Automata as Molecules

Implementing Interacting Automata as Biochemical Systems

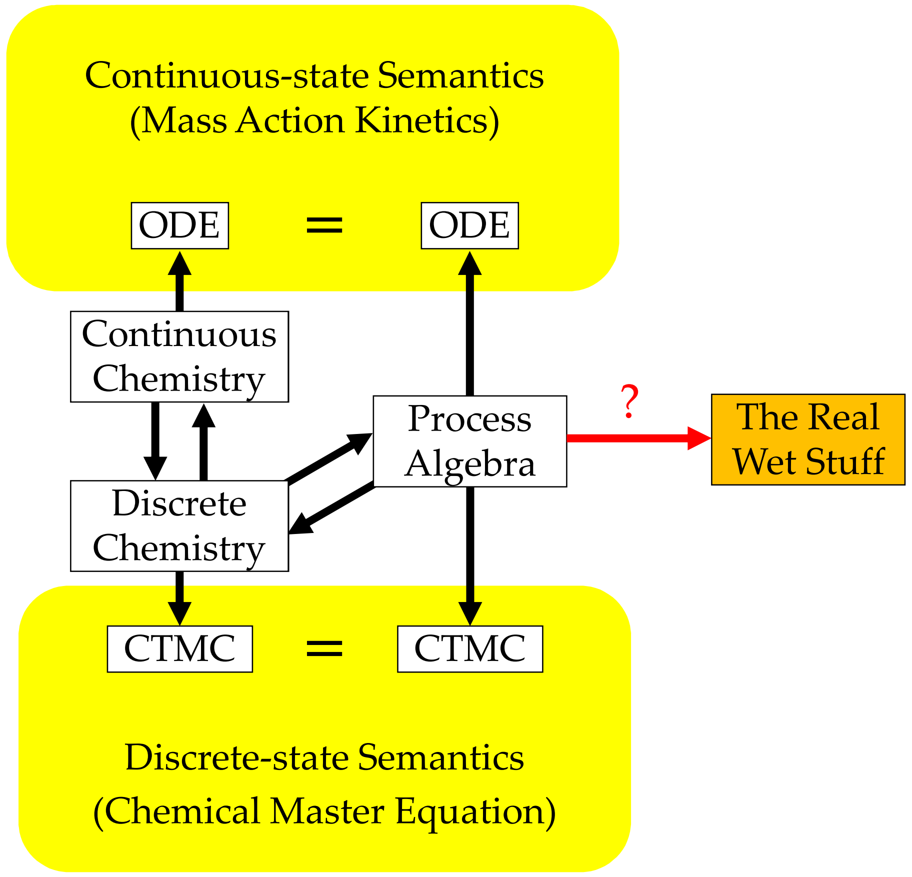
Luca Cardelli

Microsoft Research

Open Lectures for PhD Students in Computer Science
Warsaw 2009-05-07..08

http://lucacardelli.name

Motivation



How do we implement an arbitrary process?

How do we implement an arbitrary chemical system?
(how do we then implement the chemical species?)

Automata to Molecules

- 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 (means 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.

- Direct Compilation of Interacting Automata to DNA.
  - We can more simply go directly from Interacting Automata to DNA.
  - In doing so, we want to preserve the stochastic semantics (rates).

DNA Computing

- Early DNA Computing
  - Demonstrated computation by DNA hybridization [Adelman].
  - Why DNA? Widely available mature technology.
  - Massively concurrent (but still not enough for NP-complete problems).
  - Slow *and* awkward (manual cycling).

- New Focus
  - Not going to compete with Intel in speed (hours … days).
  - But can interface with biological systems!
  - For detection and intervention in live organisms.

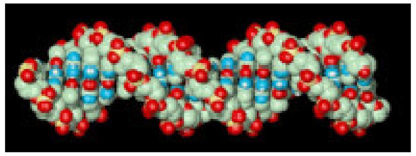
- New Paradigm
  - Autonomous DNA computation (mix-and-go) [Yurke&Mills].
  - Output readout by fluorescence or atomic microscopy, in vitro.
  - Or by influencing cellular mechanisms in vivo [Shapiro survey].

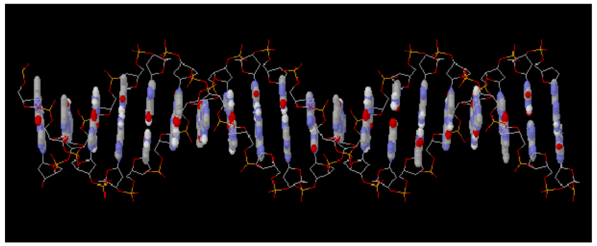
Computation by DNA Strand Displacement

ACGT

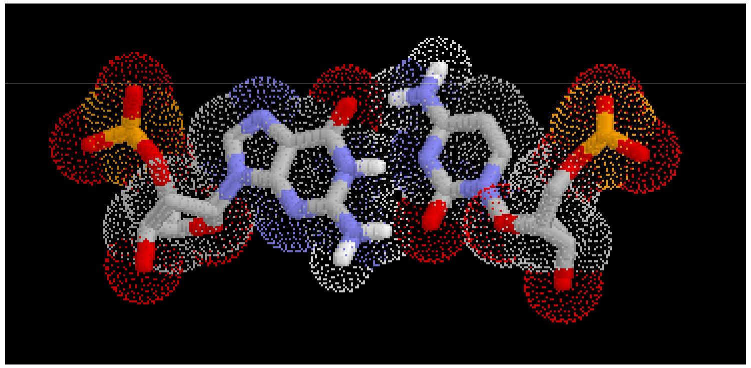
Interactive DNA Tutorial

(http://www.biosciences.bham.ac.uk/labs/minchin/tutorials/dna.html)



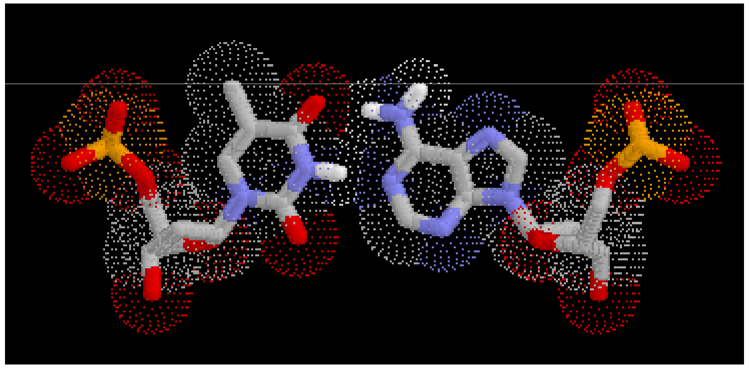


Sequence of Base Pairs



GC Base Pair

Guanine-Cytosine



TA Base Pair

Thymine-Adenine

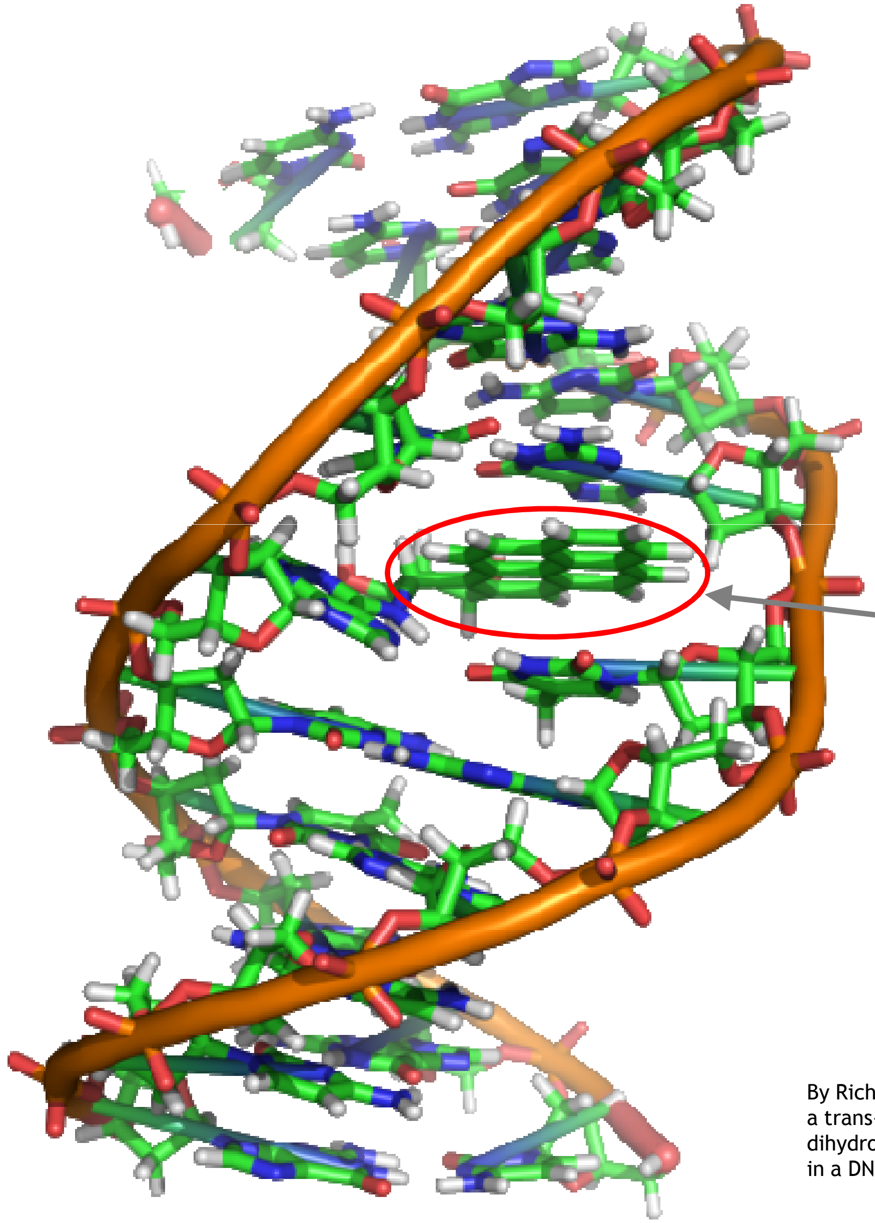
Hence DNA is a string over a 4-letter ACGT alphabet

Human genome : ~3 billion base pairs

= 750 Megabytes (since 1 byte encodes 4 base pairs)

= 1 movie download!

