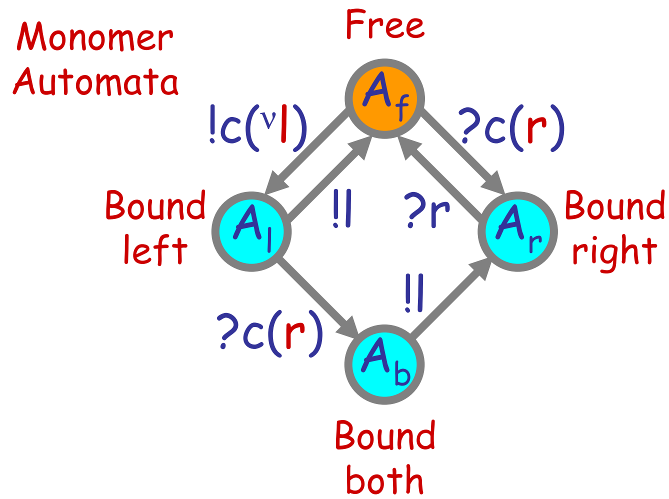
A_{brht}(rht) =

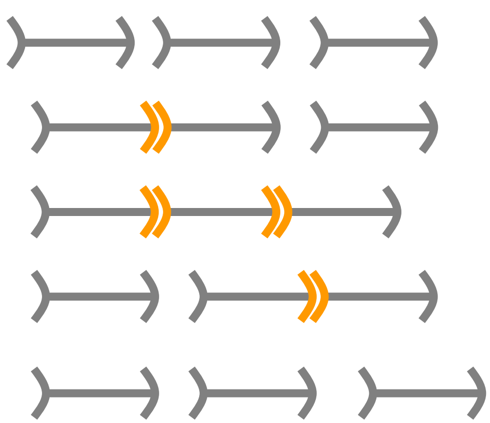
  ?rht; A_{free}

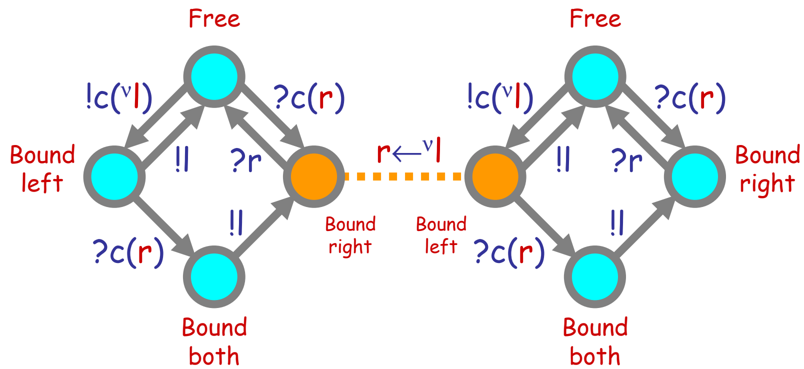
A_{bound}(lft,rht) =

  !lft; A_{brht}(rht)







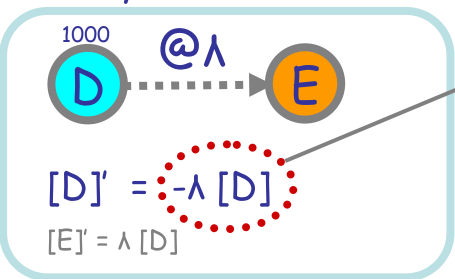


The Law of Mass Interaction

Law of Mass Interaction

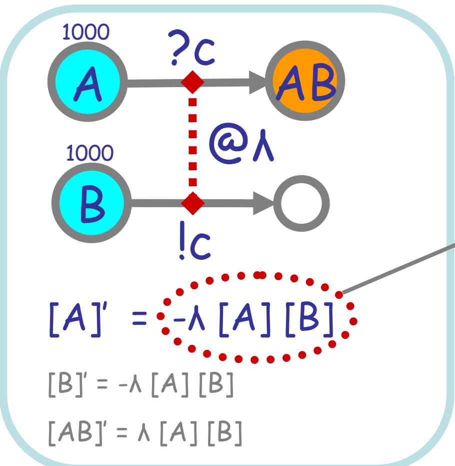
The speed of interaction† is proportional to the number of *possible interactions*.

