# Molecules as Automata


## Luca Cardelli

Microsoft Research
Cambridge UK

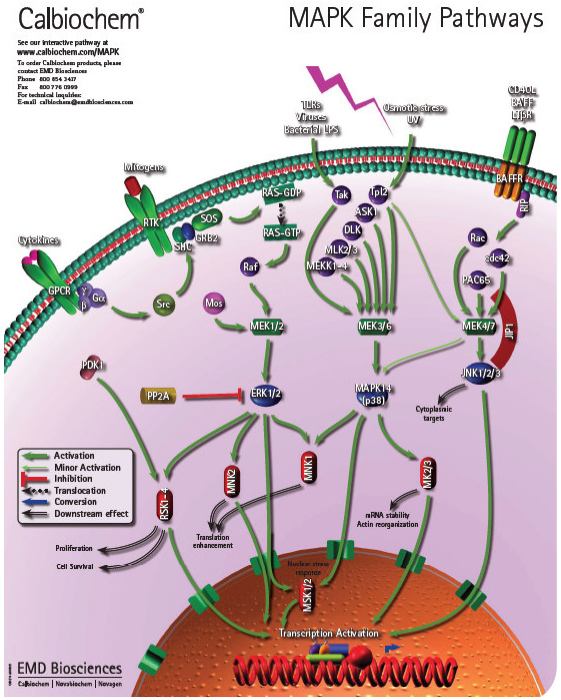


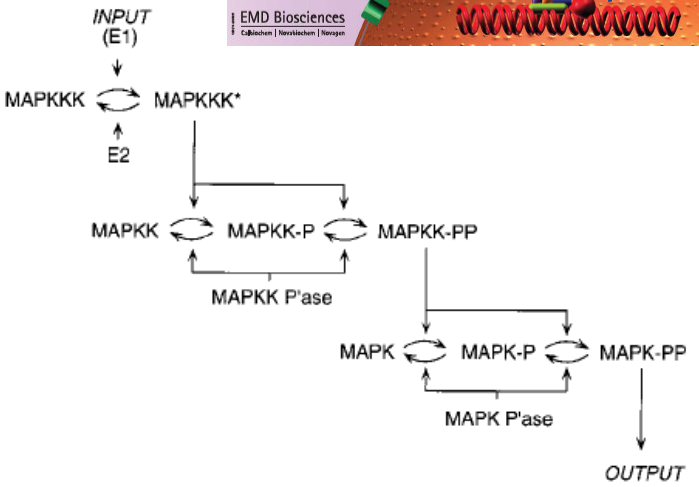
Molecular Programming Project Workshop
Oxnard, 2009-01-08


http://LucaCardelli.name

# Motivation: Cells Compute

- No survival without computation!
  - Finding food
  - Avoiding predators
- *How* do they compute?
  - Unusual computational paradigms.
  - Proteins: do they work like electronic circuits?
  - Genes: what kind of software is that?
- Signaling networks
  - Clearly "information processing"
  - They are "just chemistry": molecule interactions
  - But what are their principles and algorithms?
- Complex, higher-order interactions
  - MAPKKK = MAP Kinase Kinase Kinase:
    that which operates on that which operates on that which operates on protein.
- General models of biological computation
  - What are the appropriate ones?




Ultrasensitivity in the mitogen-activated protein cascade, Chi-Ying F. Huang and James E. Ferrell, Jr., 1996, *Proc. Natl. Acad. Sci. USA*, 93, 10078-10083.

# Theory of Computation

- Alan Turing
  - Defined what it means for a problem to be “computable”.
  - Showed that deciding weather an arbitrary mathematical conjecture is true or false is *not computable* (shocking mathematicians). (1936)
  - Also introduced the notion of “universal computation” (now called Turing Completeness): a *single machine* can be built that can compute *any* computable problem. We now call it a computer.
  - These were results in Mathematical Logic, but eventually established Computer Science as a separate discipline.

- John von Neumann
  - Was involved in the design of early electronic computers. The so-called von Newman architecture is at the basis of most computers from the 50’s on.
  - The von Neumann architecture is now seen as a liability: it is strictly sequential and arguably does not make good use of the massive concurrency of electronic hardware. (C.f. massive concurrency of biological systems.)
  - He also developed the foundations of Automata Theory (including cellular automata and robotic self-replication).

# Theory of Concurrency

- Early Automata Theory
  - Either about single isolated automata, or about "synchronous" homogeneous collections of automata, like cellular automata.
  - But what about multiple heterogeneous automata talking to each other? This question led to two major developments:

- Petri Nets
  - Dedicated to the study of *causality* relationships between *events*.
  - Providing a basic mathematical model with rich analytical techniques.

- Process Algebra
  - Dedicated to the study of concurrent, nondeterministic, *reactive systems*.
  - Endowing *concurrent languages* with a mathematical semantics.
  - Can provide foundations and inspiration for molecular programming, because molecular interactions are massively concurrent and heterogeneous.

# Reactive Systems

- A complex system does not compute a function
  - What function does E-coli compute?
  - Organisms, operating systems, computer networks, do not compute functions: they *indefinitely* react to stimuli and hold *internal state*.
  - Hence we need a mathematical treatment of *interactions*, not of *functions*. This has long been recognized and addressed in Computer Science.

- Reactive Systems
  - A system of components that each *react* to other components.
  - Each component is independently described in terms of its reactions to stimuli (which may come from many different other components).
  - The behavior of the system emerges from the free interactions of the components.
  - No a-priori description of all possible states (e.g. all possible molecular complexes) is needed.

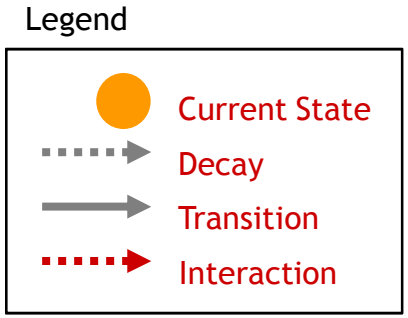
# Process Algebra

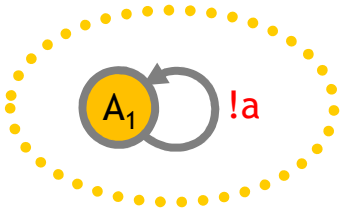
[Hoare, Milner, Pnueli, etc.]

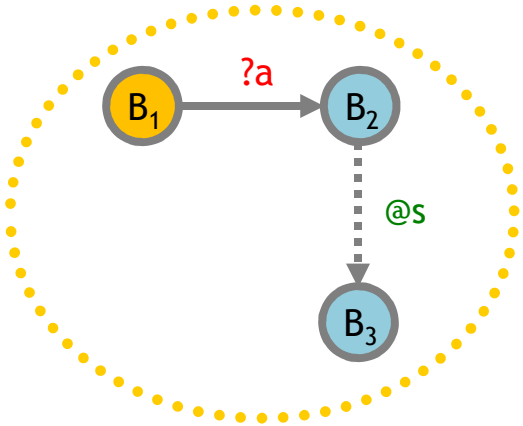
- Reactive systems (living organisms, computer networks, operating systems, …)
  - Math is based on *entities that react/interact with their environment* (“*processes*”), not on functions from domains to codomains.
- Concurrent
  - Events (reactions/interactions) happen concurrently and asynchronously, not sequentially like in function composition.
- Stochastic
  - Or probabilistic, or nondeterministic, but is never about deterministic system evolution.
- Stateful
  - Each concurrent activity (“process”) maintains its own local state, as opposed to stateless functions from inputs to outputs.
- Discrete
  - Evolution through discrete transitions between discrete states, not incremental changes of continuous quantities.
- Kinetics of interaction
  - An “interaction” is anything that moves a system from one state to another.

# (Macro-) Molecules as (Interacting) Automata

# Interacting Automata



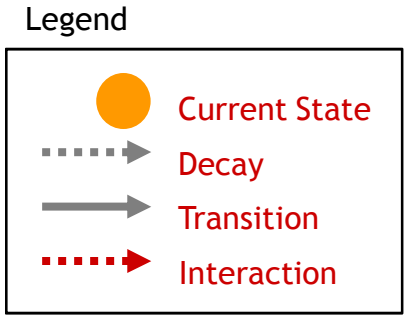



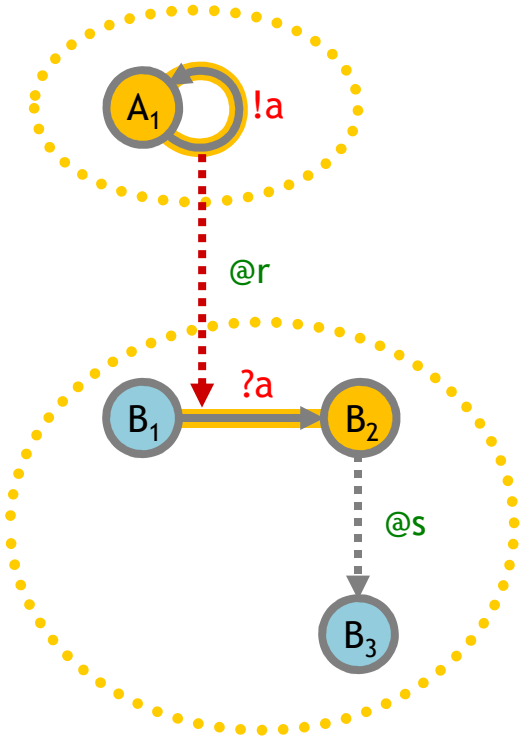




| | |
|---|---|
| $A_1$ | is a *state* |
| a | is a *channel* i.e. a named *interaction interface* (e.g. a surface patch) |
| ?,! | indicate any *complementarity* of interaction (e.g. charge) |
| ?a, !a | indicate *complementary actions*, |
| @r, @s | are rates |

***Kinetic laws:***

# Interacting Automata



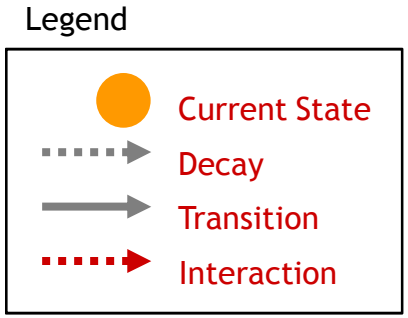



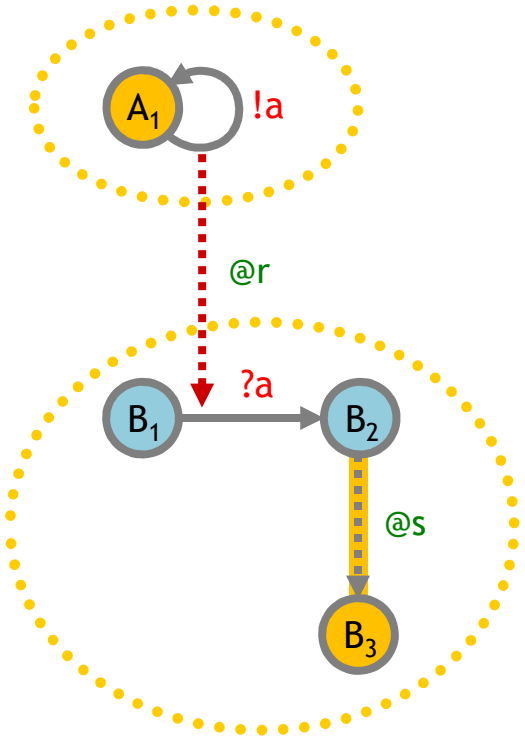

| | |
|---|---|
| $A_1$ | is a *state* |
| a | is a *channel* i.e. a named *interaction interface* (e.g. a surface patch) |
| ?,! | indicate any *complementarity* of interaction (e.g. charge, shape) |
| ?a, !a | indicate *complementary actions*, joined by an interaction arrow ⋯▸ |
| @r, @s | are rates |

***Kinetic laws:***

***Two complementary actions may result in an interaction.***

# Interacting Automata



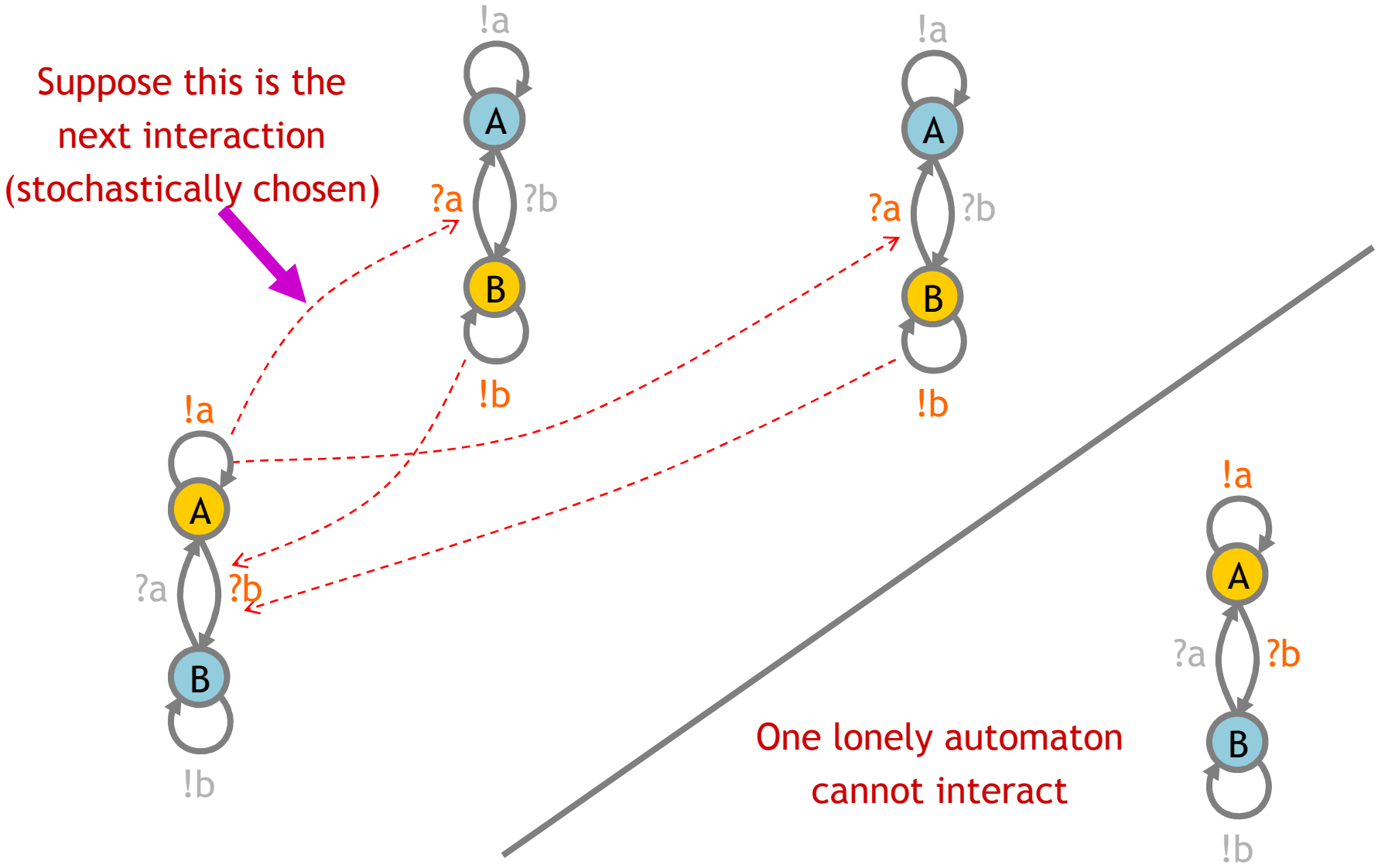

| | |
|---|---|
| $A_1$ | is a *state* |
| a | is a *channel* i.e. a named *interaction interface* (e.g. a surface patch) |
| ?,! | indicate any *complementarity* of interaction (e.g. charge) |
| ?a, !a | indicate *complementary actions*, joined by an interaction arrow ⋯▸ |
| @r, @s | are rates |