DNA Double Helix



Benzopyrene

(What happens when you smoke)

By Richard Wheeler (Zephyris) 2007. Solution structure of a trans-opened (10S)-dA adduct of +)-(7S,8R,9S,10R)-7,8-dihydroxy-9,10-epoxy-7,8,9,10-tetrahydrobenzo[a]pyrene in a DNA duplex. Wikimedia Commons.

Watson-Crick Duality

G - C
T - A
Affinity

$$\begin{array}{c}\text{G:A:C:T}\\ \overrightarrow{}\\ \overleftarrow{}\\ \text{C:T:G:A}\end{array}$$

Double Strand

$G^{\perp} = C$
$T^{\perp} = A$
Complementarity

$X^{\perp\perp} = X$

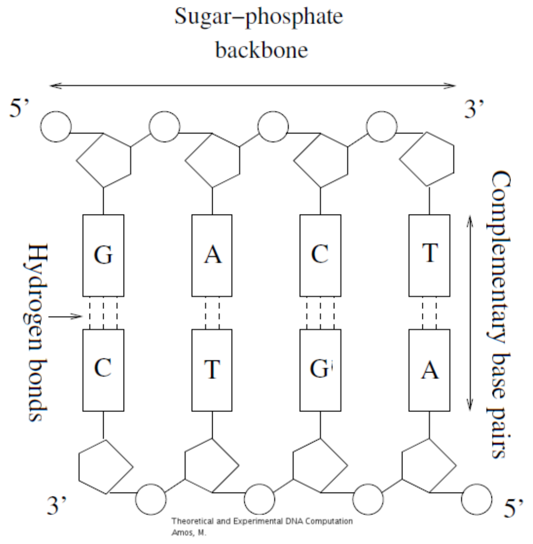
3’ end

$$\underset{\text{C:T:G:A}}{\longleftarrow} = \overset{\text{A:G:T:C}}{\longrightarrow}$$

Equal Single Strands

$$\begin{array}{c}\overset{\text{G:A:C:T}}{\longrightarrow}\\ \perp \\ \underset{\text{C:T:G:A}}{\longleftarrow}\end{array} = \overset{\text{A:G:T:C}}{\longrightarrow}$$

Complementary Single Strands



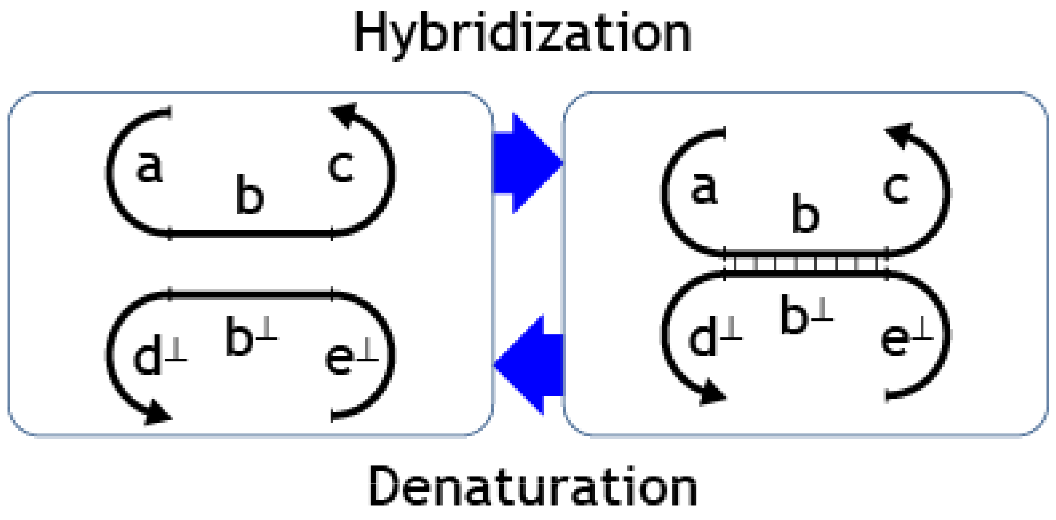
Hence $(\text{G:A:C:T})^{\perp} = \text{A:G:T:C} = T^{\perp}\text{:}C^{\perp}\text{:}A^{\perp}\text{:}G^{\perp}$

all written from 5’ to 3’

$$(X\text{:}Y)^{\perp} = Y^{\perp}\text{:}X^{\perp}$$

Watson-Crick duality
(for any sequences of bases X,Y)

Hybridization

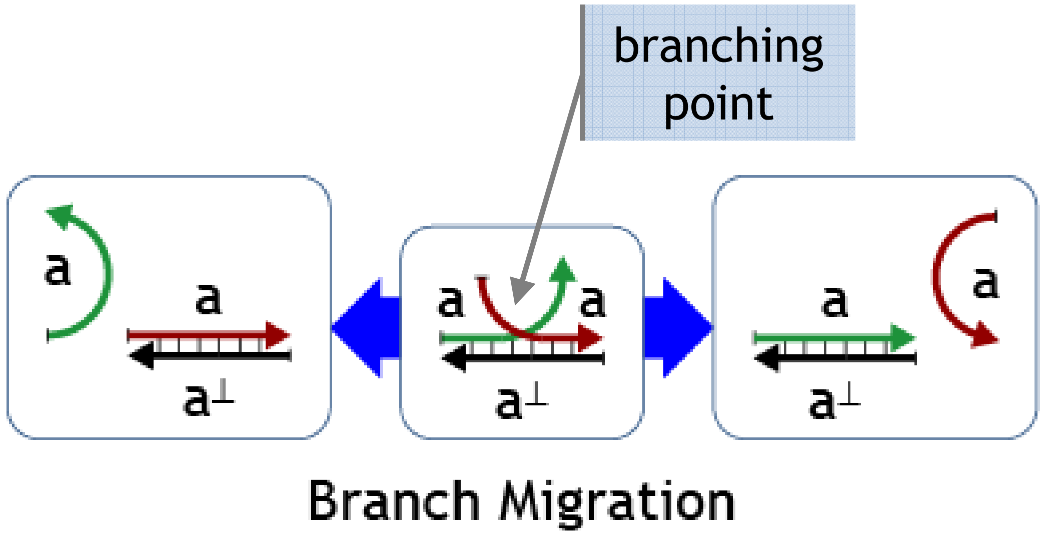


Hybridization is also called annealing; denaturation is also called melting.

The direction of the reaction (or in general the equilibrium between the two states) is determined by a number of factors, e.g. temperature.

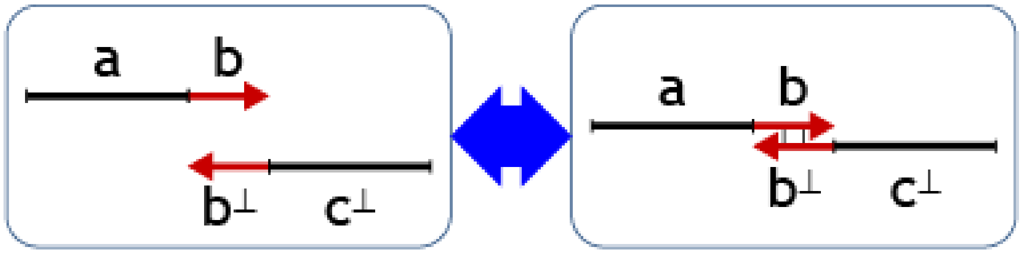
We assume we are in conditions that favor hybridization beyond a certain length of matching region.

Branch Migration

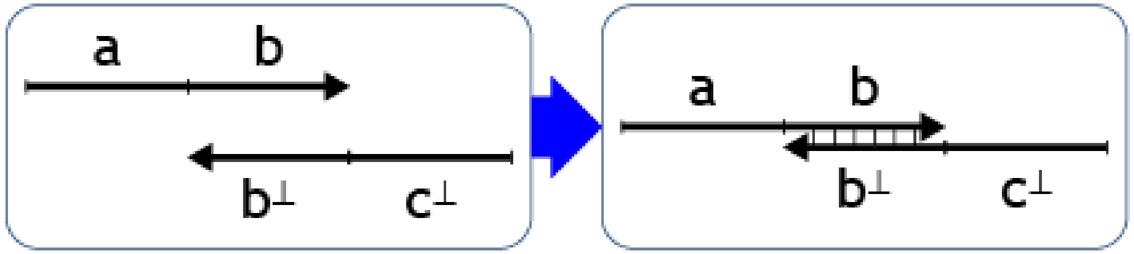


The branching point moves left and right by a random walk. Until it reaches an end point.

