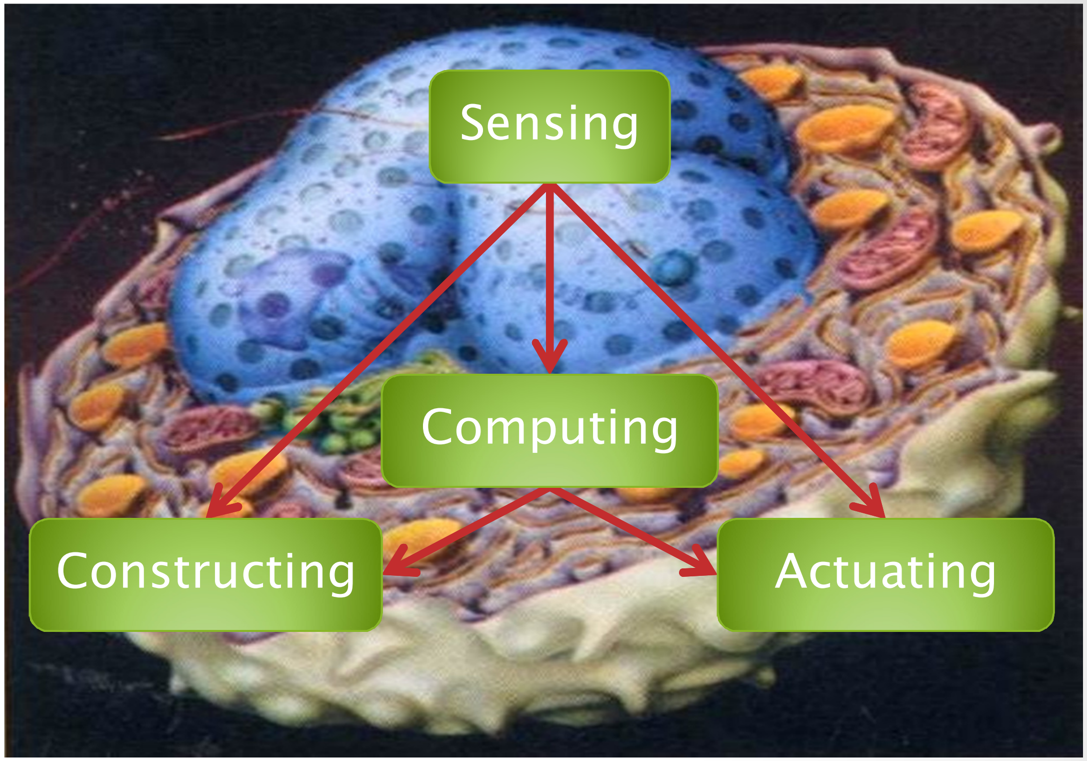
Two-Domain DNA Strand Displacement

Luca Cardelli
Microsoft Research

Dagstuhl, 2010-07-06
http://lucacardelli.name

Nanoscale Engineering

- Sensing
  - Reacting to forces
  - Binding to molecules
- Actuating
  - Releasing molecules
  - Producing forces
- Constructing
  - Chassis
  - Growth
- Computing
  - Signal Processing
  - Decision Making

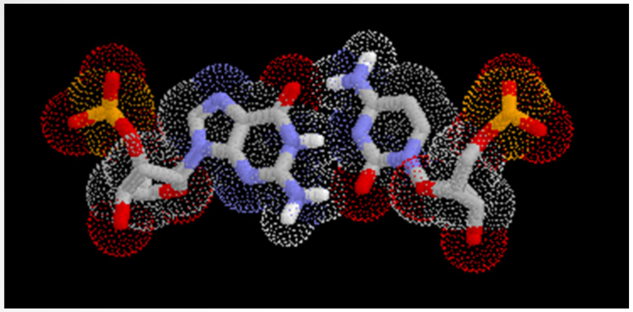


Nucleic Acids can do all this.