***Kinetic laws:***

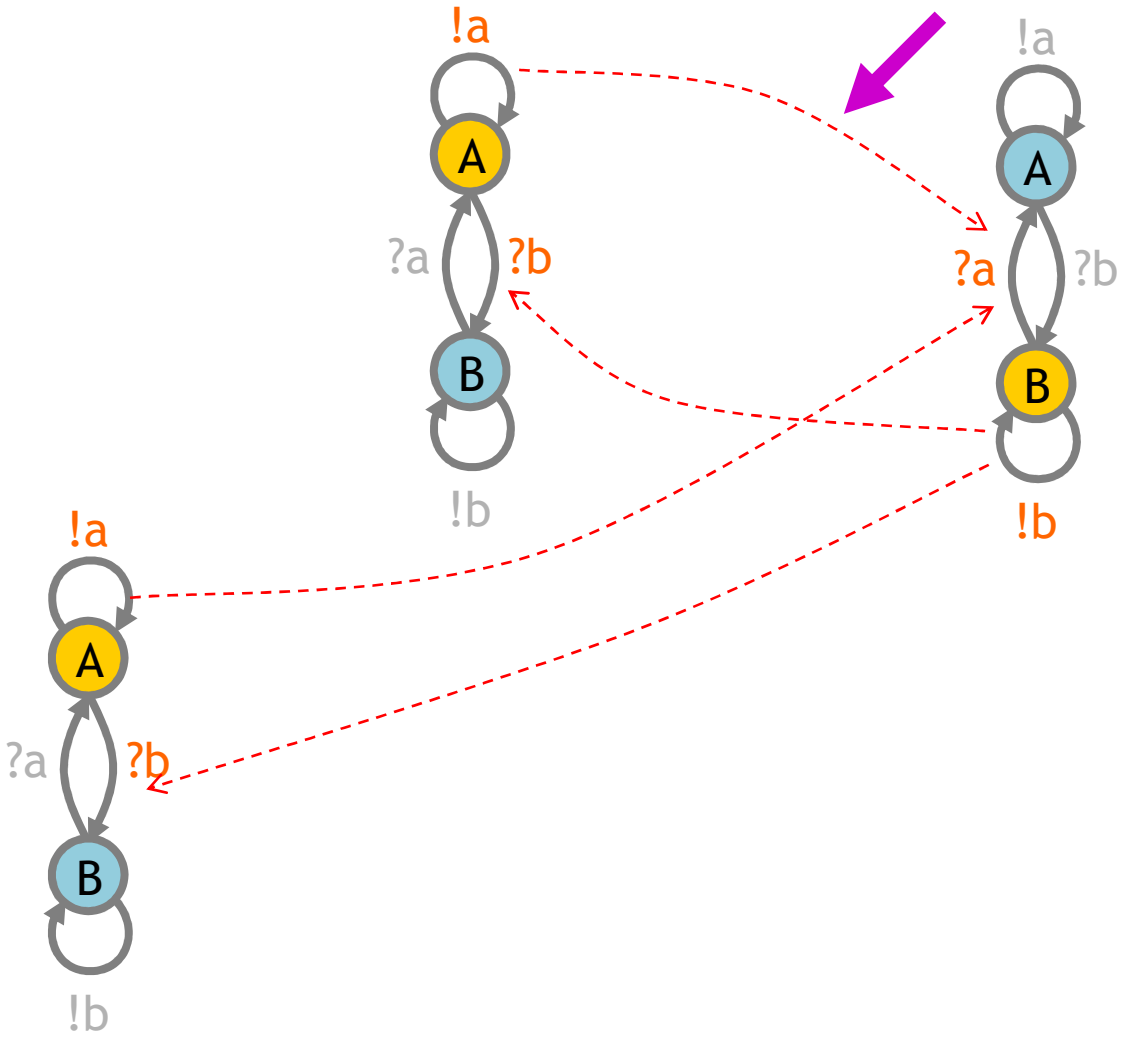
***Two complementary actions may result in an interaction.***

***A decay may happen spontaneously.***

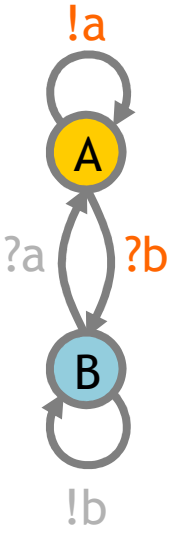
# Interactions in a Population

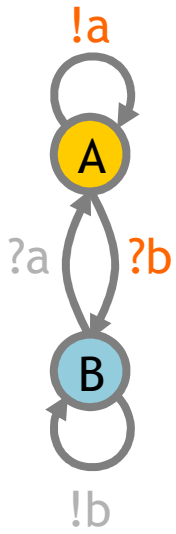


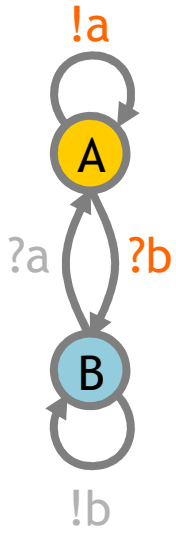
# Interactions in a Population



# Interactions in a Population

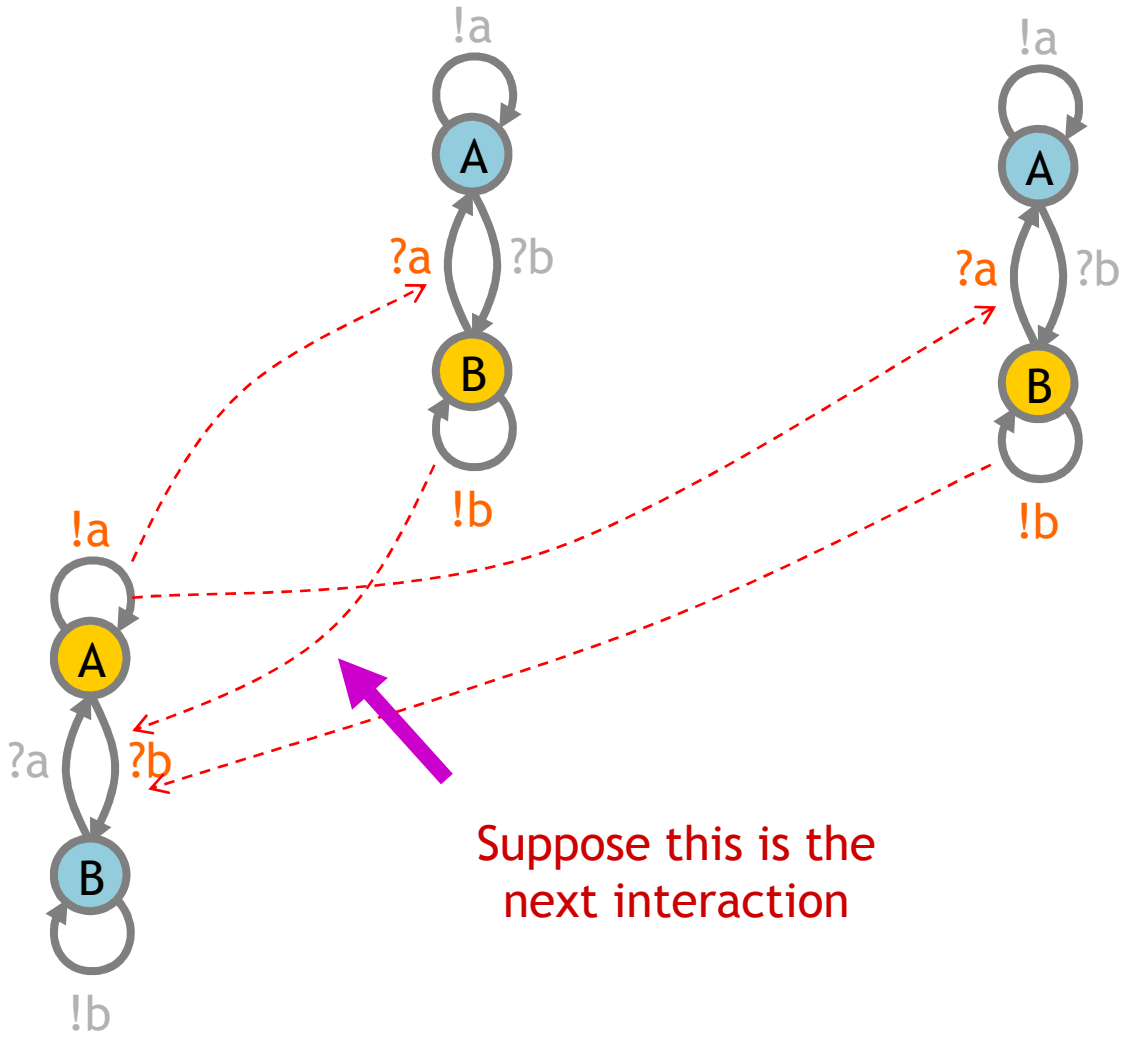


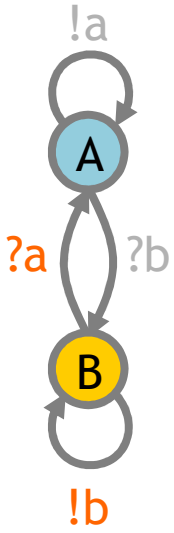



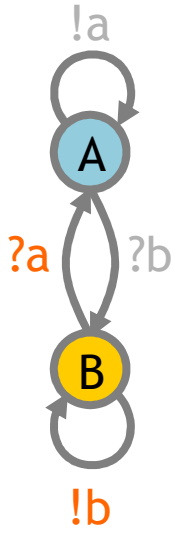

All-A stable population

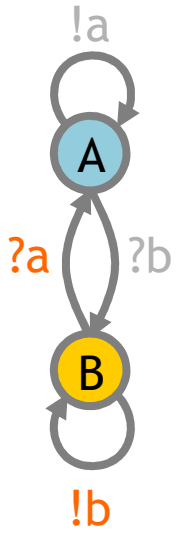
# Interactions in a Population (2)



# Interactions in a Population (2)



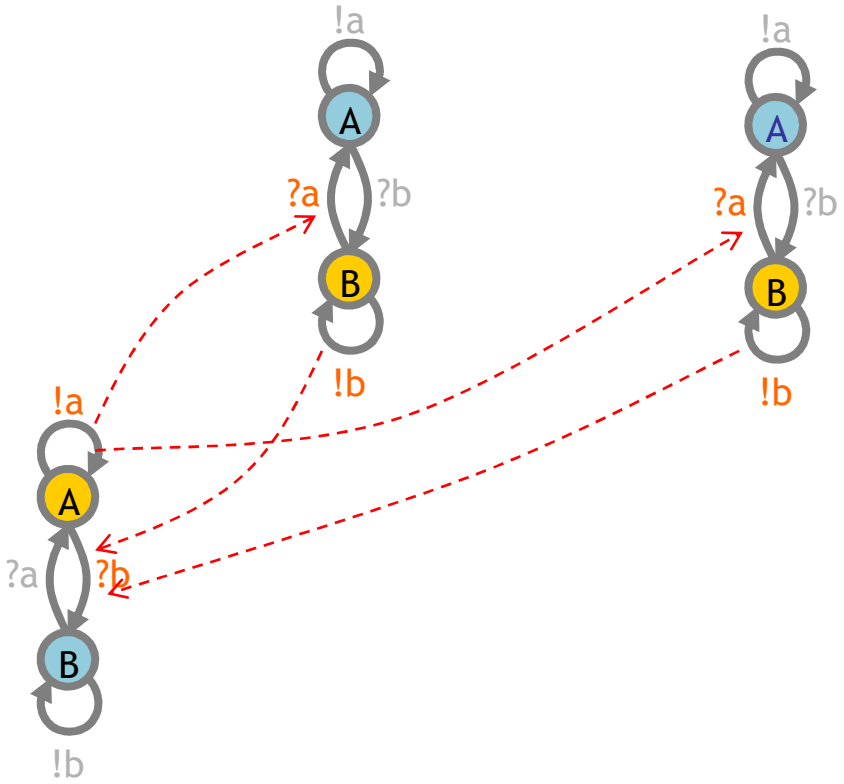



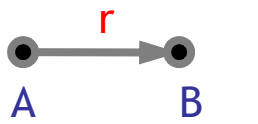




All-B stable population

Nondeterministic population behavior ("multistability")