Short and Long Segments

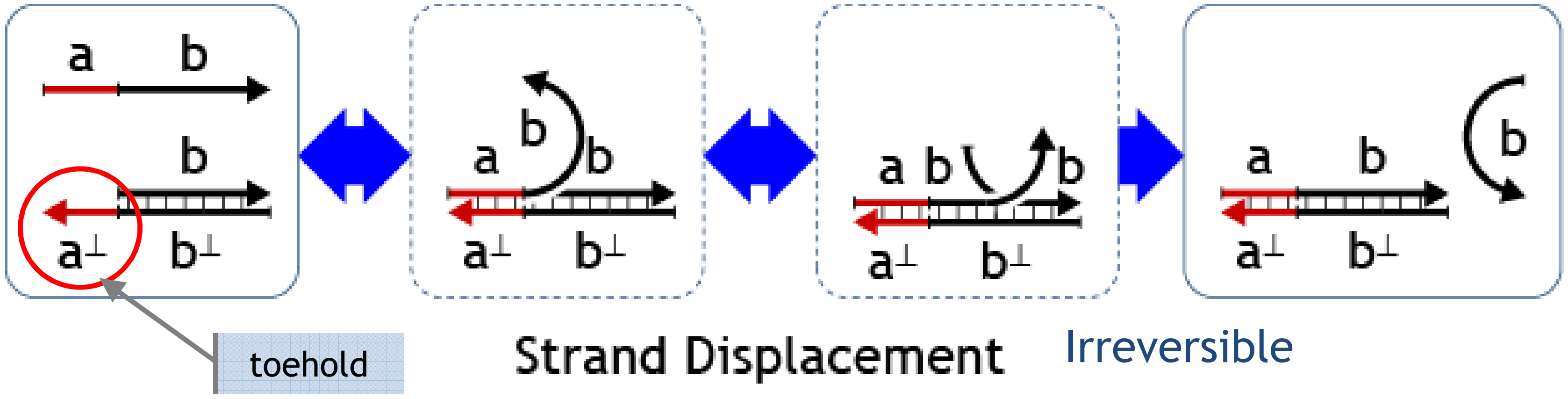


Reversible Binding



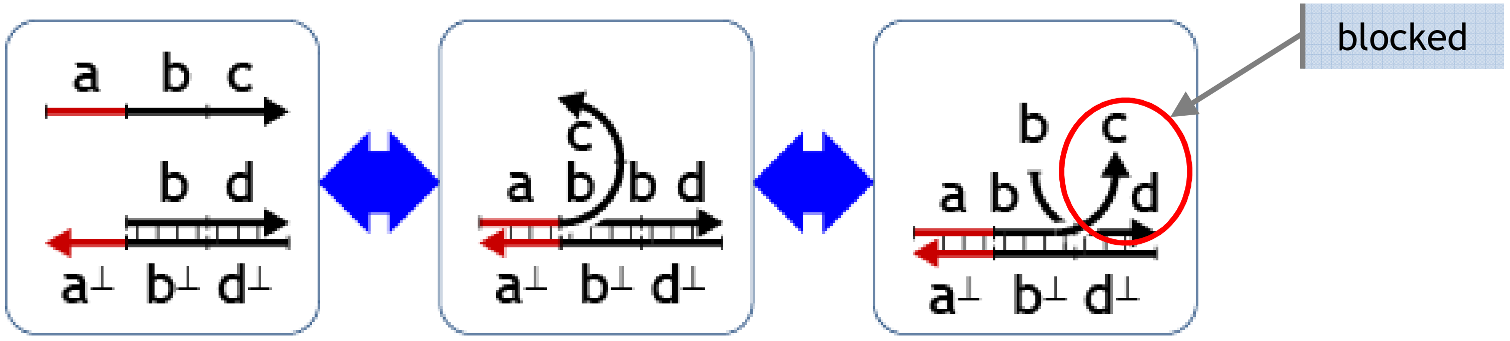
Irreversible Binding

Strand Displacement Reaction



Strand Displacement Irreversible

Partial Match

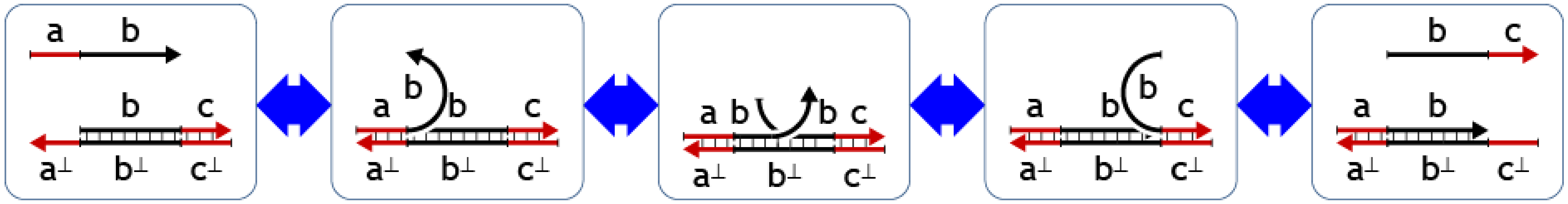


Reversible!

because the random walk is 'reflected' by the blockage

Irreversible match is determined by the toehold plus the branch migration region. That is, the toehold is a *cache* for the full address. The toehold must be short enough to guarantee reversible binding, but the branch migration region is practically unlimited. This means that the address space is unlimited.

Toehold Exchange Reaction



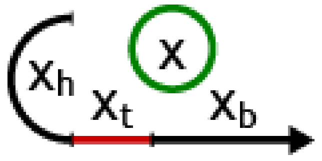
Toehold Exchange

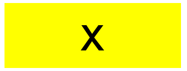
Reversible

Signal Strand

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

(We work with a simpler version of their signal stands.)





x_h = history
x_t = toehold
x_b = binding

The history x_h is not part of signal recognition: strands with different histories should behave the same. Hence, x denotes an equivalence class of strands with different histories.

The combination x_t,x_b identifies the signal x.

If $x \neq y$ then x and $y^\perp$ are not supposed to hybridize.

Signals and Gates

- Signals “x” are always positive strands

- Gates “x.y” always have a negative strand toehold and backbone.
  - that is, the input “x” is implicitly perp’ed
  - and the output “y” is another positive signal

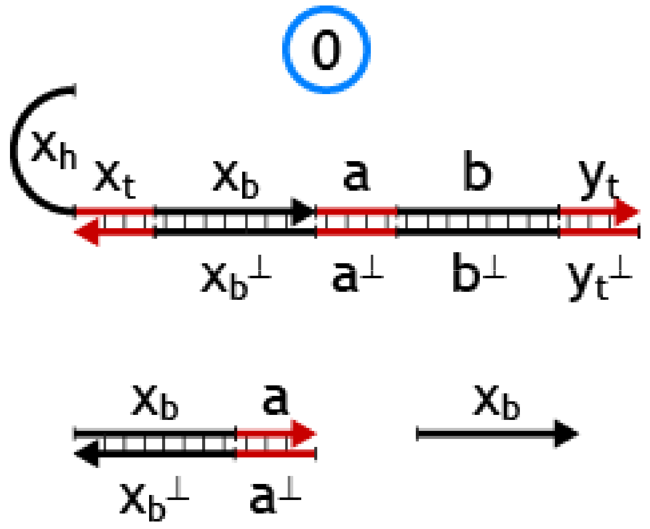
- This separation helps the DNA realization, as one can use 3-letter alphabets (ATC/ATG) for each strand, minimizing secondary structure and entanglement.

- This way, by the way, we appear to give up Turing completeness, which is possible by freely using positive and negative strands. (Turing completeness has been demonstrated in DNA tiling systems.)

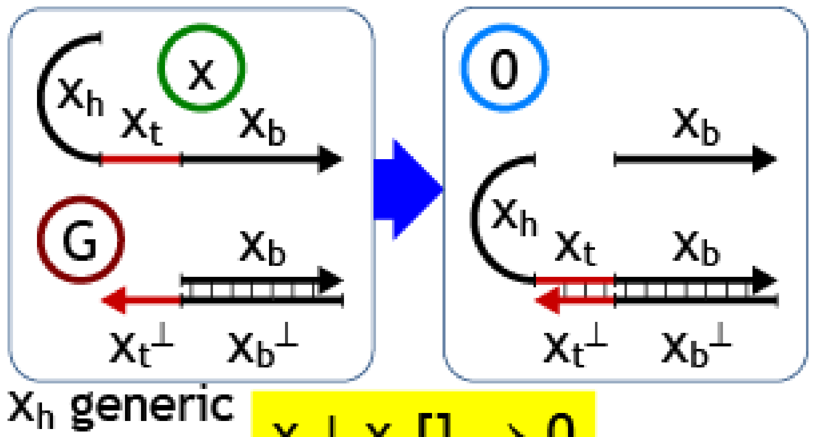
- (It is not clear how to achieve Turing completeness based on this signal/gate structure.)

Inert Systems

A system is considered *inert* (terminated) if it has no free toeholds.



x.[] Annihilator Gate

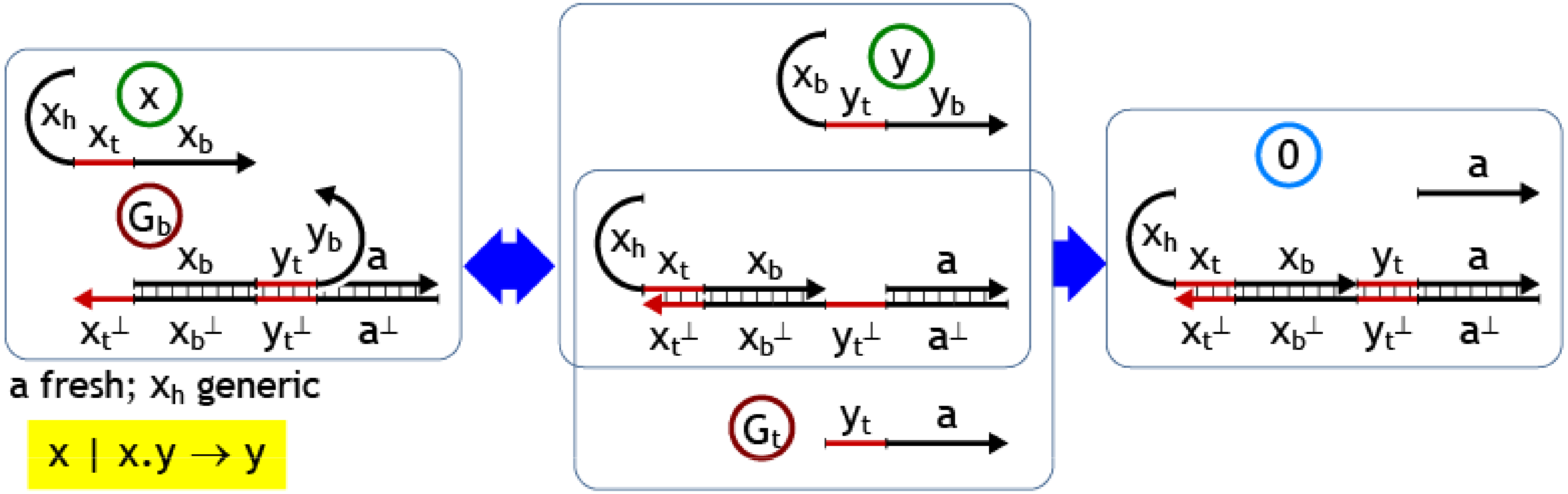


$x \mid x.[] \to 0$

This is just the strand displacement reaction, but seen as a gate absorbing a signal x and producing nothing (0 = inert).