Decay



Exponential Decay law
Rate of change proportional to number of possible decays.

Mass interaction

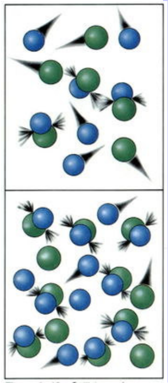


Interaction Law generalizes Decay Law

Mass Interaction law
Rate of change proportional to number of possible interactions

† speed of interaction (formally definable)
= number of interactions over time
not proportional to the number of interacting processes!

[P] is the number of processes P (this is informal; it is only meaningful for a set of processes offering a given action, but a set of such processes can be counted and plotted)



Chemical Law of Mass Action

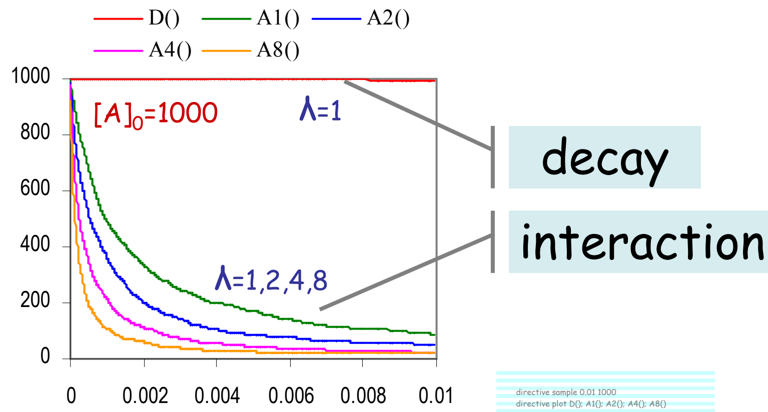
http://en.wikipedia.org/wiki/Chemical_kinetics

The **speed** of a chemical reaction is proportional to the **activity** of the reacting substances.

(Activity = concentration, for well-stirred aqueous medium)

(Concentration = number of moles per liter of solution)

(Mole = 6.022141×10^{23} particles)




```
directive sample 0.01 1000
directive plot D(); A1(); A2(); A4(); A8()

new c1@1.0: chan()   new c2@2.0: chan()
new c4@4.0: chan()   new c8@8.0: chan()

let D() = delay@1.0
let A1() = ?c1 and B1() = !c1
let A2() = ?c2 and B2() = !c2
let A4() = ?c4 and B4() = !c4
let A8() = ?c8 and B8() = !c8

run 1000 of (D() | A1() | B1() | A2()
| B2() | A4() | B4() | A8() | B8())
```

Activity and Speed

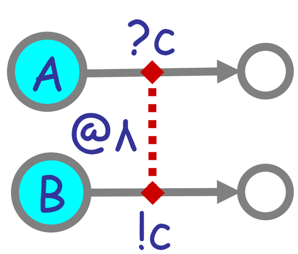
stochastic algebras disagree!

The speed of interaction is proportional to the number of possible interactions. =

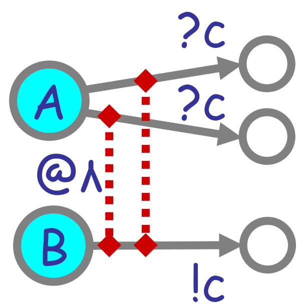
The *activity* (= "concentration") on a channel is the number of *possible interactions* on that channel.

The *speed of interaction* on a channel, is the activity multiplied by the base rate of the channel.