# CTMC Semantics



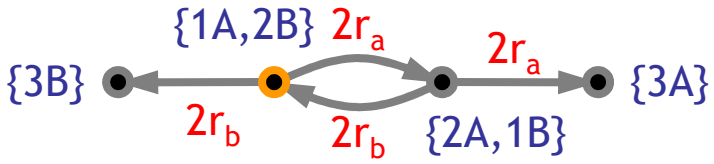




CTMC
(homogeneous) Continuous Time Markov Chain
- directed graph with no self loops
- nodes are system states
- arcs have transition rates

Probability of holding in state A:

$Pr(H_A>t) = e^{-rt}$

in general, $Pr(H_A>t) = e^{-Rt}$ where R is the sum of all the exit rates from A




CTMC

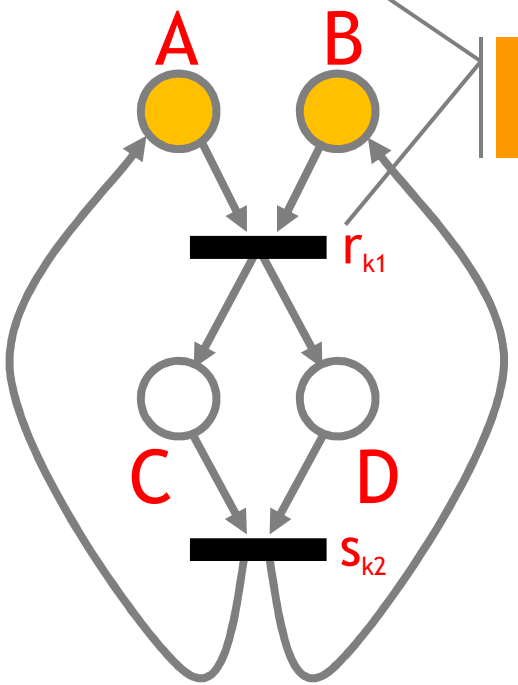
# Reactions vs. Components

Says what "A" *does*.

$r: A + B \rightarrow_{k1} C + D$

$s: C + D \rightarrow_{k2} A + B$

Does A become C or D?

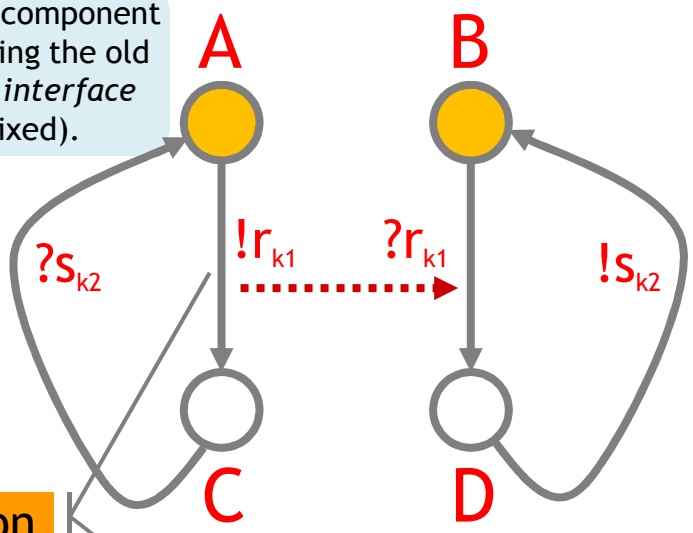



Reaction oriented

1 line per reaction

Says what "A" *is*.

Can add a new component without changing the old ones (if their *interface* remains fixed).




Interaction oriented

1 line per component

$A = !r_{k1}; C$

$C = ?s_{k2}; A$

$B = ?r_{k1}; D$

$D = !s_{k2}; B$

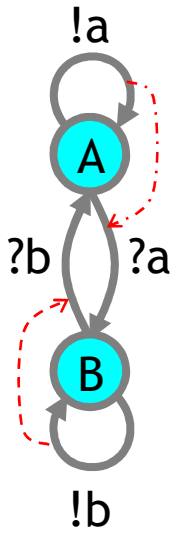
A becomes C not D!

The same "state space"

CTMC

# Groupies and Celebrities

# Groupies and Celebrities



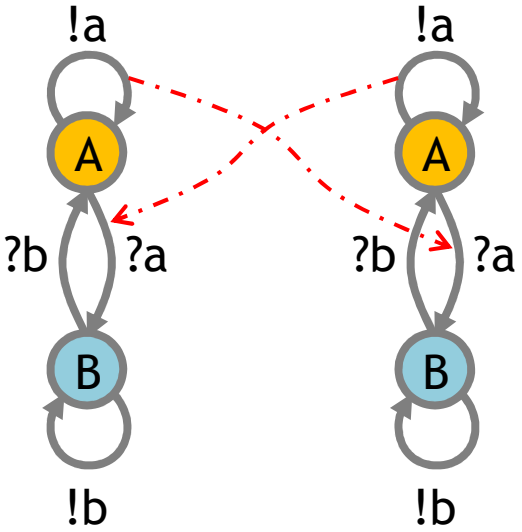

## Celebrity

(does not want to be like somebody else)

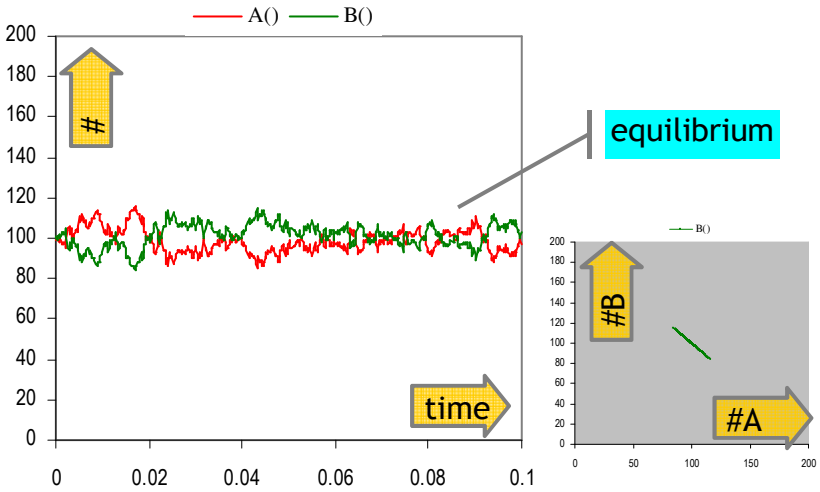
```
directive sample 1.0 1000
directive plot A(); B()

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())
```

a@1.0

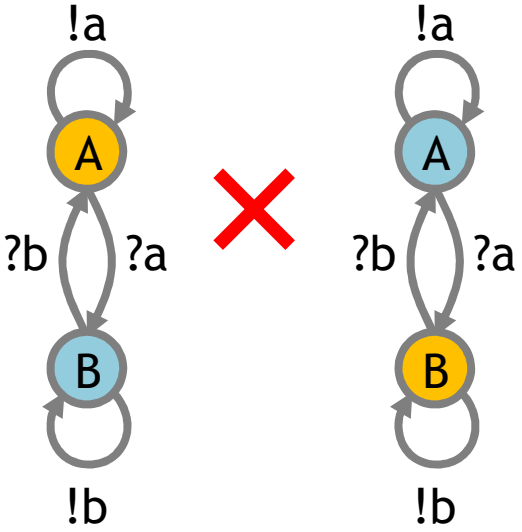
b@1.0



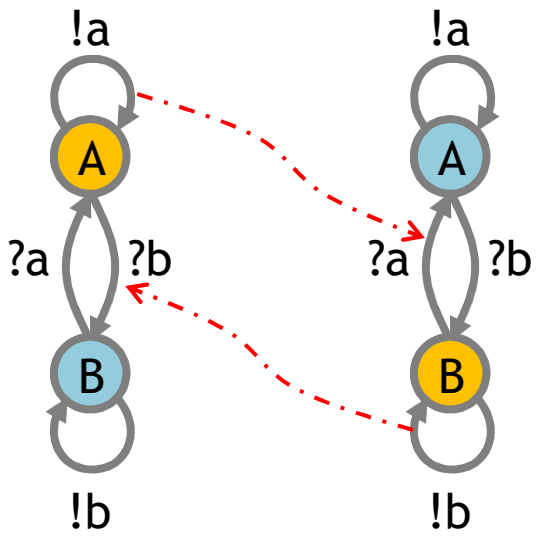

## A stochastic collective of celebrities:

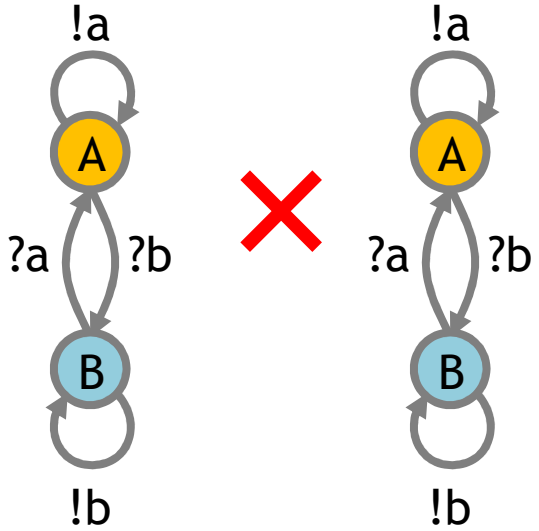


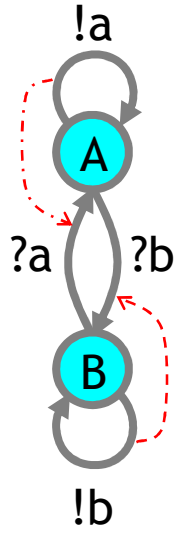

Stable because as soon as a A finds itself in the majority, it is more likely to find somebody in the same state, and hence change, so the majority is weakened.



# Groupies and Celebrities




## Groupie

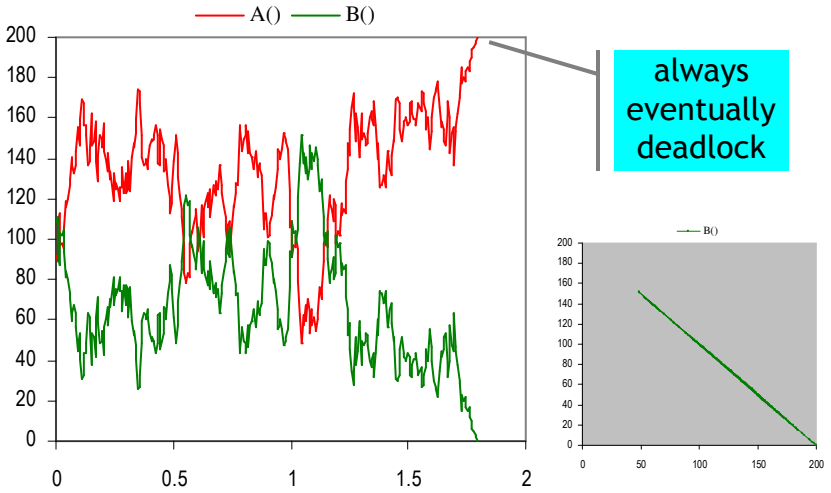
(wants to be like somebody different)

```
directive sample 1.0 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@1.0

b@1.0

## A stochastic collective of groupies:

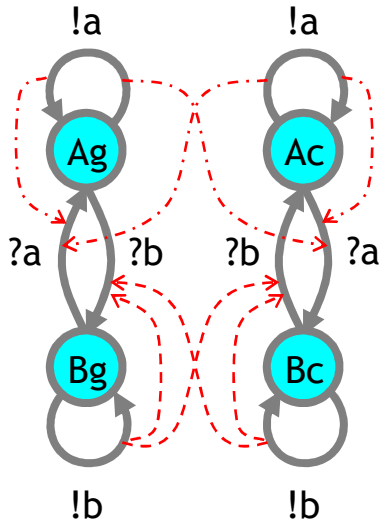



Unstable because within an A majority, an A has difficulty finding a B to emulate, but the few B's have plenty of A's to emulate, so the majority may switch to B. Leads to deadlock when everybody is in the same state and there is nobody different to emulate.

# Both Together

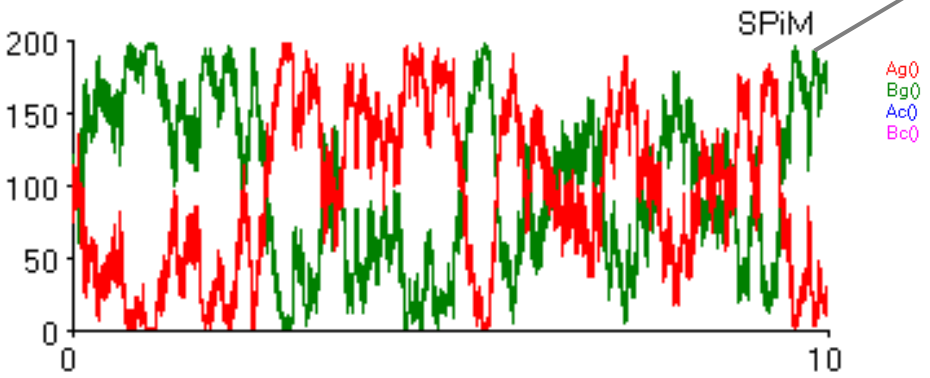
A way to break the deadlocks: Groupies with just a few Celebrities

Many Groupies




A few Celebrities

```
directive sample 10.0
directive plot Ag(); Bg(); Ac(); Bc()