Any history segment that is not determined by the gate structure is said to be ‘generic’ (can be anything).

x.y Transducer Gate



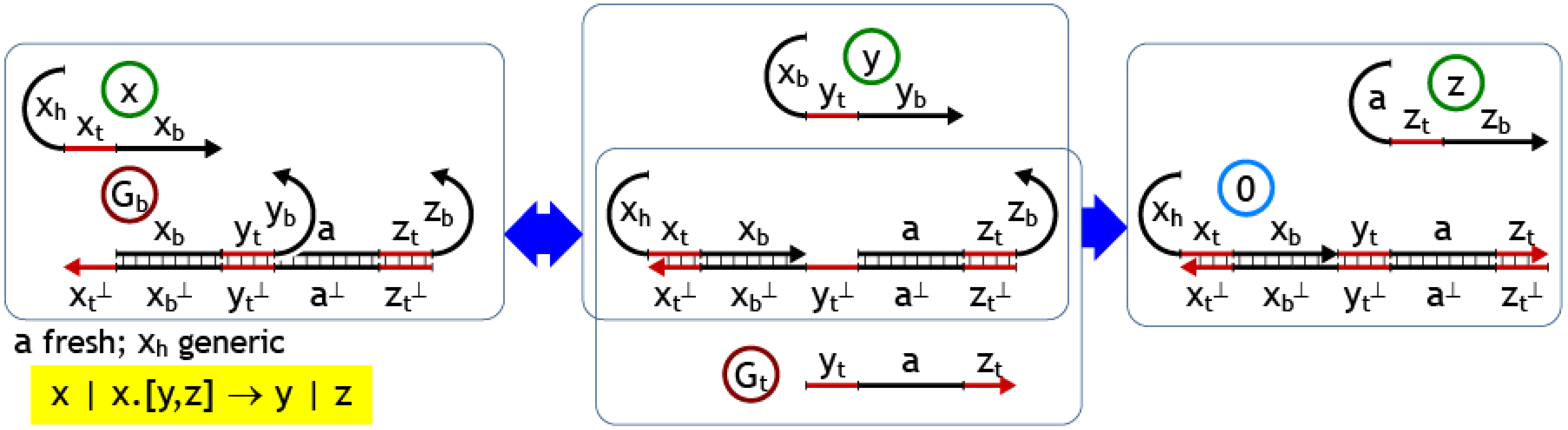
x_h
x_t
x
x_b
G_b
x_b
y_t
y_b
a
$x_t^{\perp}$ $x_b^{\perp}$ $y_t^{\perp}$ $a^{\perp}$
a fresh; x_h generic
$x \mid x.y \rightarrow y$
x_b y_t y y_b
x_h x_t x_b a
$x_t^{\perp}$ $x_b^{\perp}$ $y_t^{\perp}$ $a^{\perp}$
G_t y_t a
0
a
x_h x_t x_b y_t a
$x_t^{\perp}$ $x_b^{\perp}$ $y_t^{\perp}$ $a^{\perp}$

G_b,G_t (gate backbone and trigger) form the transducer.

Any history segment that is not determined by the gate structure is said to be ‘generic’ (can be anything).

Any gate segment that is not a non-history segment of an input or output signal is taken to be ‘fresh’ (globally unique for the gate), to avoid possible interferences.

x.[y,z] Fork Gate

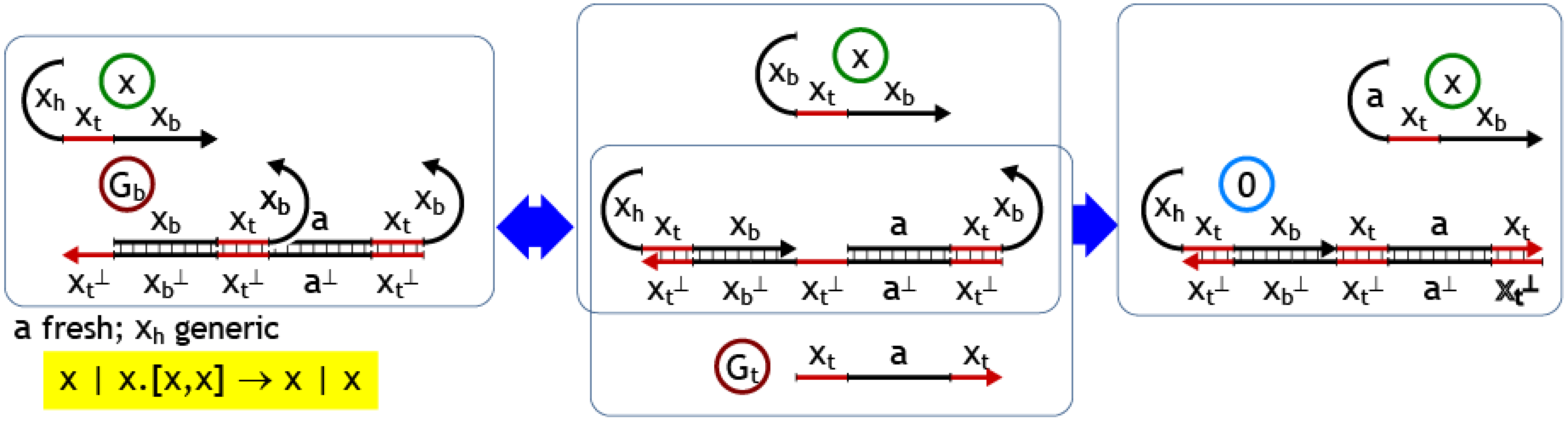


G_b,G_t (gate backbone and trigger) form the transducer.

Any history segment that is not determined by the gate structure is said to be ‘generic’ (can be anything).

Any gate segment that is not a non-history segment of an input or output signal is taken to be ‘fresh’ (globally unique for the gate), to avoid possible interferences.

x.[x,x] Fork Gate



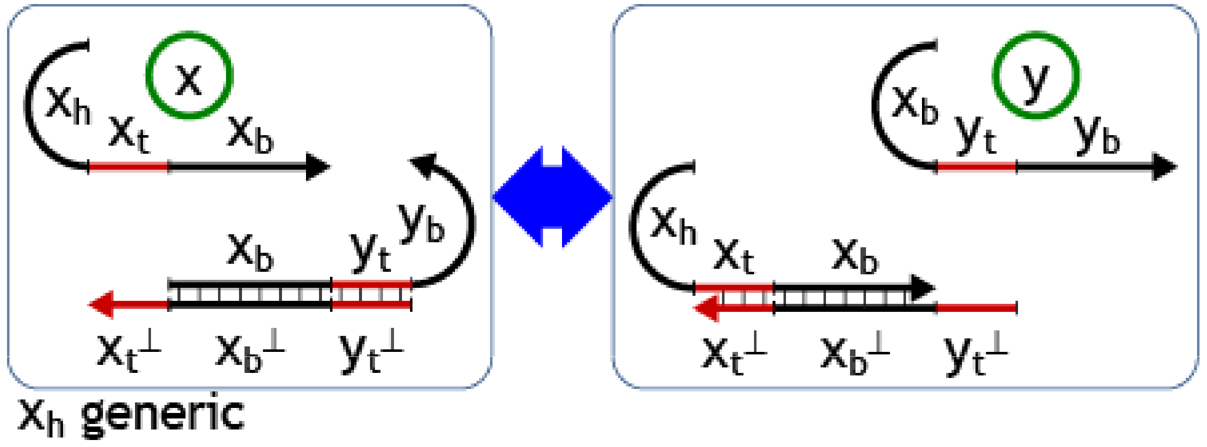
a fresh; x_h generic

$x \mid x.[x,x] \to x \mid x$

Autocatalyst

Adapter (a non-Gate)

Consider the reversible 'first half' of the transducer,
which works by toehold exchange:

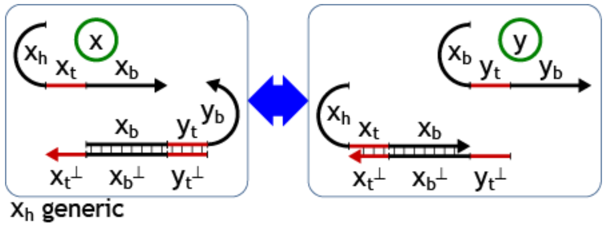


This has an interesting function: it adapts an x signal to a y signal:

> if x is present then both x and y are available by reversibility
> if y is consumed, then x is consumed.

If a gate produces an x and another gate expects a y, then we can (perhaps) use an adapter to connect them. Note that a full x to y transducer would not work as well as an adapter, because it would always remove x even if nobody wants y.