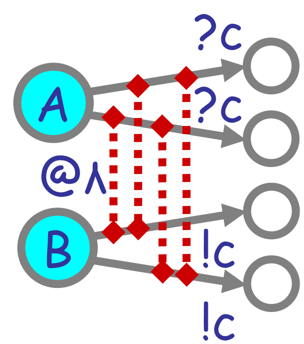
```
directive sample 0.01 10000
directive plot A1(); A2(); A3()

new c1@1.0:chan
new c2@1.0:chan
new c3@1.0:chan

let A1() = ?c1
and B1() = !c1

let A2() = do ?c2 or ?c2
and B2() = !c2

let A3() = do ?c3 or ?c3
and B3() = do !c3 or !c3

run 1000 of (A1() | B1()
| A2() | B2() | A3() | B3())
```



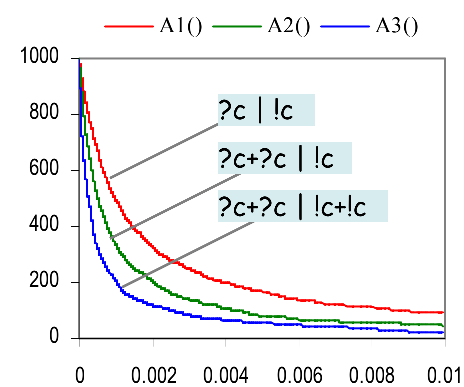
c activity: 1
speed: λ



c activity: 2
speed: 2λ



c activity: 4
speed: 4λ



The mass interaction law [Buchholz] [Priami-Regev-Shapiro-Silverman] is compatible with chemistry [Gillespie] and *incompatible* with any other stochastic algebra in the literature! (including [Priami]; see [Hermanns])

Other algebras assign rates to actions, not channels, with speed laws:

- $2\lambda*2\lambda = 4\lambda^2$
- max(2λ,2λ) = 2λ [Goetz]
- min(2λ,2λ) = 2λ [Priami]
- 1/(1/(2λ)+1/(2λ)) = λ [PEPA]
- 2λ*1 = 2λ (passive inputs)

Possible Interactions

The speed of interaction is proportional to the number of possible interactions.
But a process cannot interact with itself.

Assume each process P is in restricted-sum-normal-form. For each channel x:

In(x,P) = Num of active ?x in P

Out(x,P) = Num of active !x in P

Mix(x,P) = In(x,P)*Out(x,P)
#interactions that cannot happen in a given summation P

In(x) = Sum P of In(x,P)

Out(x) = Sum P of Out(x,P)

Mix(x) = Sum P of Mix(x,P)
total #interactions that cannot happen

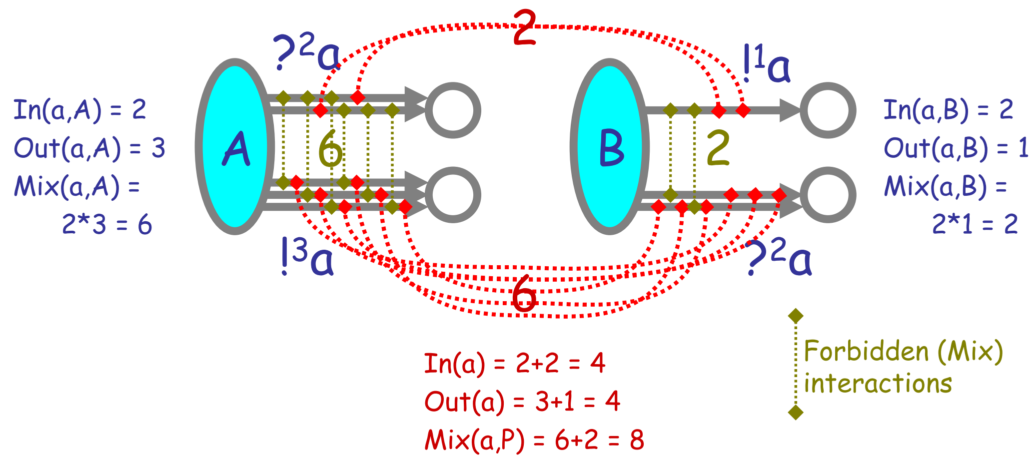
The global Activity on channel x:

Act(x) = (In(x)*Out(x))-Mix(x)

total cross product of inputs and outputs minus total #interactions that cannot happen

The global speed of interaction on a channel x:

speed(x) = Act(x)*rate(x)

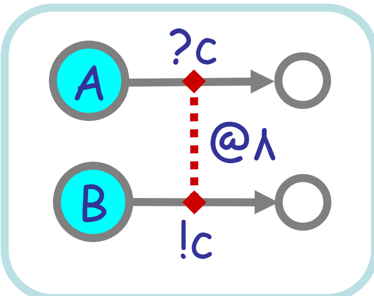


Act(a) = (In(a) * Out(a)) – Mix(a) = 4*4 – 8 = 8

speed(a) = Act(a)*rate(a) = 8*rate(a)

Deriving Back Interaction Laws

The mass action law:



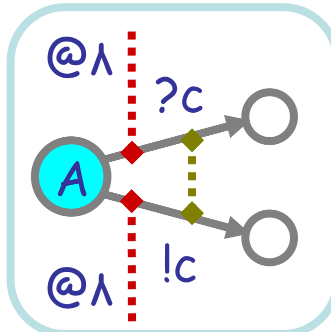
$[A]' = -speed(c) = -\lambda\ Act(c)$

$Act(c) = (In(c)*Out(c))-Mix(c)$

$= ([A]*[B]) - 0$

hence $[A]' = -\lambda [A][B]$

The mixed interaction law:



$[A]' = -speed(c) = -\lambda\ Act(c)$

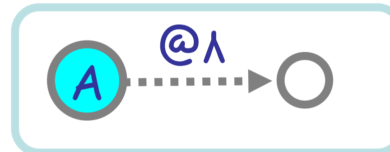
$Act(c) = (In(c)*Out(c))-Mix(c)$

$= ([A]*[A]) - [A] = [A]*([A]-1)$

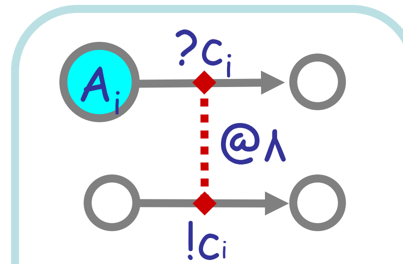
hence $[A]' = -\lambda [A] ([A]-1)$

$Act(x) = (In(x)*Out(x))-Mix(x)$

The decay law:



$=_{def}$



(Each A_i has its own private channel c_i)

$[A]' = \Sigma(c_i) -speed(c_i)$

$= \Sigma(c_i) -\lambda\ Act(c_i)$

$Act(c_i) = (In(c_i)*Out(c_i))-Mix(c_i)$

$= (1*1) - 0 = 1$

hence $[A]' = -\lambda [A]$

Conclusions

Conclusions

- Stochastic Collectives
  - Complex global behavior from simple components
  - Emergence of collective functionality from "non-functional" components
  - (C.f. "swarm intelligence": simple global behavior from complex components)

- Artificial Biochemistry
  - Stochastic collectives with Law of Mass Interaction kinetics
  - Connections to classical Markov theory,
    chemical Master Equation, and Rate Equation

- The agent/automata/process point of view
  - "Individuals" that transition between states
    (vs. transmutation between "unrelated" chemical species)
  - More appropriate for Systems Biology
  - Stochastic π-calculus (SPiM) for investigating stochastic collectives
    - Restriction+Communication $\Rightarrow$ Polymerization: FSA that "stick together"