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

let Ac() = do !a; Ac() or ?a; Bc()
and Bc() = do !b; Bc() or ?b; Ac()

let Ag() = do !a; Ag() or ?b; Bg()
and Bg() = do !b; Bg() or ?a; Ag()

run 1 of Ac()
run 100 of (Ag() | Bg())
```

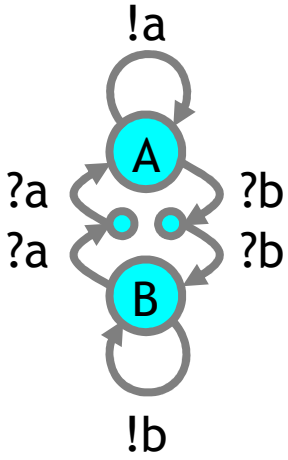


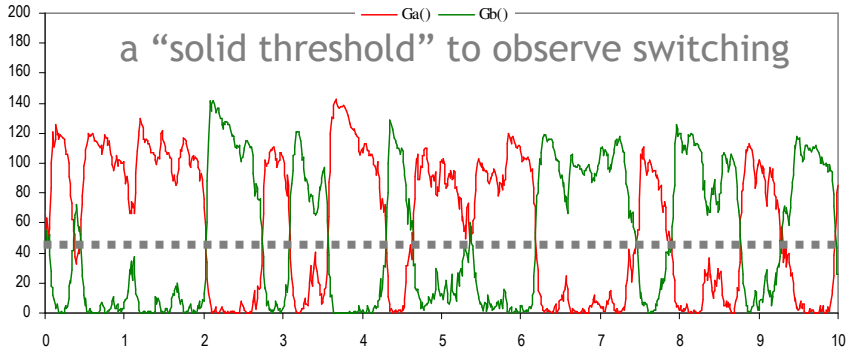

never deadlock

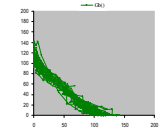
A tiny bit of "noise" can make a huge difference

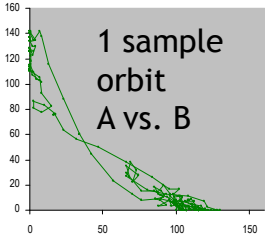
# Hysteric Groupies

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

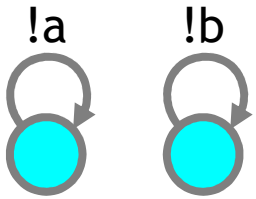




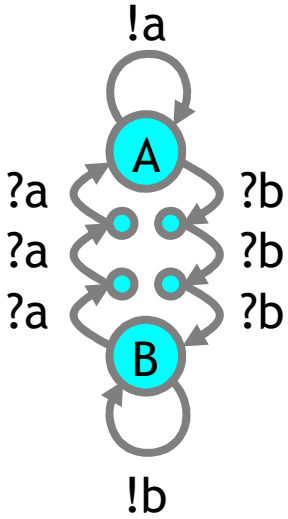

```
directive sample 10.0 1000
directive plot Ga(); Gb()

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())
```

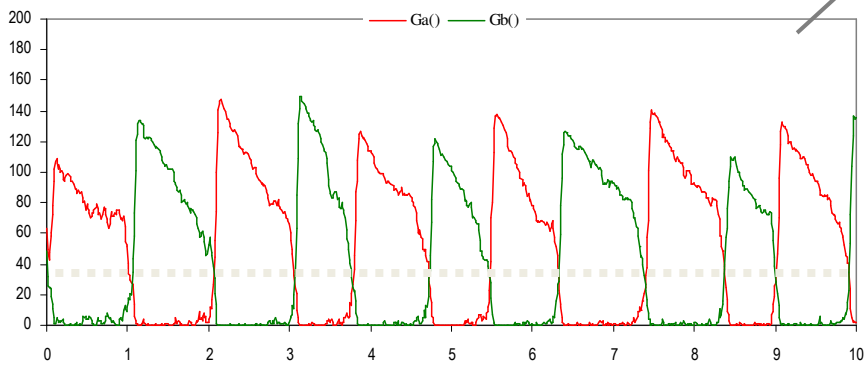


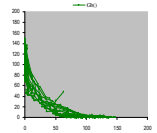

(With doping to break deadlocks)

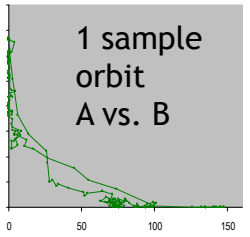
N.B.: It will not oscillate without doping (noise)

"regular" oscillation



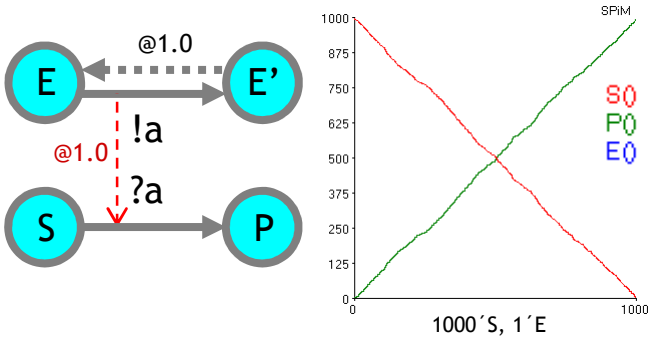





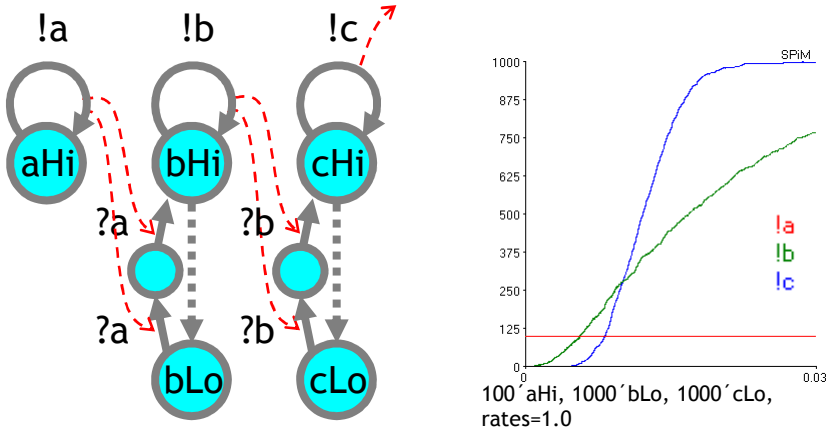

```
directive sample 10.0 1000
directive plot Ga(); Gb()