Non-gates



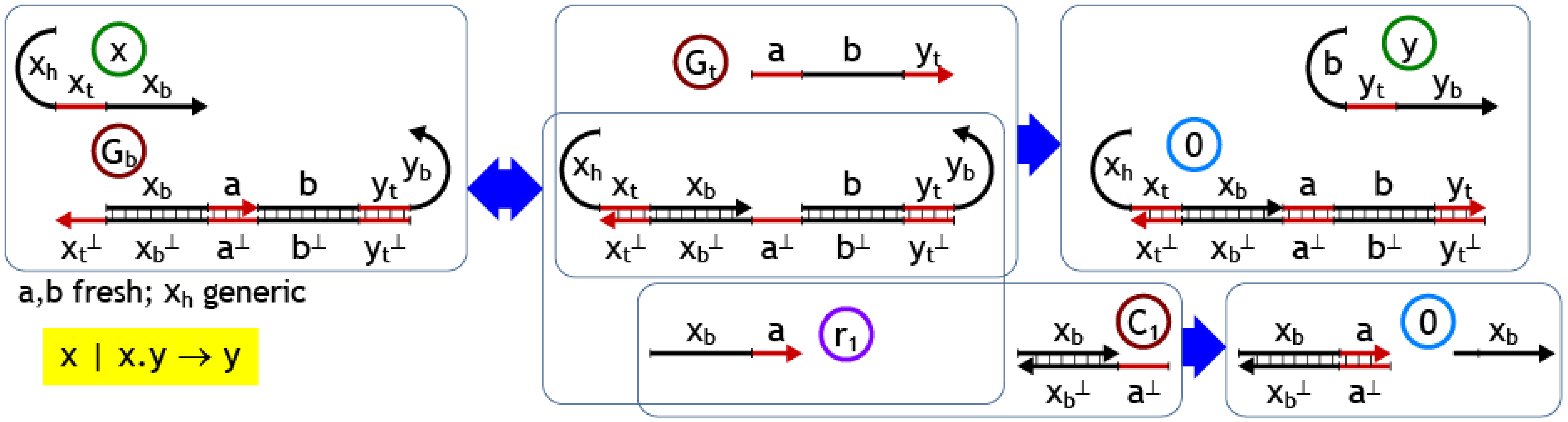
- However, the adapter is *not* a gate:
  - The inverse reaction works only for y's with x_b history. Gates must work on equivalence classes of signals, for any history. There is in fact no way to write the adapter as a gate reaction: x | adapt(x,y) $\rightarrow$ y | ?.
  - When y is consumed it leaves behind a non-inert components which eventually reduces the availability of x by speeding up the reverse reaction. These non-inert components should be removed.
  - Since both x and y are public signals, there is a possibility that some other part of the system may produce a y signal with x_b history, interfering with this adapter (slowing it down, and removing y's from somewhere else).
  - A proper adapter gate is instead (x.y | y.x), assuming a population of them.

- Although not a gate, an adapter can be used as part of a larger proper gate, like the Transducer, which:
  - works on equivalence classes of signals
  - does not leave active garbage around
  - but still admits interference on y (a alternative transducer is coming up).

Another Architecture

- We now start working with a slightly different gate structure.
  - The order of Trigger and Output is swapped.
- This is slightly more complex.
  - It requires a 'garbage collection' step.
- But it generalizes better to more complex gates.
  - Removes the worry about interference on x_b:y_t.
  - Join gates require garbage collection anyway.
- This results in a uniform structure for *all* gates.

x.y Transducer Gate

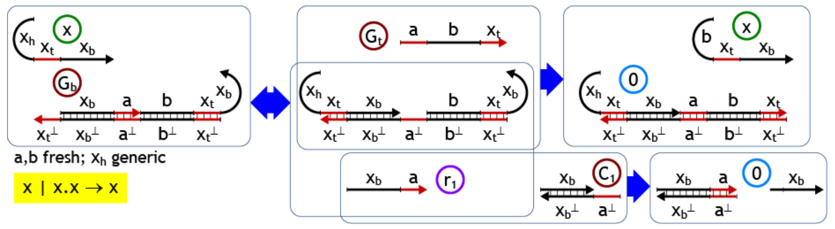


G_b,G_t,C_1 (gate backbone, trigger, collector) form the transducer.

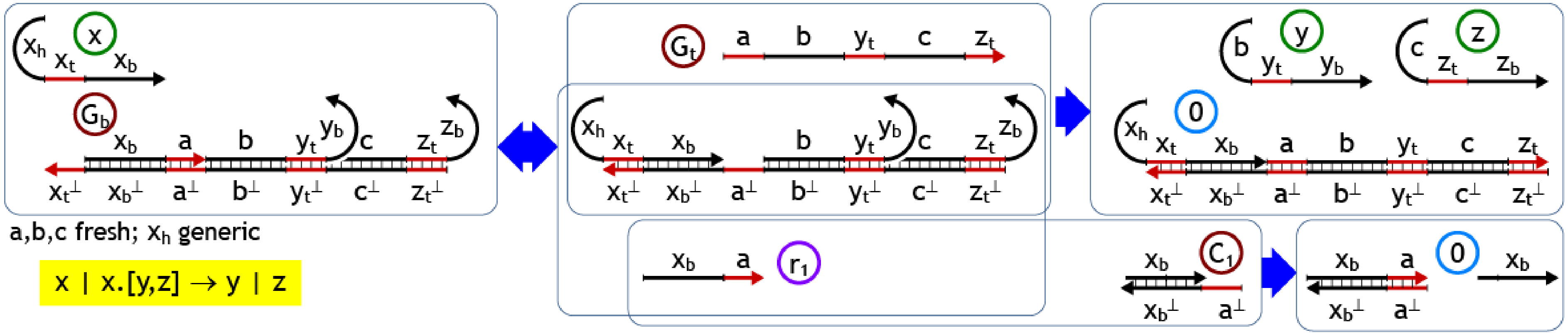
We need to collect the x_b:a strand to end up with an inert system.

If we do not collect it, it accumulates and *slows down* further transductions by pushing the reversible reaction to the left.

No problem with x.x:



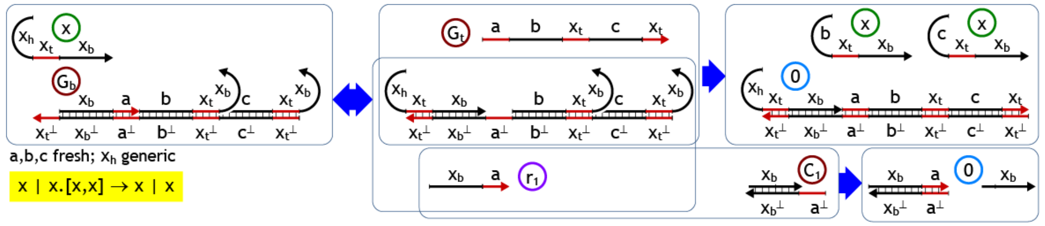
x.[y,z] Fork Gate



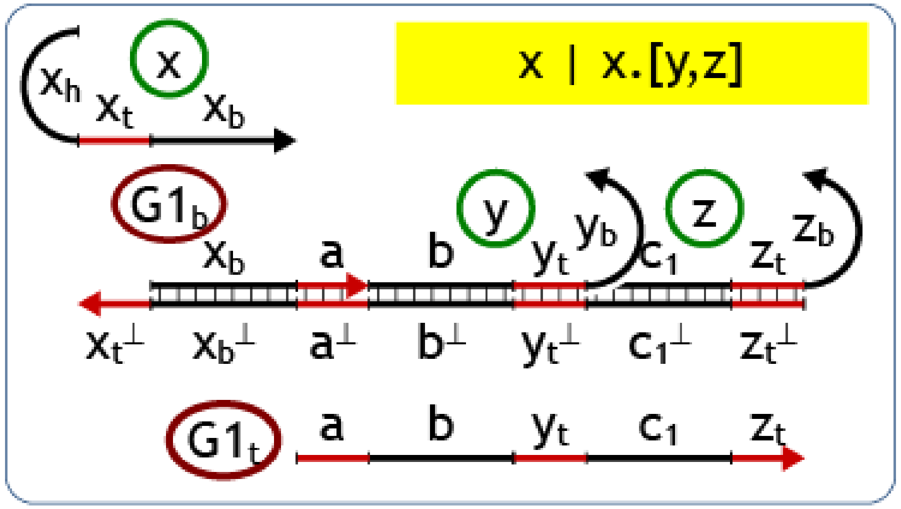
The triggering is now more uniform: all the outputs are released together.

This fork structure (although slightly more complex than the earlier fork) generalizes smoothly to multiple *inputs* as well, because in that case we cannot avoid a garbage collection phase.

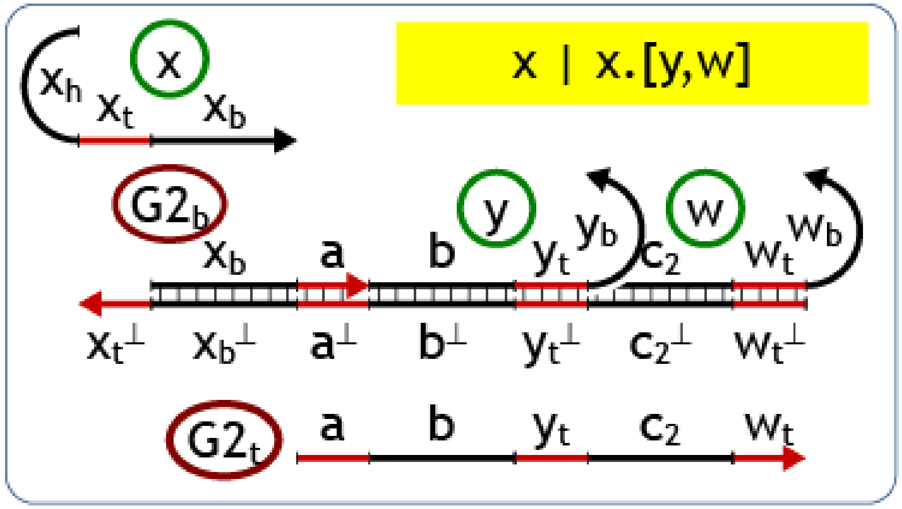
No problem with x.[x,x]:



Exercise 3: x.[y,z] | x.[y,w] Interference



~~a,b~~,c_1 fresh; x_h generic



~~a,b~~,c_2 fresh; x_h generic

- Suppose we 'forgot' to take a,b fresh, so they are shared by the two gates. Something goes horribly wrong from these initial conditions:

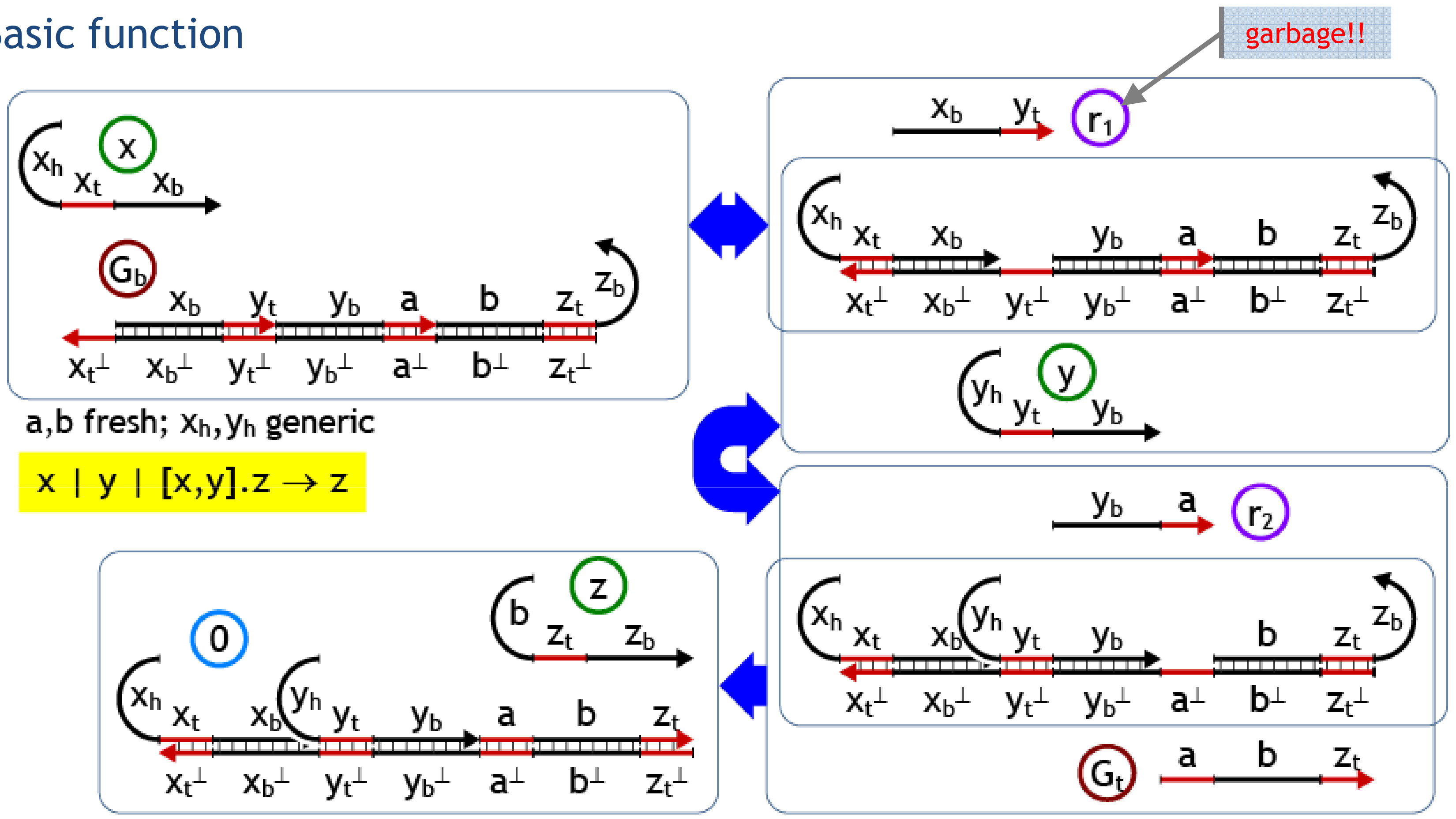
  x | x.[y,z] | x | x.[y,w]

  where x.[y,z] = $G1_b$,$G1_t$ and x.[y,w] = $G2_b$,$G2_t$

- What goes wrong?

[x,y].z Join Gate (function)

Basic function

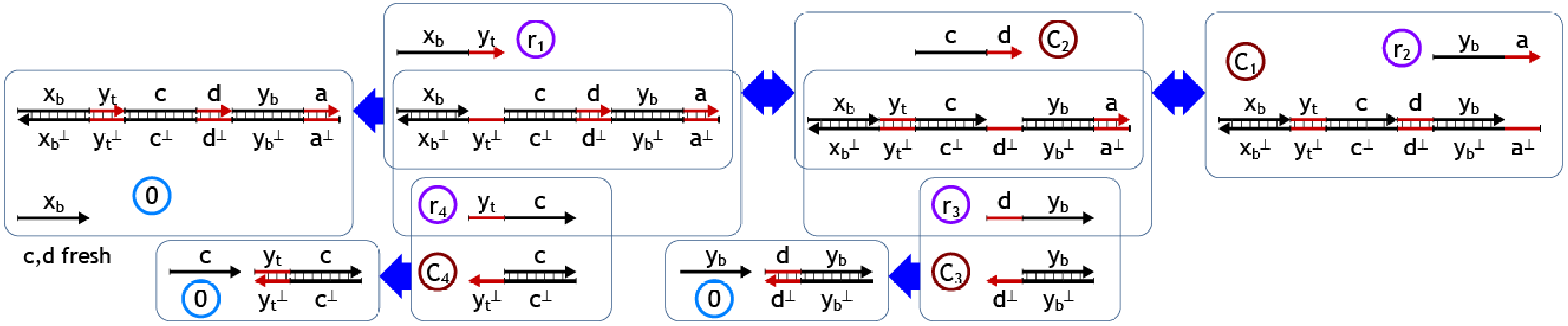


garbage!!
a,b fresh; x_h, y_h generic
$x \mid y \mid [x,y].z \rightarrow z$

Join can be implemented by a ‘reversible-AND gate’ taking two sequential inputs where the first one is reversible (Soloveichik Fig.3), so that x is not actually absorbed until y is found. The ‘garbage’ r_1 must not be collected until y is found: this is signaled by the release of r_2.

[x,y].z Join Gate (collection)

Garbage Collection

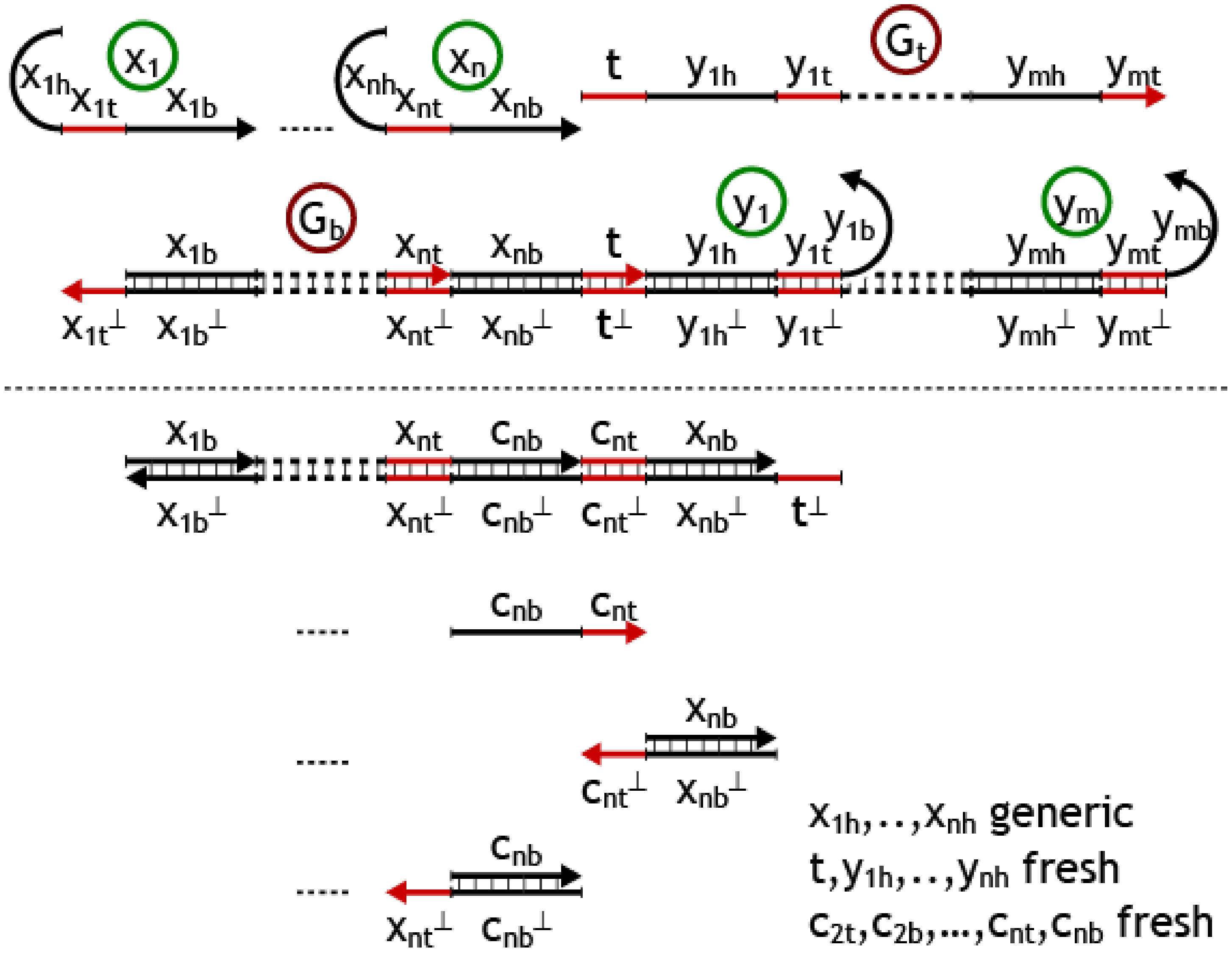


Garbage collection of r_1 is needed for join to work well. This is done by another reversible-AND between r_1 and r_2, triggered by the release of r_2. This second reversible-AND leaves garbage too (r_3, r_4), but this can be collected immediately, as we know by construction that both inputs r_1,r_2 are available and we need not wait to revert their bindings.

The extra intermediate c,d segments separate the r_1 binding from the r_2 binding. Without them, a segment y_t:y_b (instead of y_t:c and d:y_b) would be released: that is y!