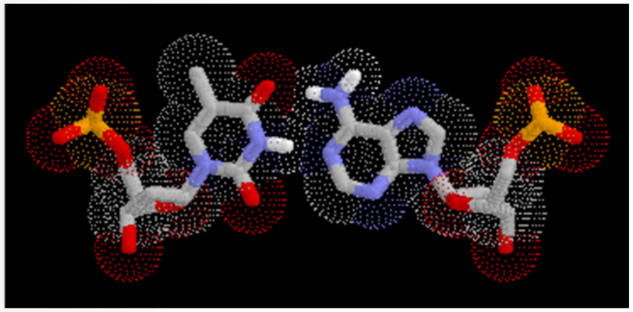
And interface to biology.
And are programmable.

Strand Displacement Basics

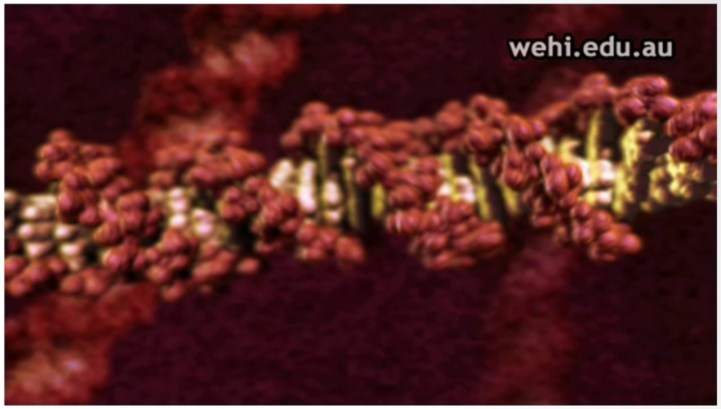
DNA



GC Base Pair
Guanine–Cytosine

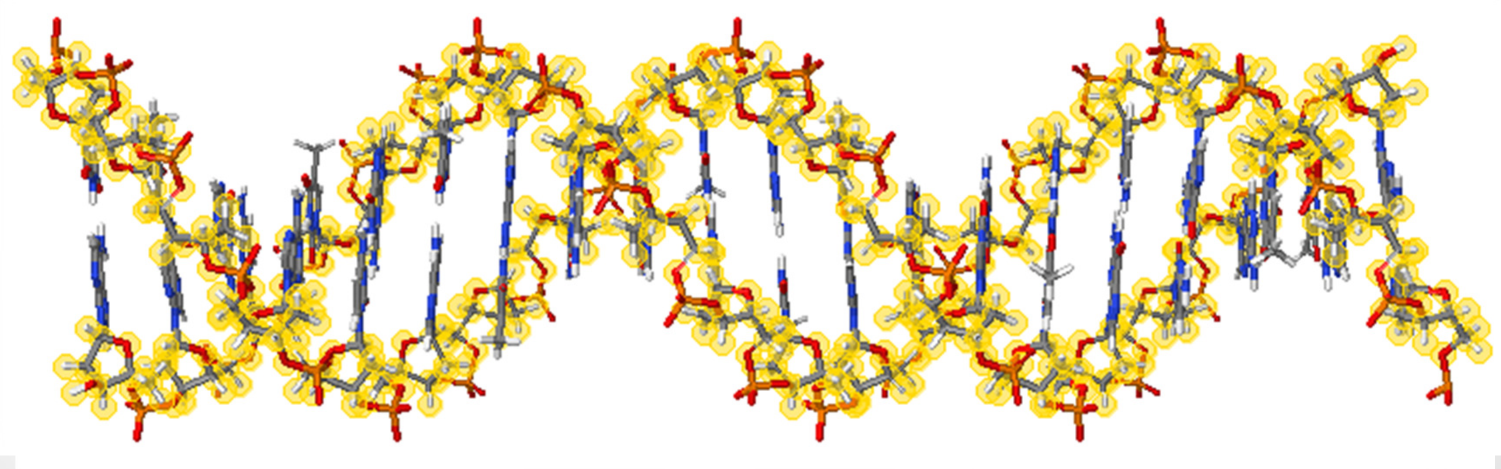


TA Base Pair
Thymine–Adenine



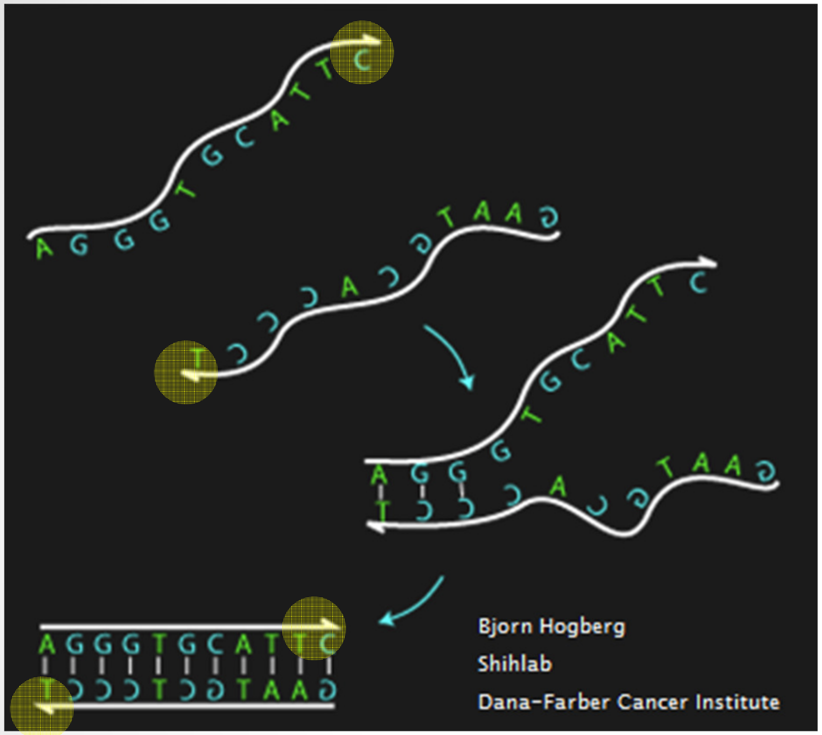
Interactive DNA Tutorial

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

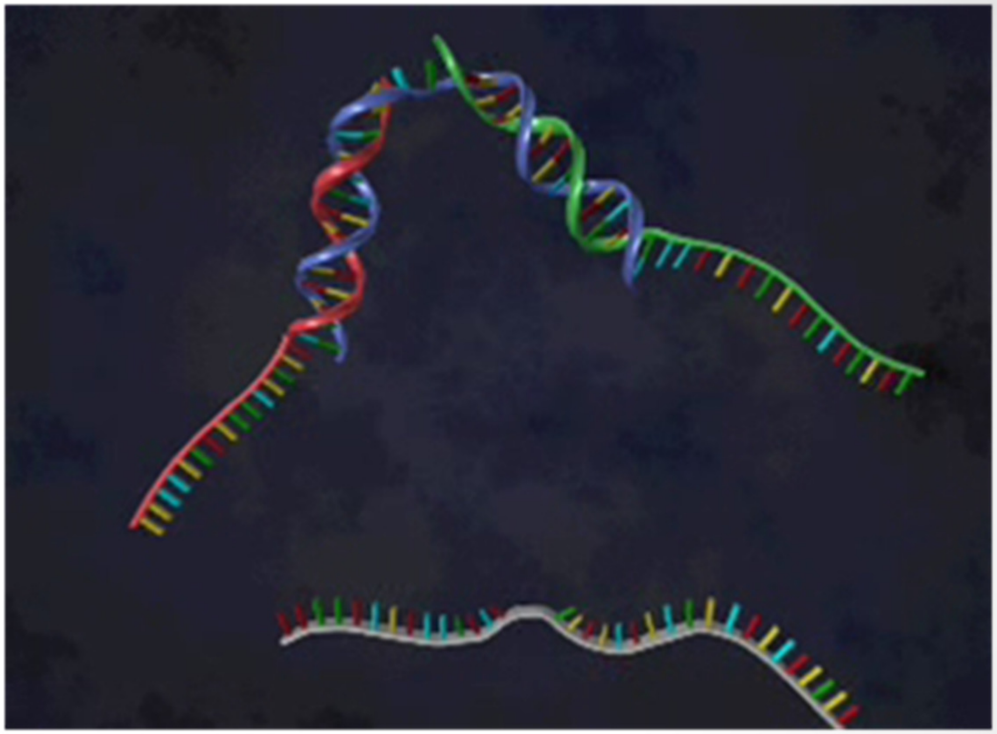


Sequence of Base Pairs (GACT alphabet)

Hybridization