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())
```

# Some Devices

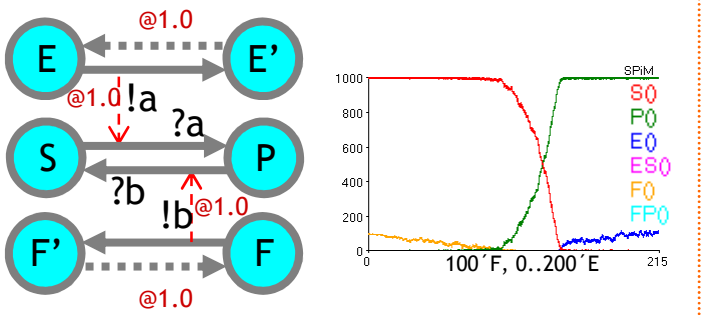
## Linear Pump



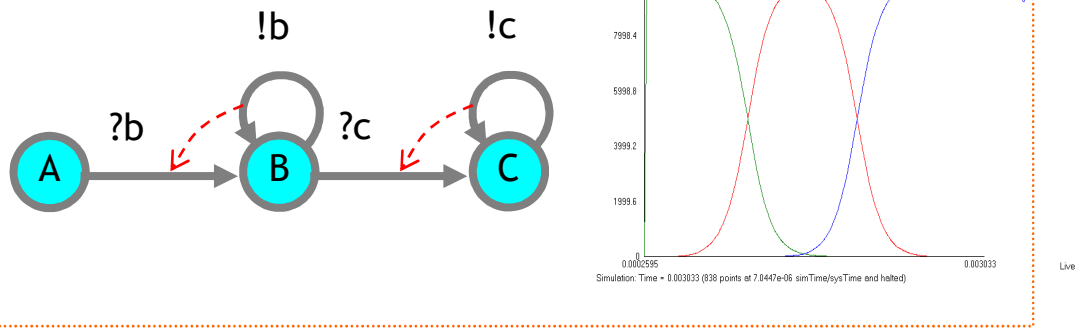

## Cascade Amplifier




## Ultrasensitive Switch

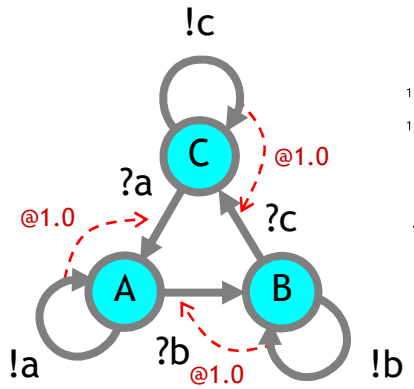


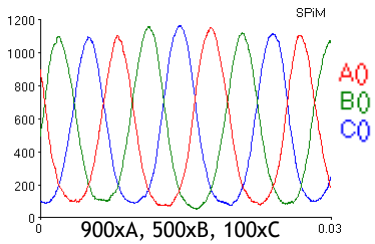

## Symmetric Wave Generator



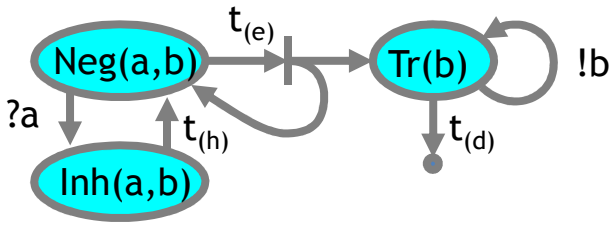
# More Devices

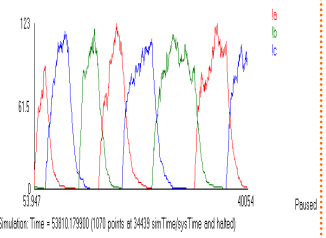
## Oscillator



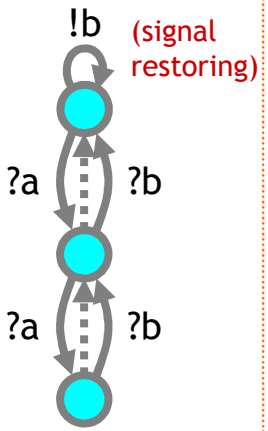



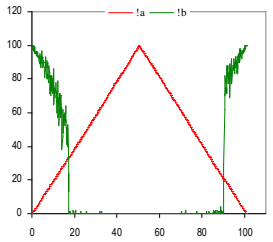

## Repressilator (1 of 3 similar gates)



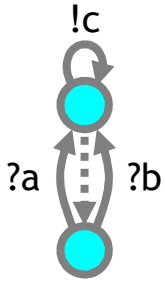



## b = not a

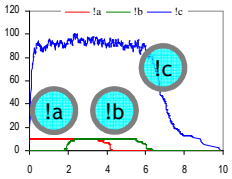


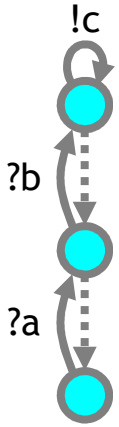
## c = a or b

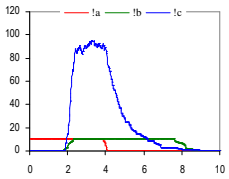



Inputs:
10 !a for 4t
2t; 10 !b for 4t

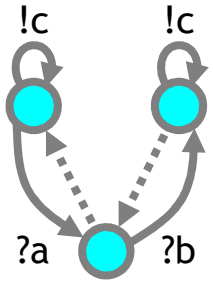


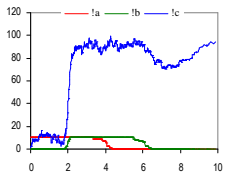

## c = a and b



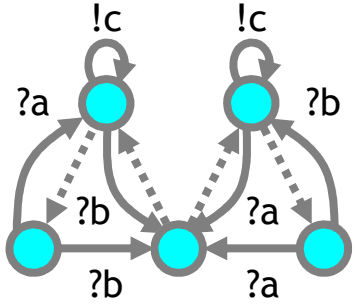



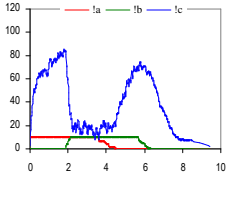
## c = a imply b

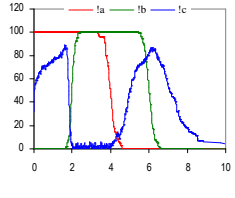


## c = a xor b

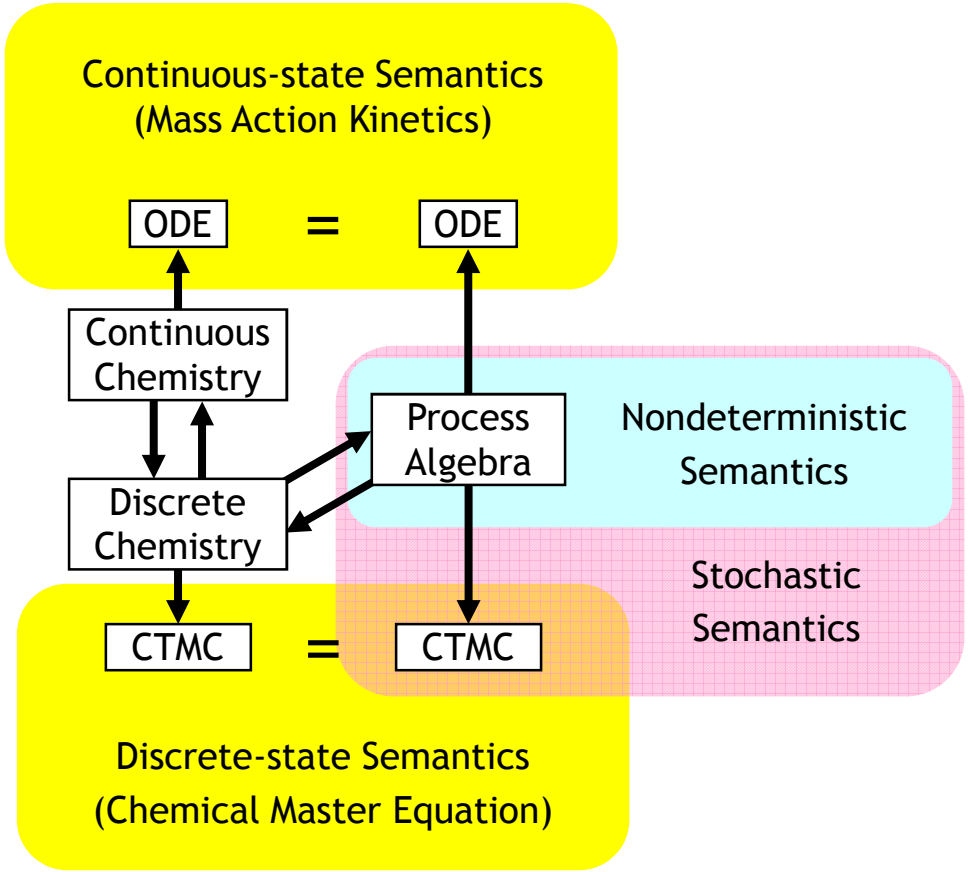




# Semantics of Collective Behavior

# The Two Semantic Sides of Chemistry

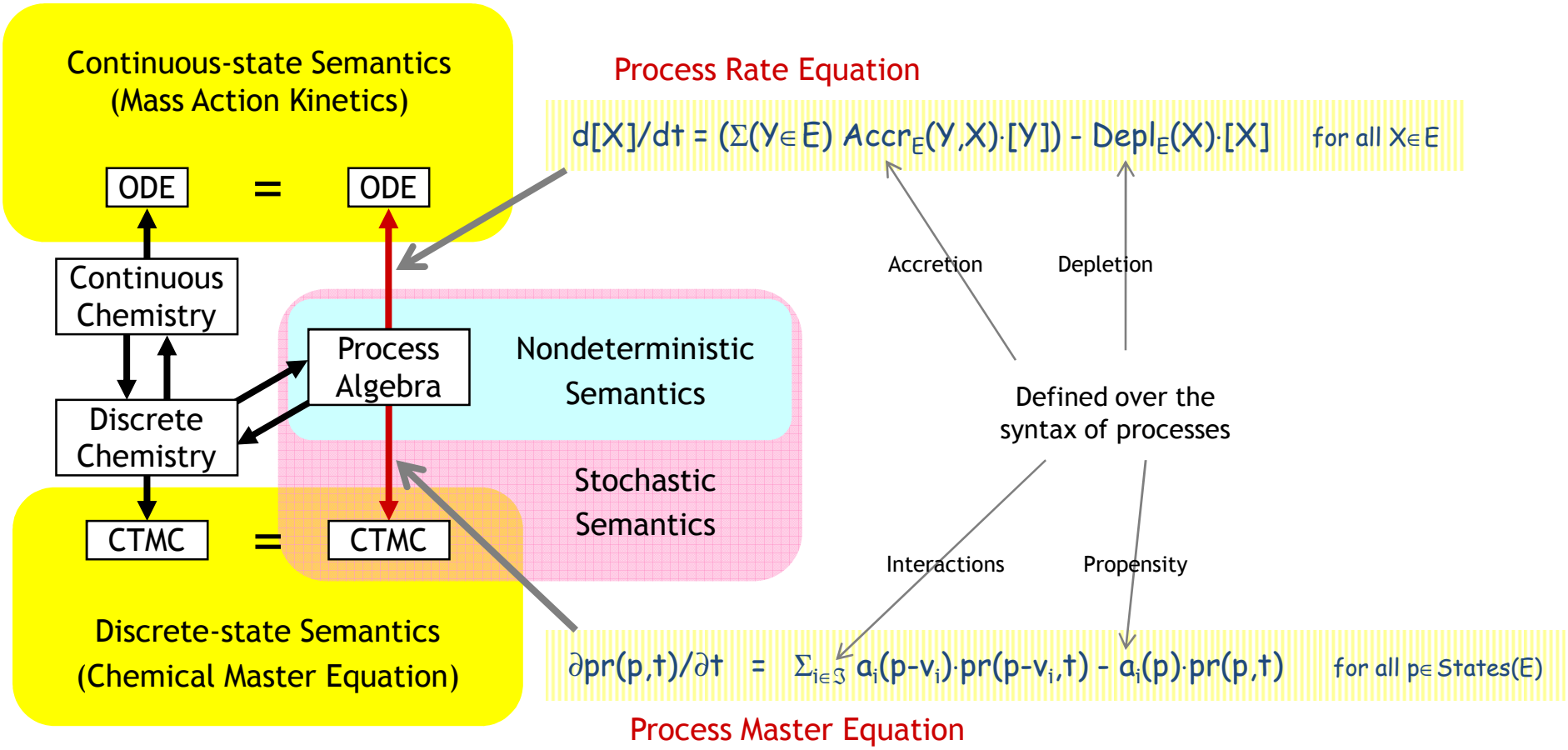



These diagrams commute via appropriate maps.

L. Cardelli: “On Process Rate Semantics” (TCS)

L. Cardelli: “A Process Algebra Master Equation” (QEST’07)

# Quantitative Process Semantics



# Process Algebra Beyond Chemistry
## (with Gianluigi Zavattaro)

# Turing Completeness

- Turing Completeness
  - A Turing Machine is “universal”: it can emulate any other computing device.
  - Your laptop is similarly a universal computing device.
  - Is chemistry universal: can chemistry emulate any computing device?

- Basic chemistry is equivalent to Petri Nets.
  - It is possible to translate any system of chemical reactions into a Place/Transition Petri Net (ignoring rates). Reachability of a dead (“halting”) states in P/T nets is decidable (an algorithm can answer yes/no).
  - By Turing’s theorem, if termination is decidable, i.e. if it is a simple problem, then the computational system is not universal. In particular, it cannot emulate a Turing machine (or your laptop).

- Hence “basic” chemistry is not Turing-complete!
  - Basic chemistry can’t compute! (Soloveichik et. al., Natural Computing 2008)
  - Even though stochastic chemistry is extremely rich, e.g. it includes chaotic systems.

# “Turifying” Chemistry

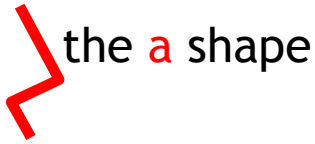
- Interacting Automata are not Turing complete
  - They are equivalent to chemistry, and to Petri Nets.

- What can we add to achieve Turing completeness?
  - It is not easy to add something to basic chemistry.
  - But it is easy to add power to simple automata.
  - E.g. we can go to full $\pi$-calculus, which is Turing complete.

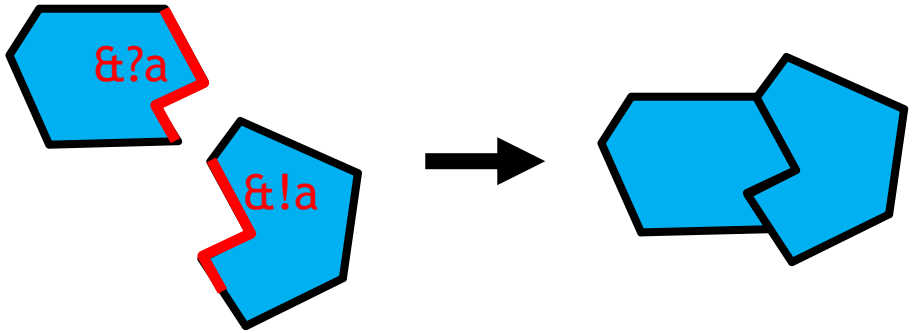
- But is there…
  - A *basic* mechanism
  - which is also biologically *realistic*?

# Association and Dissociation

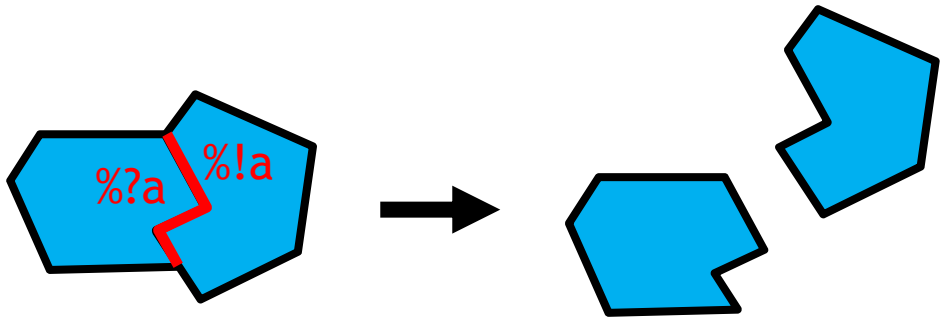
- Association patches are named




- & – association
  - &?a associate
  - &!a co-associate




- % – dissociation
  - %?a dissociate
  - %!a co-dissociate



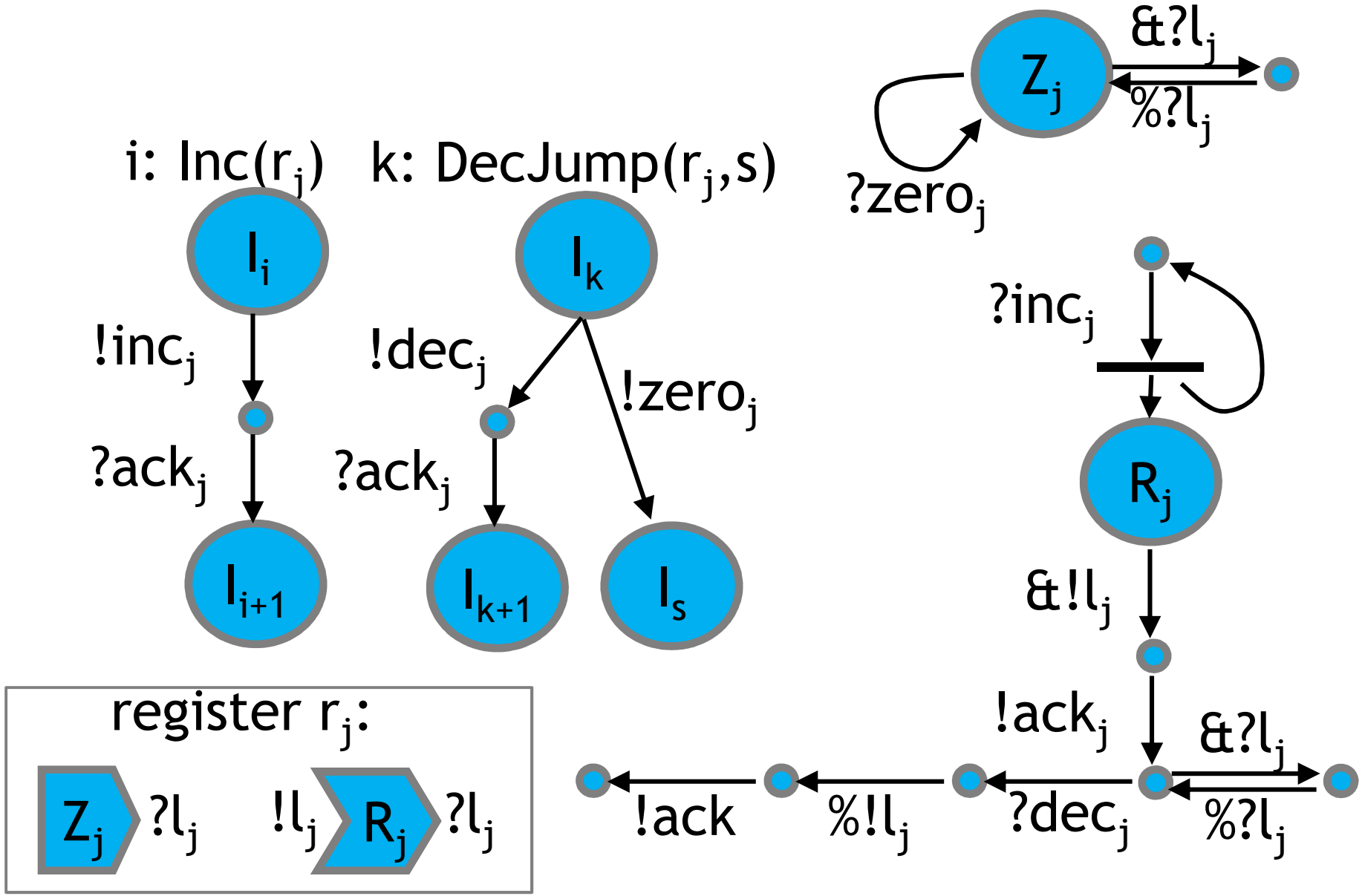

- A given patch can *hold* only one association at a time
- Two molecules can dissociate only if *they* are associated

# Turing completeness of "Biochemistry"

- Random Access Machines: [Min67]
  - **Registers**: $r_1$ … $r_n$ hold natural numbers (unbounded)
  - **Program**: finite sequence of numbered instructions
    - **i: Inc($r_j$)**: add 1 to the content of $r_j$ and go to the next instruction