$[x_1,..,x_n].[y_1,..,y_m]$ General Join/Fork Gate

$x_1 \mid .. \mid x_n \mid [x_1,..,x_n].[y_1,..,y_m] \rightarrow y_1 \mid .. \mid y_m$

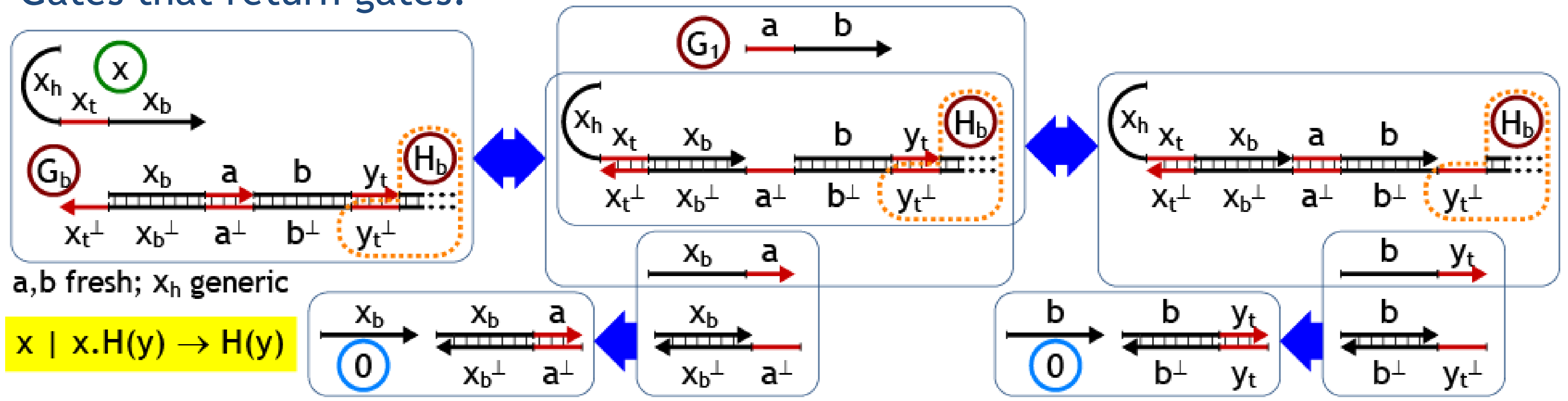


x_1
x_{1h} x_{1t} x_{1b}
x_n
x_{nh} x_{nt} x_{nb}
G_t
t y_{1h} y_{1t} y_{mh} y_{mt}
G_b
x_{1b} x_{nt} x_{nb} t
y_1 y_{1h} y_{1t} y_{1b}
y_m y_{mh} y_{mt} y_{mb}
$x_{1t}^\perp$ $x_{1b}^\perp$ $x_{nt}^\perp$ $x_{nb}^\perp$ $t^\perp$ $y_{1h}^\perp$ $y_{1t}^\perp$ $y_{mh}^\perp$ $y_{mt}^\perp$
x_{1b} x_{nt} c_{nb} c_{nt} x_{nb}
$x_{1b}^\perp$ $x_{nt}^\perp$ $c_{nb}^\perp$ $c_{nt}^\perp$ $x_{nb}^\perp$ $t^\perp$
c_{nb} c_{nt}
x_{nb}
$c_{nt}^\perp$ $x_{nb}^\perp$
c_{nb}
$x_{nt}^\perp$ $c_{nb}^\perp$

$x_{1h},..,x_{nh}$ generic
$t,y_{1h},..,y_{nh}$ fresh
$c_{2t},c_{2b},...,c_{nt},c_{nb}$ fresh

x.H(y) Curried Gates

Gates that return gates:

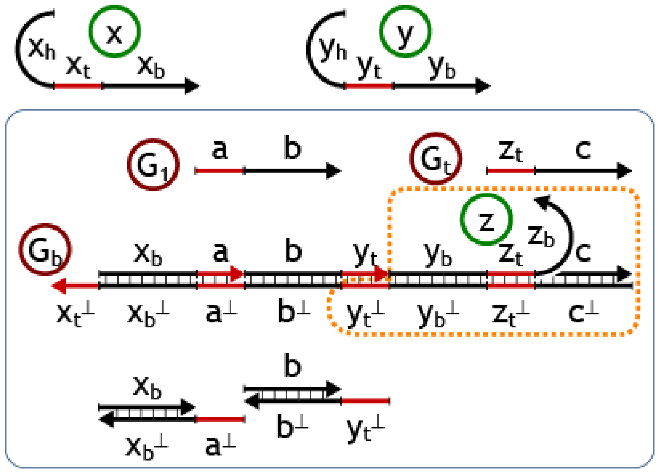


x_h x_t x x_b
G_b x_b a b y_t H_b
$x_t^\perp$ $x_b^\perp$ $a^\perp$ $b^\perp$ $y_t^\perp$
G_1 a b
x_h x_t x_b b y_t H_b
$x_t^\perp$ $x_b^\perp$ $a^\perp$ $b^\perp$ $y_t^\perp$
x_h x_t x_b a b H_b
$x_t^\perp$ $x_b^\perp$ $a^\perp$ $b^\perp$ $y_t^\perp$
x_b a
x_b $x_b^\perp$ $a^\perp$
x_b 0 x_b a $x_b^\perp$ $a^\perp$
b y_t
b $b^\perp$ $y_t^\perp$
b 0 b y_t $b^\perp$ y_t

a,b fresh; x_h generic

$x \mid x.H(y) \rightarrow H(y)$

For example, x.y.z:



x_h x_t x x_b
y_h y_t y y_b
G_1 a b
G_t z_t c
z z_b
G_b x_b a b y_t y_b z_t c
$x_t^\perp$ $x_b^\perp$ $a^\perp$ $b^\perp$ $y_t^\perp$ $y_b^\perp$ $z_t^\perp$ $c^\perp$
x_b b
$x_b^\perp$ $a^\perp$ $b^\perp$ $y_t^\perp$

a,b,c fresh; x_h,y_h generic

$x \mid x.y.z \rightarrow y.z$

This means we can have gates of the form:

$$G \ ::= \ [x_1,..,x_n].[x'_1,..,x'_m] \ \vdots \ [x_1,..,x_n].G$$

$n\geq 1,\ m\geq 0$

Exercise 4: x.y.z | [x,y].w Interference

[David Soloveichik]

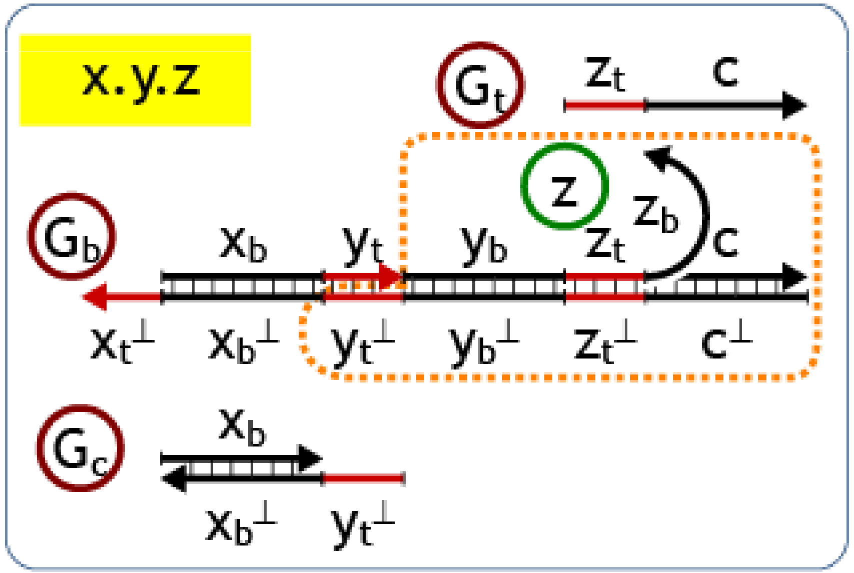
Consider curried gates without the a,b segments (example below): instead of releasing x_b,a and b,y_t segments, they would release x_b, y_t.

But that is exactly the strand r_1 of an [x,y].w gate: the strand that reverts the x input. This definitely causes an interference between x.y.z and [x,y].w.

Find a situation where the presence (x.y.z as below) or absence (x.y.z as in previous slide) of this interference causes different outcomes.

Hint: it changes outcome *probability*.

Note: the a,b segments prevent the interference.



c fresh; x_h, y_h generic
(without the a,b segments)

Summary

- DNA strand displacement technology provides a way of implementing abstract signal transducer networks.
- Fork gates and Join gates are the main components.
- How powerful it this style of computation?

Combinatorial Strand Algebra

Formalizing a Process Algebra

- A term syntax (almost always including parallel composition):
  - $P ::= \ldots \; ⋮ \; P|P \; ⋮ \; \ldots$
- A structural congruence relation (chemical mixing):
  - $P \equiv Q$ (e.g. commutative monoid rules of '|')
- A reduction relation (chemical reactions):
  - $P \to Q$
- Standard rules to connect the two, which have a 'chemical' meaning:
  - 'Dilution': $P \to Q \;\Rightarrow\; P \mid R \to Q \mid R$
  - 'Well-mixing': $P \equiv P', P' \to Q', Q' \equiv Q \;\Rightarrow\; P \to Q$
- Various equivalences (e.g. bisimulation) derived from the above.
- Algebraic laws proved (not taken as axioms) from the equivalences.