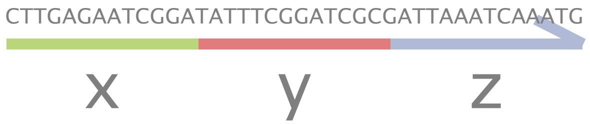
Bjorn Hogberg
Shihlab
Dana-Farber Cancer Institute



- Strands with **opposite orientation** and **complementary base pairs** stick to each other (Watson-Crick duality).
- This is all we are going to use
  - We are not going to exploit DNA replication, transcription, translation, **restriction and ligation enzymes**, etc., which enable other classes of tricks.

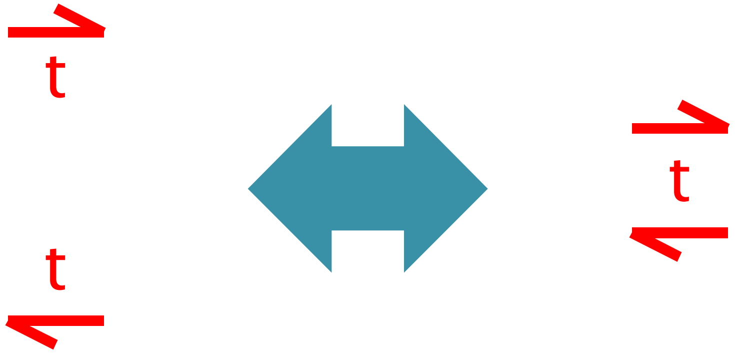
Domains

- Subsequences on a DNA strand are called domains.