    - **i: DecJump($r_j$,s)**: if the content of $r_j$ is not 0 then decrease by 1 and go to the next instruction; otherwise jump to instruction s

- There is a RAM encoding in BGF (automata with complexation)

# RAM encoding in BGF



# Why is This Easier in Process Algebra?

- Example: Linear Polymerization

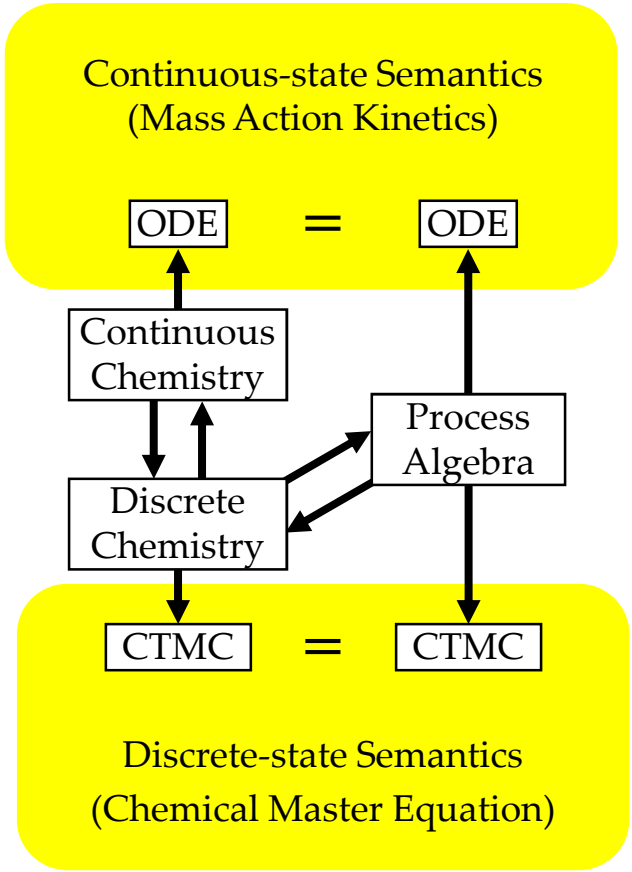
- In chemistry you have to write an *infinite list of reactions*
  - $P_0 + M \rightarrow P_1$
  - $P_1 + M \rightarrow P_2$
  - etc.
  - An infinite list of things is *not* a computation device!

- In process algebra you can write a *finite set of interactions*
  - A polymer (of any length) with a free surface, plus a monomer with a complementary surface, gives you another polymer with a free surface.
  - That's it.

- Process algebra descriptions are intrinsically more compact
  - This is an extreme case (finite vs. infinite)
  - But is also true for finite cases (linear vs quadratic or exponential).

# DNA Algebras

# Motivation

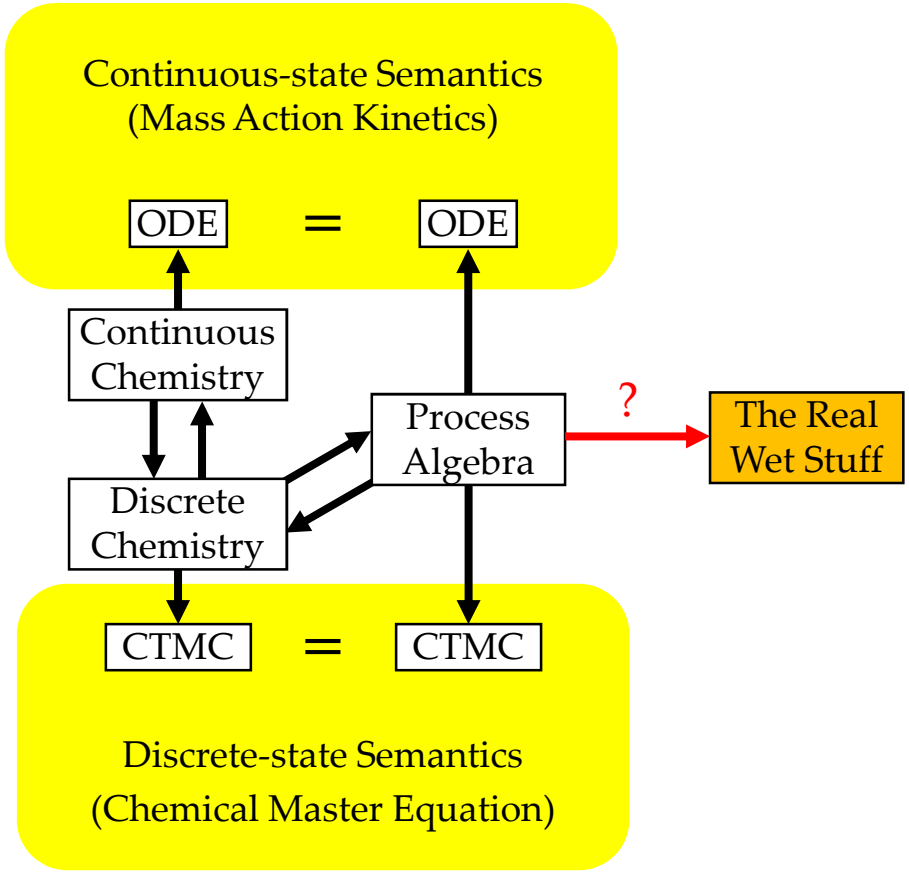



These diagrams commute via appropriate maps.

L. Cardelli: "On Process Rate Semantics" (TCS)

L. Cardelli: "A Process Algebra Master Equation" (QEST'07)

# Motivation

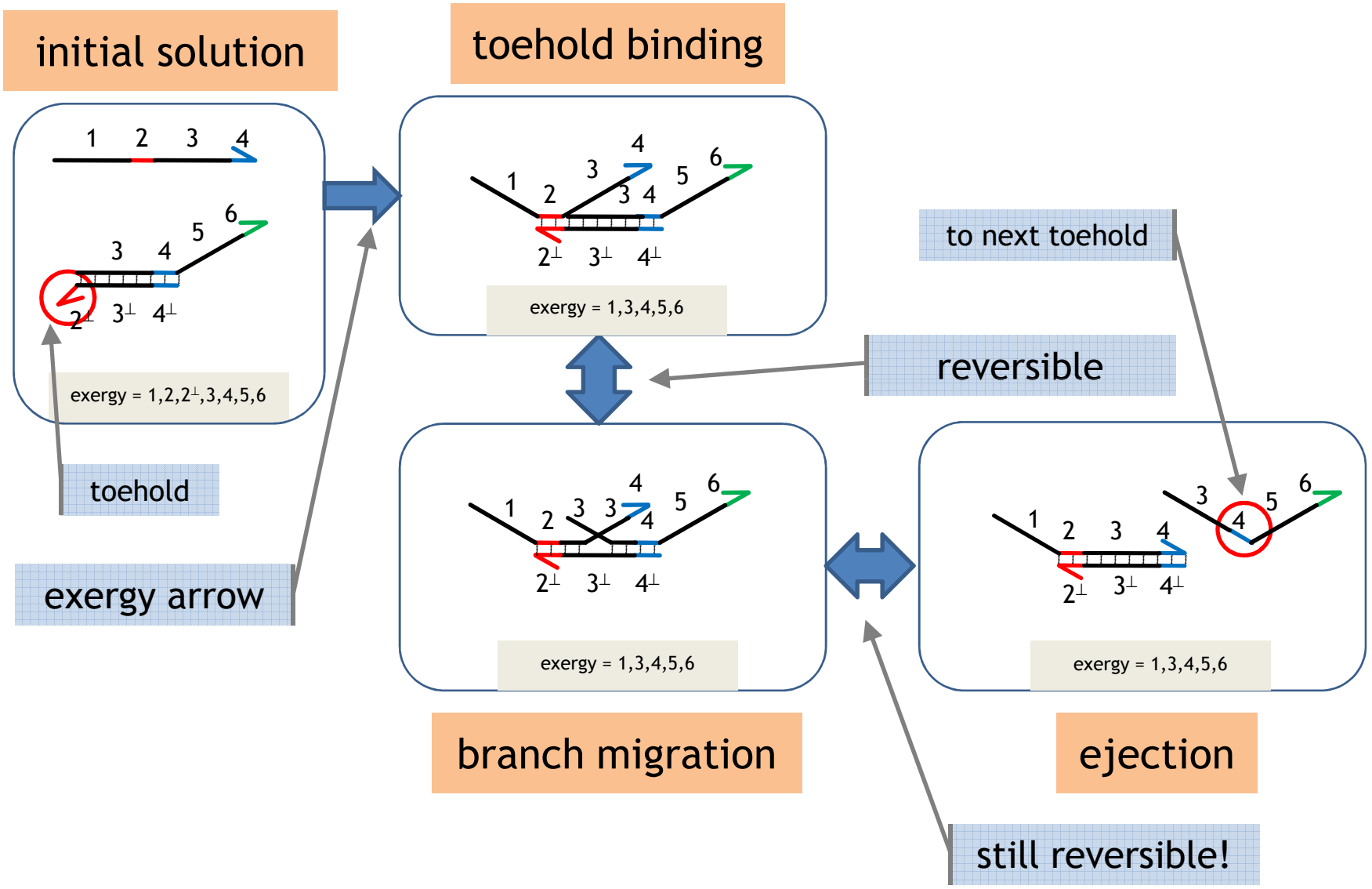



How do we implement an arbitrary process?

Chemistry does not necessarily help
(how do we then implement the chemical species?)

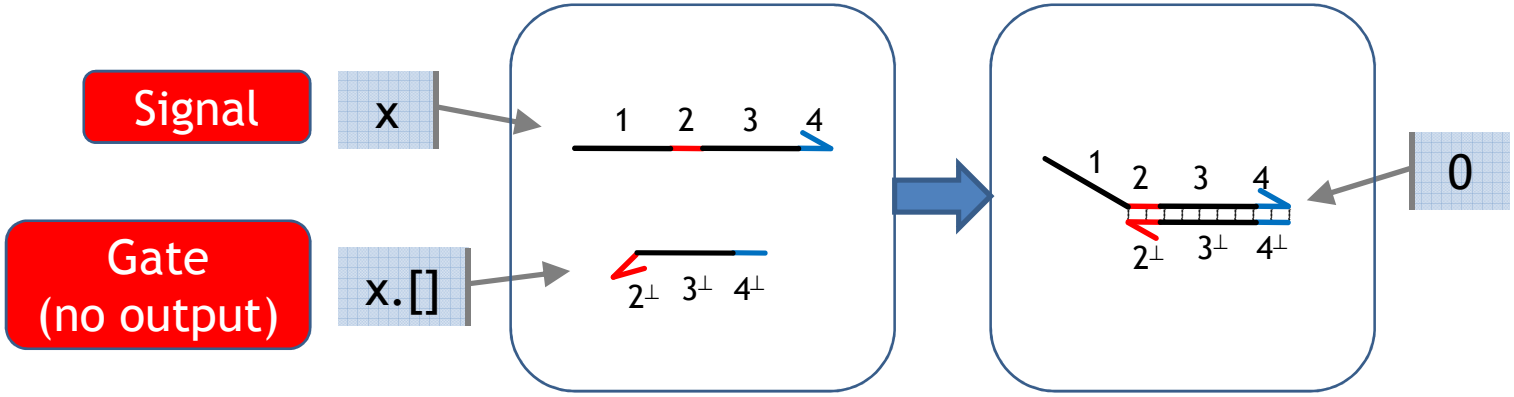
# Branch Migration

D. Soloveichik, G. Seelig, E. Winfree. **DNA as a Universal Substrate for Chemical Kinetics.** Proc. DNA14.



# Hybridization

$x \mid x.[] \rightarrow 0$

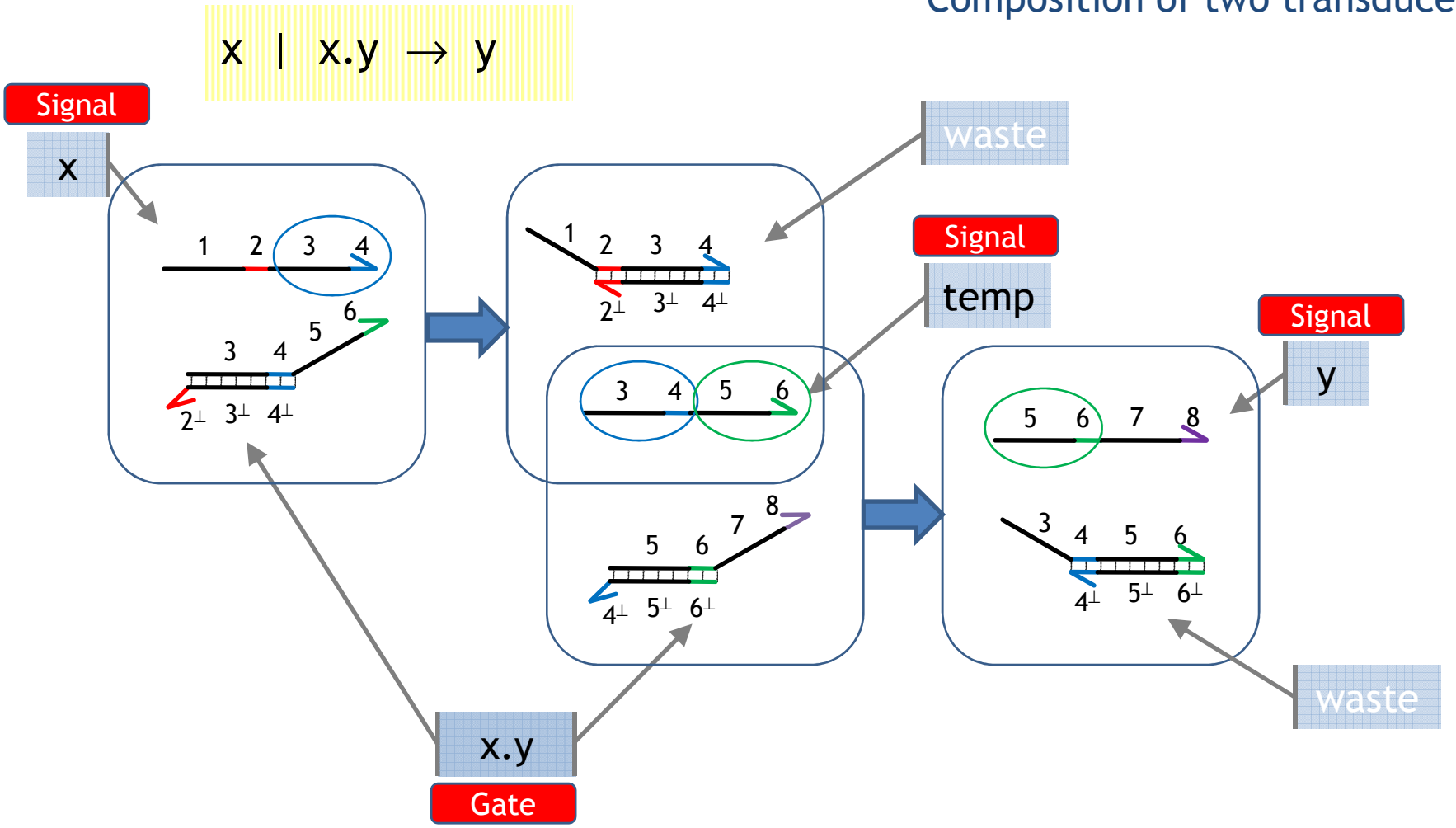



- A positive toehold indicates a signal.
- (A positive strand is a signal.)

- A negative toehold indicates a gate.
- (A negative strand is a gate with no output.)

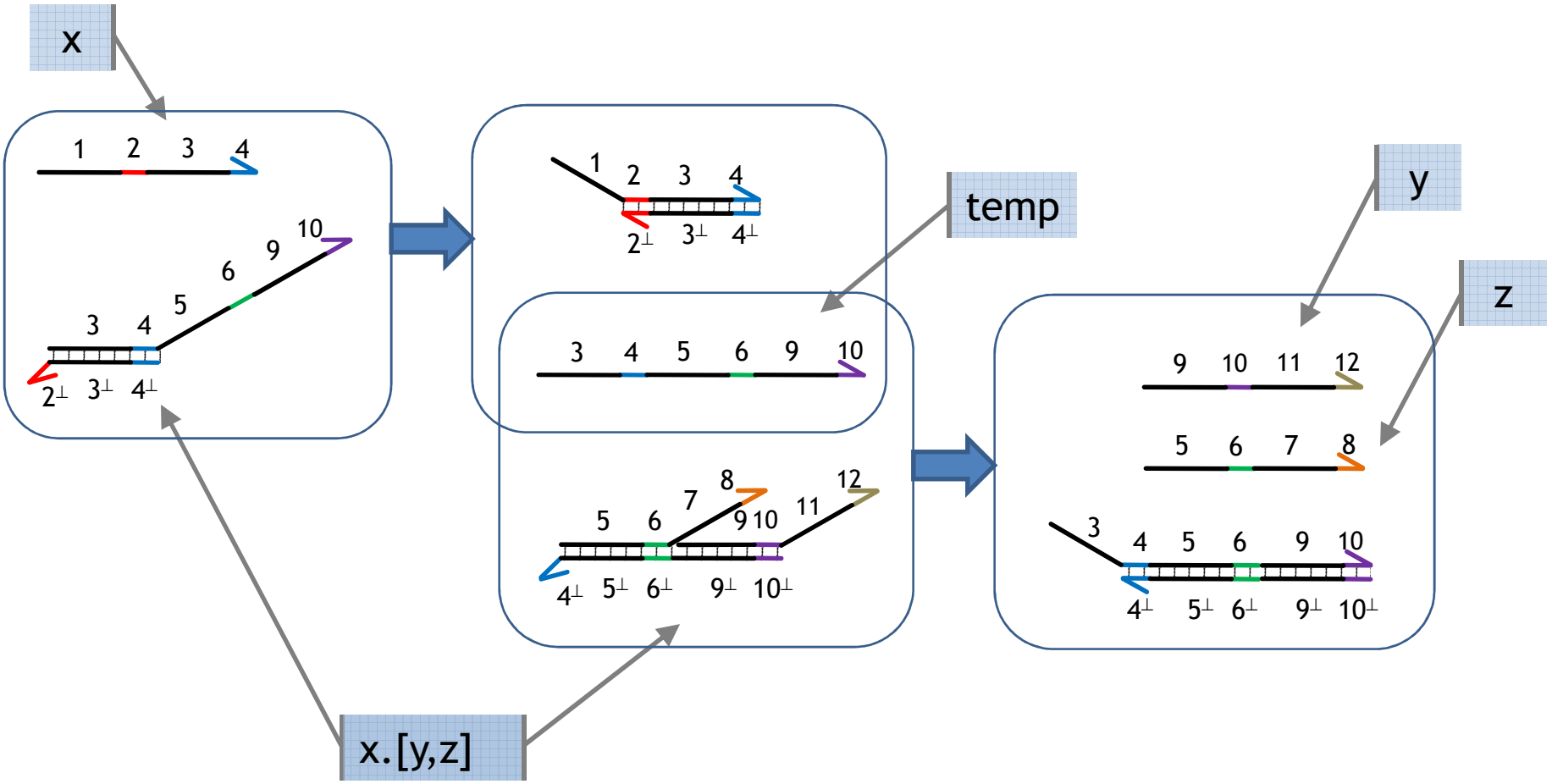
# Sequence

Composition of two transducers.



# Fork

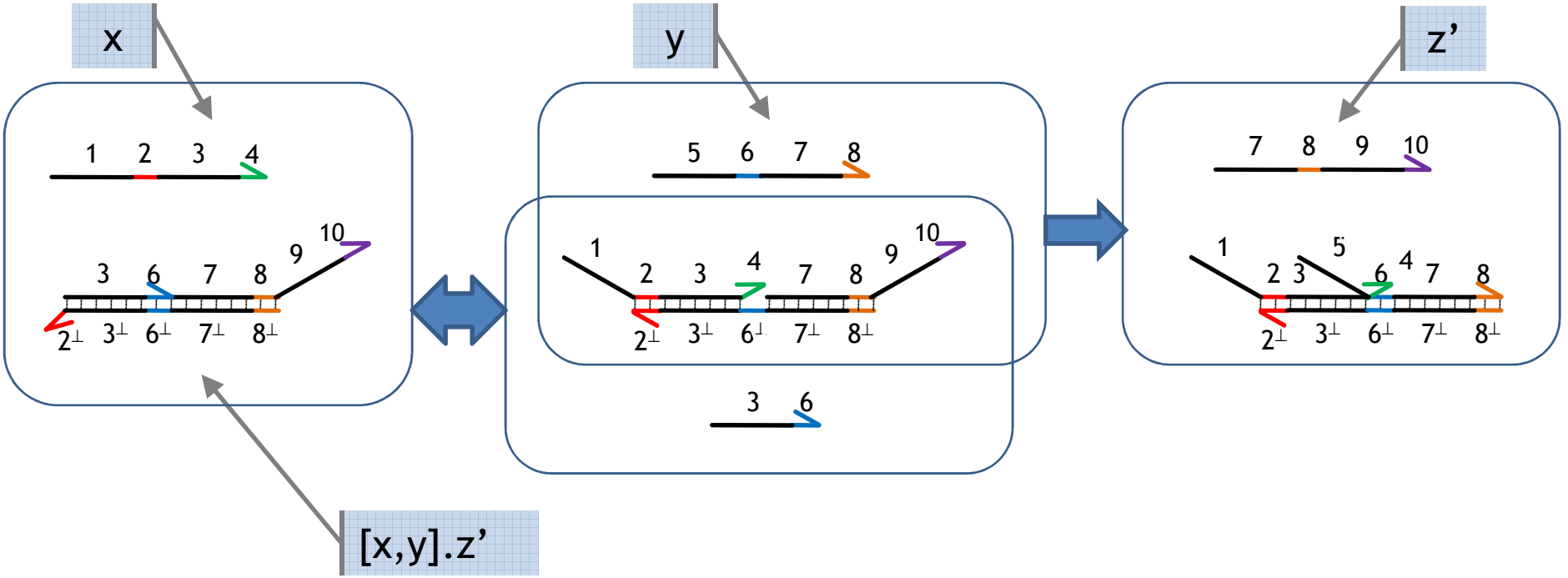
$x \mid x.[y,z] \rightarrow y \mid z$




Similarly to Sequence, with two output strands in second step (Soloveichik Fig 2)

# 2-way Join

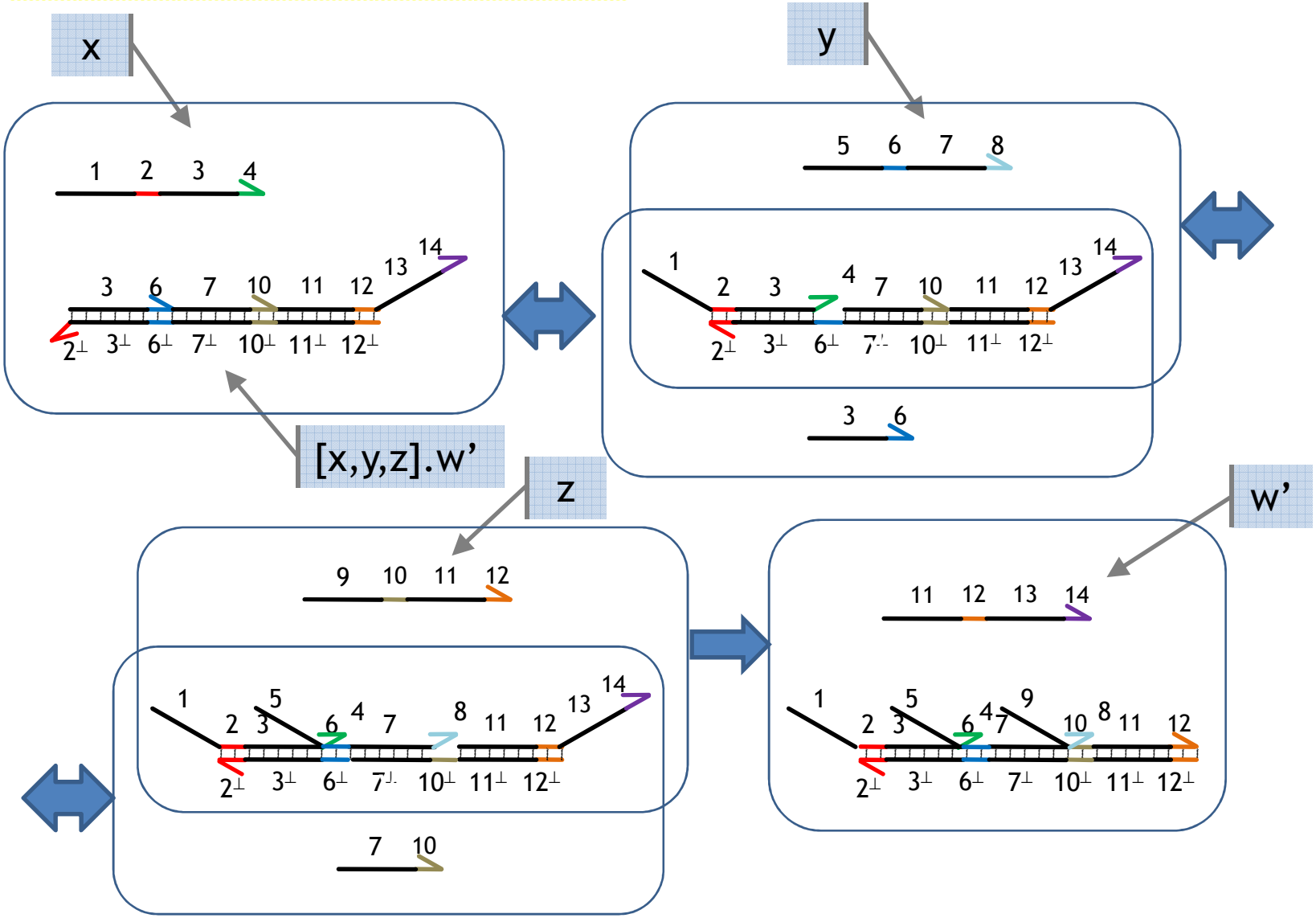
x | y | [x,y].z → z




This can be implemented by two sequential inputs where the first one is reversible (Soloveichik Fig.3). An output transducer is then needed from z’ to z, because z’ overlaps with y.

# 3-way Join

x | y | z | [x,y,z].w → w



# Strand Algebra

No compound expressions except for parallel composition P|P and populations P*.
Hence this is a combinator-based ("assembly") language.

Here x is a *strand*, and [..].[..] is a *gate*.

$$P \;::=\; x \;\vdots\; [x_1,..,x_n].[x_1',..,x_m'] \;\vdots\; 0 \;\vdots\; P|P \;\vdots\; P^* \qquad n\geq 1,\ m\geq 0$$

- $x$ — is a *strand*
- $x_1.x_2$ $\stackrel{\text{def}}{=}$ $[x_1].[x_2]$ — is a *sequence gate*
- $x.[x_1,..,x_m]$ $\stackrel{\text{def}}{=}$ $[x].[x_1,..,x_n]$ — is a *fork gate*
- $[x_1,..,x_n].x$ $\stackrel{\text{def}}{=}$ $[x_1,..,x_n].[x]$ — is a *join gate*

- 0 — is *inert*
- P|P — is *parallel composition* of strands and gates
- P* — is a *population* of strands and gates

Note: σ.P* is not in the syntax: populations are only top-level.
*C.f.*: Petri net *tokens* (strands) and *transitions* (gates).
However, here both strands and gates are *consumed* by interaction.

# Reaction Rule

$$x_1 \mid .. \mid x_n \mid [x_1,..,x_n].[x_1',..,x_m'] \rightarrow x_1' \mid .. \mid x_m' \qquad \text{Gate}$$

Unlike a transition in Petri Nets (which has a very similar rule) the gate in the strand algebra is *consumed* by the reaction. Petri Nets are emulated by taking large gate populations.

Unlike a reaction chemistry, the gate 'species' already contains an explicit description of what it will do, independently of the other species in the system.