Strand Algebra $\mathcal{P}$

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

Here x is a *signal*, 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*
$x_1.x_2$	$\overset{\text{def}}{\equiv} [x_1].[x_2]$	is a *sequence gate*
$x.[x_1,..,x_m]$	$\overset{\text{def}}{\equiv} [x].[x_1,..,x_n]$	is a *fork gate*
$[x_1,..,x_n].x$	$\overset{\text{def}}{\equiv} [x_1,..,x_n].[x]$	is a *join gate*
0	is *inert*	
P|P	is *parallel composition* of signals and gates	
P*	is a *population* (multiset) of signals and gates	

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

Structural Congruence for $\mathcal{P}$

$P \equiv P$	equivalence
$P \equiv P' \;\Rightarrow\; P' \equiv P$	
$P \equiv P',\ P' \equiv P'' \;\Rightarrow\; P \equiv P''$	
$P \equiv P' \;\Rightarrow\; P \mid P'' \equiv P' \mid P''$	congruence
$P \equiv P' \;\Rightarrow\; P^* \equiv P'^*$	
$P \mid 0 \equiv P$	diffusion
$P \mid P' \equiv P' \mid P$	
$P \mid (P' \mid P'') \equiv (P \mid P') \mid P''$	
$P^* \equiv P^* \mid P$	population
$0^* \equiv 0$	
$(P \mid P')^* \equiv P^* \mid P'^*$	
$P^{**} \equiv P^*$	

Reduction for $\mathcal{P}$

$x_1 \mid .. \mid x_n \mid [x_1,..,x_n].[x_1',..,x'_m] \;\rightarrow\; x'_1 \mid .. \mid x'_m$ Gate

$P \rightarrow P' \;\Rightarrow\; P \mid P'' \rightarrow P' \mid P''$ Dilution

$P \equiv P_1,\ P_1 \rightarrow P_2,\ P_2 \equiv P' \;\Rightarrow\; P \rightarrow P'$ Well Mixing

Technically, the fully parenthesized Gate rule is:

$x_1 \mid (.. \mid (x_n \mid [x_1,..,x_n].[x'_1,..,x'_m])..) \;\rightarrow\; x'_1 \mid (.. \mid (x'_m)..)$

but we have structural congruence to reassociate.

Examples

$x_1 \mid x_1.x_2 \quad \rightarrow \quad x_2$

$x_1 \mid x_1.x_2 \mid x_2.x_3 \quad \rightarrow\rightarrow \quad x_3$

$x_1 \mid x_2 \mid [x_1,x_2].x_3 \quad \rightarrow \quad x_3$

$x_1 \mid x_1.x_2 \mid x_1.x_3 \quad \rightarrow \quad x_2 \mid x_1.x_3$

and also $\quad \rightarrow \quad x_3 \mid x_1.x_2$

$x_1 \mid x_2 \mid x_3 \mid [x_1,x_2].x_4 \mid [x_1,x_3].x_5 \quad \rightarrow \quad x_3 \mid x_4 \mid [x_1,x_3].x_5$

and also $\rightarrow \quad x_2 \mid [x_1,x_2].x_4 \mid x_5$

$X \mid ([X,x_1].[x_2,X])^*$

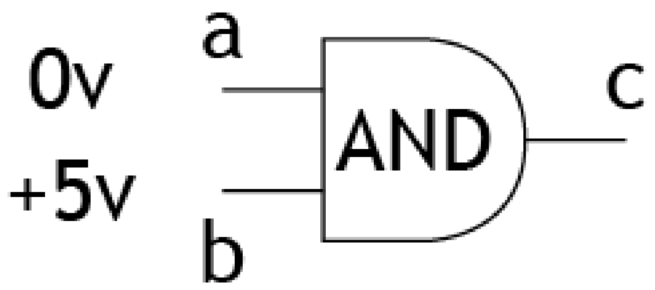
a catalytic system ready to transform multiple x_1 to x_2, with catalyst X

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
  - Petri Nets
  - Finite State Automata
  - Finite Stochastic Reaction Networks (Chemistry) [Soloveichik et al.]
  - Interacting Automata
  - π-calculus (no, not in the current strand algebra)
- And also
  - Higher-level strand algebras, which may form more convenient intermediate languages.
  - Such as the *Nested Strand Algebra* (with nested expressions).

Exercise 5: Boolean Networks

Boolean Networks to Strand Algebra



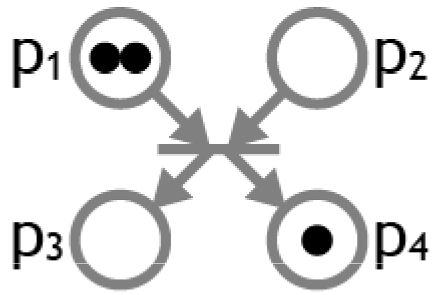
Find an encoding of Boolean networks in Strand Algebra.

It's enough to show how to encode and AND gate that takes Boolean signals on a,b wires and produces a Boolean signal on the c wire.

Petri Nets

Petri Nets to Strand Algebra

Transitions as Gates
Markings as Signals

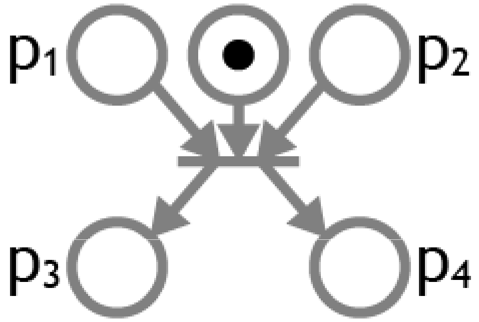


$([p_1,p_2].[p_3,p_4])^* \mid$
$p_1 \mid p_1 \mid p_4$

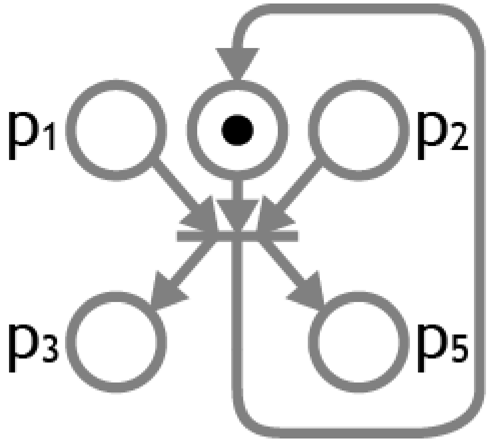
Strand Algebra to Petri Nets

Gates as Transitions

$[p_1,p_2].[p_3,p_4]$

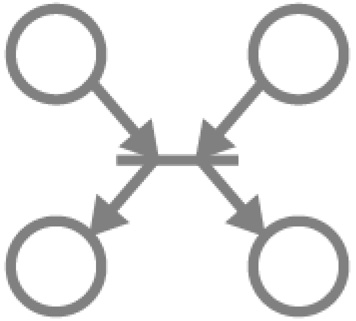


$([p_1,p_2].[p_3,p_4])^*$



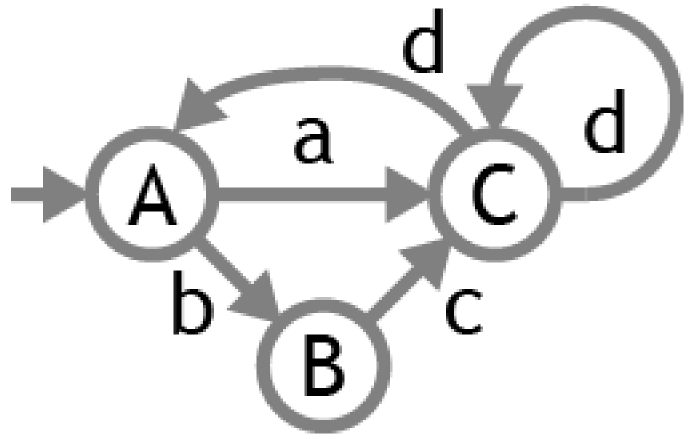
or

$([p_1,p_2].[p_3,p_4])^*$



Finite State Automata

FSA to Strand Algebra



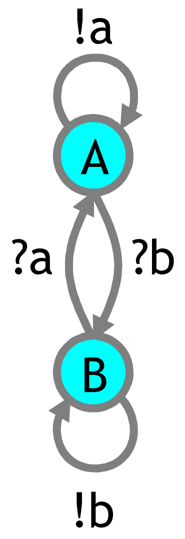
$([A,a].[C,\tau])^*$ |
$([A,b].[B,\tau])^*$ |
$([B,c].[C,\tau])^*$ |
$([C,d].[C,\tau])^*$ |
$([C,d].[A,\tau])^*$ |
$A \mid \tau$

Input strings

a,b,c,d

$\tau\ .[a,\sigma_1]$ |
$[\sigma_1,\tau].[b,\sigma_2]$ |
$[\sigma_2,\tau].[c,\sigma_3]$ |
$[\sigma_3,\tau].\ d$

Interacting Automata (first try)



Interacting Automata to Strand Algebra

Groupies	*Strand*(Groupies)
A = !a;A ⊕ ?b;B	on a: ([B,A].[A,A])* |
B = !b;B ⊕ ?a;A	on b: ([A,B].[B,B])*

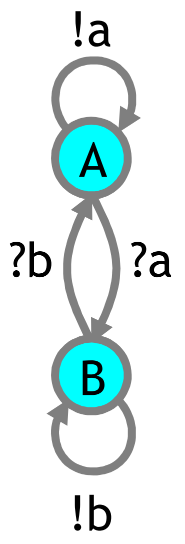
Map each possible interaction to a Join

Strand(E) = *Parallel*(

⟪ (X.[P])* *s.t.* ∃i. E.X.i = τ;P ⟫ ∪

⟪ ([X,Y].[P,Q])* *s.t.* ∃i,j,c. E.X.i = ?c;P and E.Y.j = !c;Q ⟫)

⟪…⟫ means multisets



Celebrities	*Strand*(Celebrities)
A = !a;A ⊕ ?a;B	on a: ([A,A].[B,A])* |
B = !b;B ⊕ ?b;A	on b: ([B,B].[A,B])*

However, Interacting Automata are stochastic!

Summary

- DNA strand displacement gates can be formalizes as Strand Algebra.

- Strand Algebra is equivalent place/transition Petri nets, but:
  - Has a compositional syntax.
  - Keeps track of the (DNA) resources that are consumed during computation.

- While not Turing complete, Strand algebra can embed many common and useful formalisms:
  - Boolean Networks
  - Petri Nets
  - Finite State Automata over multisets and over strings
  - Finite State Transducers over multisets, and from string to multisets (not shown) (not clear how to transduce to strings)
  - Interacting Automata, at least qualitatively.