- PROVIDED they are “independent” of each other.

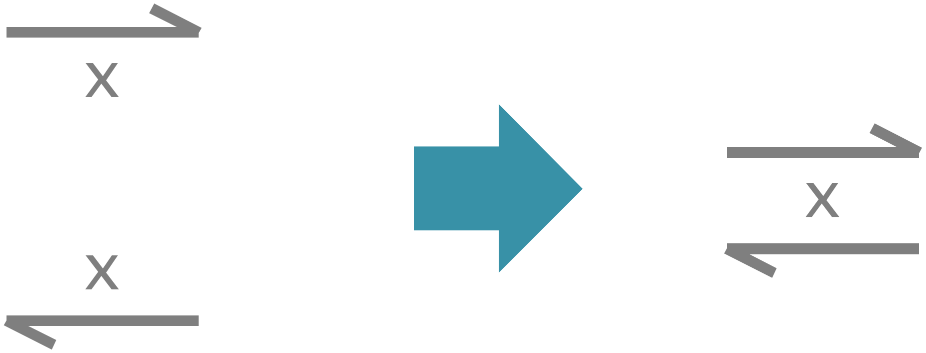


- I.e., differently named domains must not hybridize:
  - With each other
  - With each other’s complement
  - With subsequences of each other
  - With concatenations of other domains (or their complements)
  - Etc.
- How to choose domains (subsequences) that are suitably independent is a tricky issue that is still somewhat of an open problem (with a vast literature). But it can work in practice.