# Compiling to Strand Algebra

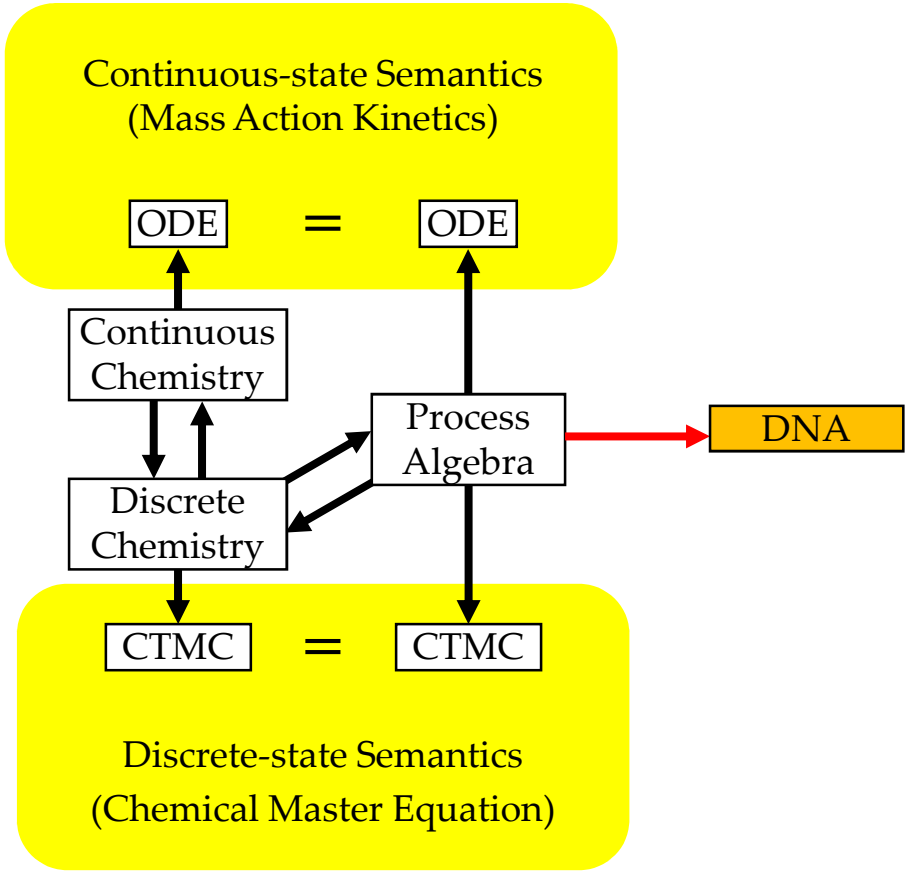
# High(er)-Level Languages

- We now have an intermediate language: the combinatorial strand algebra
  - It can be compiled "directly" to DNA following [Soloveichik et al.]
- But we really want to compile "high-level languages". Such as:
  - Boolean Networks (fairly easy)
  - Finite Stochastic Reaction Networks (Chemistry) [Soloveichik et al.]
  - Petri Nets (easy)
  - Finite State Automata and Transducers (easy)
  - Interacting Automata (hard to do directly)
  - $\pi$-calculus (??? not without DNA synthesis?)
- And also
  - Higher-level strand algebras, which may form more convenient intermediate languages.
  - Such as the *Nested Strand Algebra*.

# Automata to DNA

- There are many schemas to compile automata to molecules
  - But most (all?) are about compiling a single automaton (e.g. an FSA).
- Interacting Automata can be compiled to chemical reactions [TCS'08].
  - Are concurrent and population based (a subset of CCS).
  - The translation has an $n^2$ blowup (mean automata are "more compact").
  - But how does one engineer the necessary molecules?
- Arbitrary chemistry can be compiled to DNA [Soloveichik et al.].
  - The translation is stochastically "almost" faithful.
  - Which can be seen as a defect of the translation, if you are a chemist.
- Hence Interacting Automata can be compiled to DNA.
  - Again, stochastically this is "almost" faithful as a single transition may need to be implemented with two transitions, which have a different distribution.
  - But for automata, we are probably not picky: we mostly want them to change states.

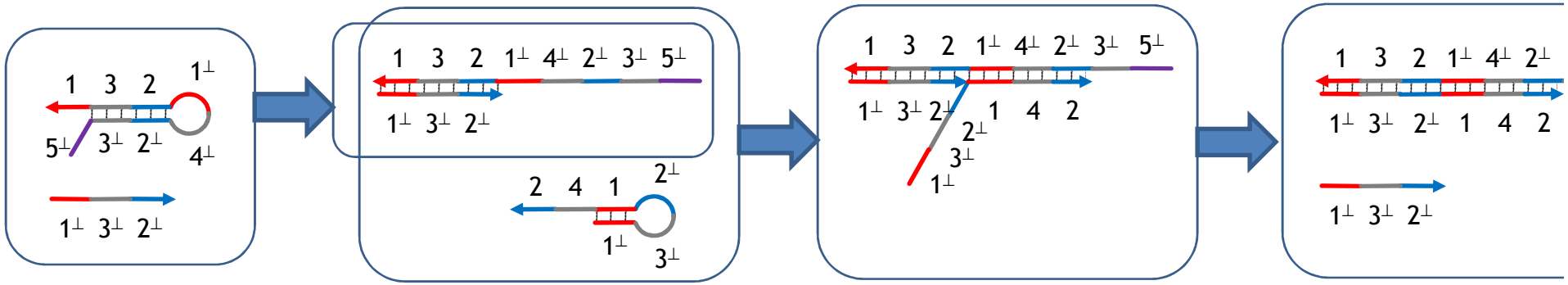
# Problem Solved!



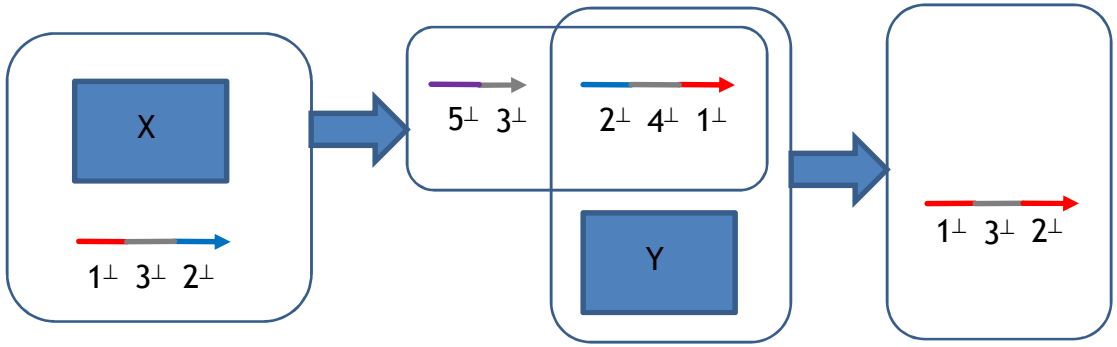
# Alternative DNA Mechanisms

# Hairpin Operators

Programming biomolecular self-assembly pathways
P. Yin, H.M.T. Choi, C.R. Calvert, N.A. Pierce
Nature, 451:318-322, 2008.

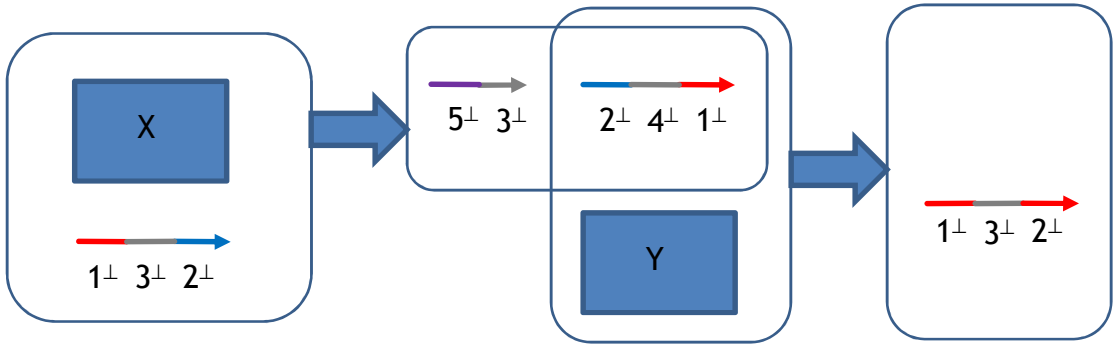



But this is the *function* of that network:



# Doing the same in Strand Algebra

If we just think of the *function* of that network:




Then we can implement that function in strand algebra:

$$X = (1^{\perp}:3^{\perp}:2^{\perp}).[(5^{\perp}:3^{\perp}),(2^{\perp}:4^{\perp}:1^{\perp})]$$
$$Y = (2^{\perp}:4^{\perp}:1^{\perp}).(1^{\perp}:3^{\perp}:2^{\perp})$$

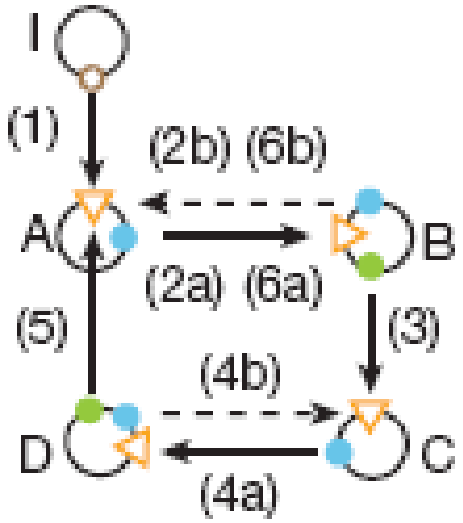
This leads to a different DNA implementation (according to the canonical DNA semantics of strand algebra).

But this is what algebra is good for: abstracting implementation models.

# A Network from Yin et al.

I = ia
A = ia.ab
B = ab.[ia,bc]
C = bc.cd
D = cd.[bc,ia]

Strand Algebra representation.




$I + A \rightarrow IA$
$IA + B \rightarrow I + AB$
$AB + C \rightarrow ABC$
$ABC + D \rightarrow AB + CD$
$CD + A \rightarrow CDA$
$CDA + B \rightarrow CD + AB$

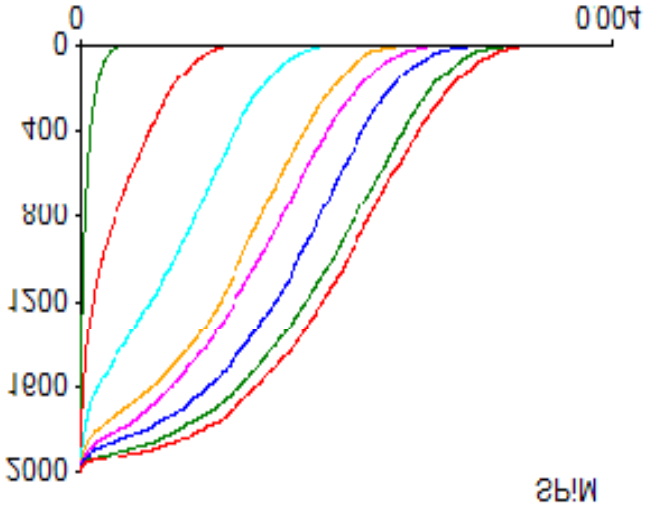
Chemical representation from Yin et al. (supp.)

Strand Algebra describes the function of each molecule independently.

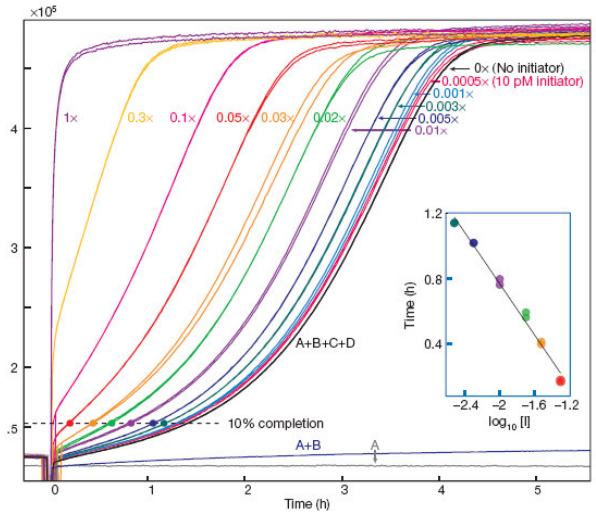
Chemistry describes how pairs of molecules react to produce other molecules.

# Simulation Results

The simulation curves on the left are the amount of A (quenched, closed) hairpin, so they represents the inverse of the fluorescence experiment on the right, which detects open hairpins. (Therefore I flipped my plot upside down for easy comparison.)



Gillespie simulation of the strand algebra system

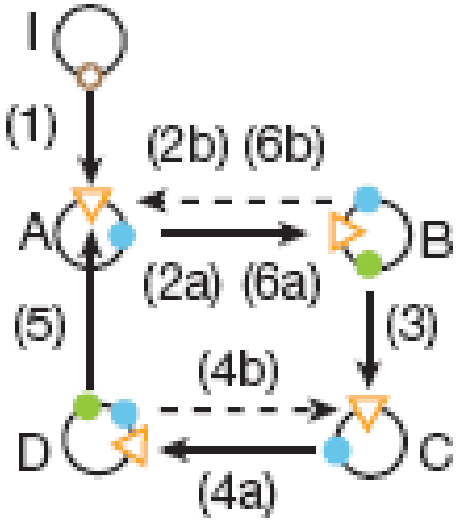



From Yin et al.

# Simulation Details

In this SPiM simulator code we vary the amount of initiator I() at the bottom, for values between 50 and 3000:

```
directive plot !ia; Gate1(ia,ab); Gate2(ab,ia,bc); Gate1(bc,cd);
Gate2(cd,bc,ia)
directive sample 0.004 1000

let Initiator(out1:chan) = !out1
let Gate1(in1:chan, out1:chan) = ?in1; !out1
let Gate2(in1:chan, out1:chan, out2:chan) = ?in1; (!out1 | !out2)

new ia@1.0:chan new ab@1.0:chan
new bc@1.0:chan new cd@1.0:chan

let I() = Initiator(ia)
let A() = Gate1(ia, ab)
let B() = Gate2(ab, ia, bc)
let C() = Gate1(bc, cd)
let D() = Gate2(cd, bc, ia)

run 50 of I()
run 2000 of (A() | B() | C() | D())
```



# Conclusions

# Conclusion

- History of computing
  - Is pointing towards increasing amounts of concurrency and heterogeneity in man-made systems.
  - Modern computer hardware is going (by necessity) multi-core.
  - Programming such systems is still a major challenge.

- Natural history
  - Massively concurrent and heterogeneous computation.
  - We do not really yet understand how concurrency works there (e.g. gene networks, neural networks).

- Future molecular computing
  - Single-function input-output devices? Individual automata? Reactive systems? Universal computers?
  - Something new: molecular computing can process information *and* build physical structures! There is still much to be done to devise suitable theories of such things.