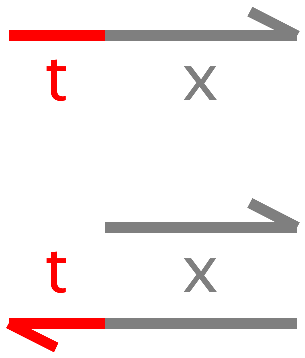
Short Domains



Long Domains

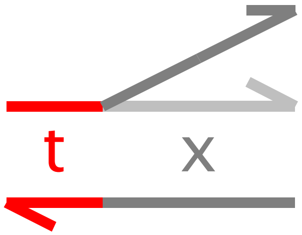


Strand Displacement



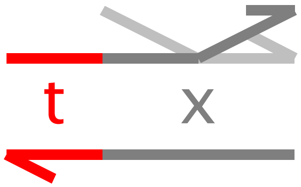
“Toehold Mediated”

Strand Displacement



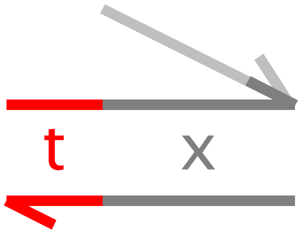
Toehold Binding

Strand Displacement



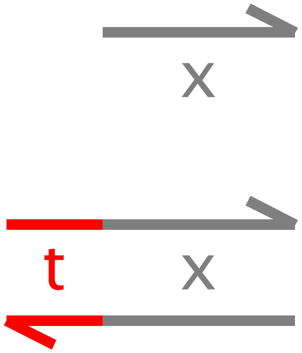
Branch Migration

Strand Displacement



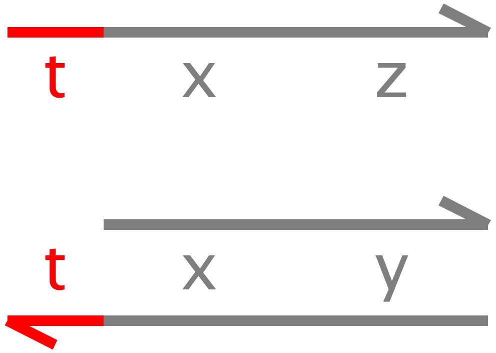
Displacement

Strand Displacement

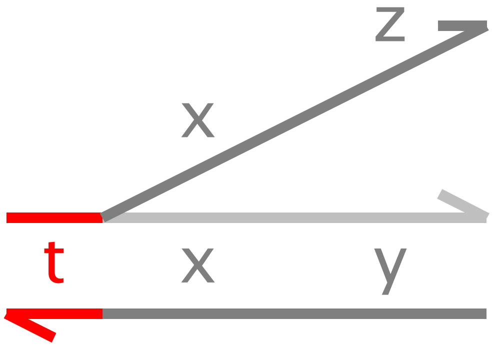


Irreversible

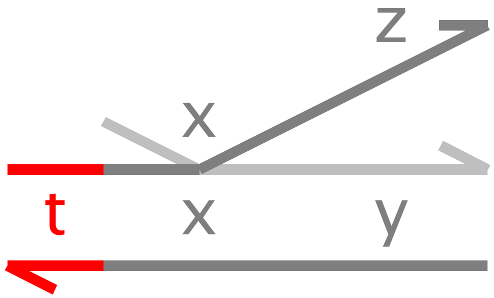
Bad Match



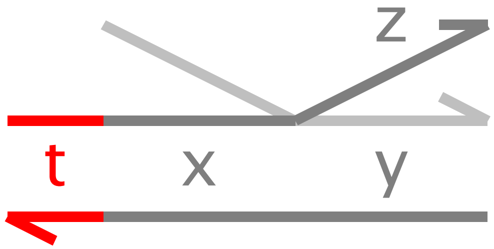
Bad Match



Bad Match

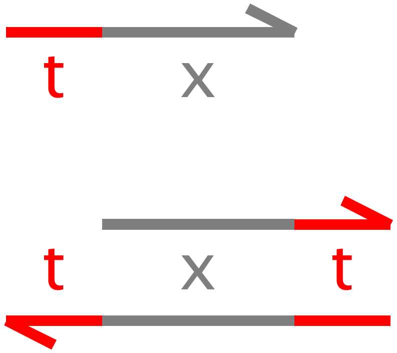


Bad Match

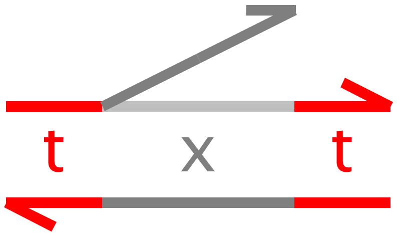


Cannot proceed
Hence will undo

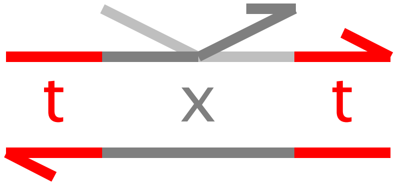
Toehold Exchange



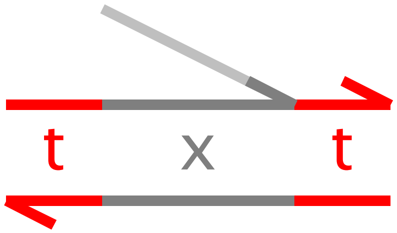
Toehold Exchange



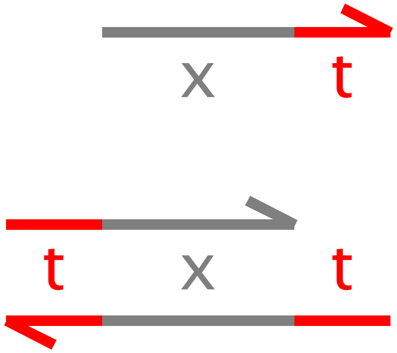
Toehold Exchange



Toehold Exchange

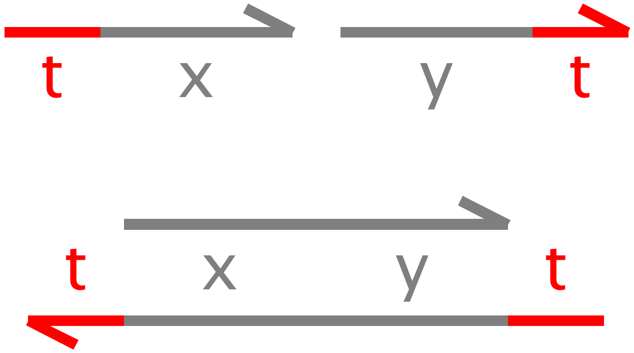


Toehold Exchange

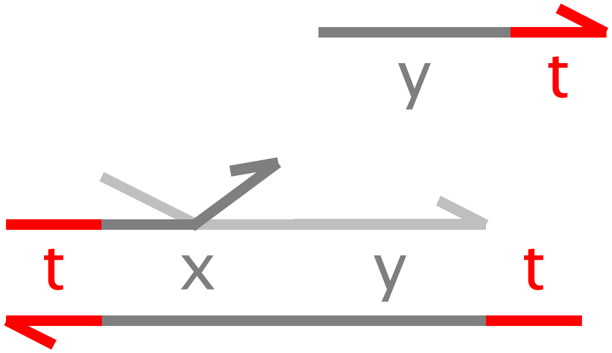


Reversible

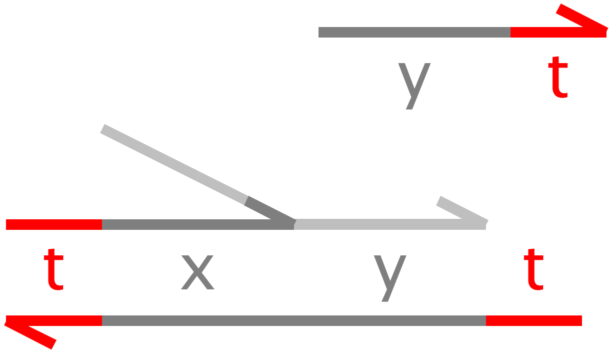
Cooperative Displacement



Cooperative Displacement

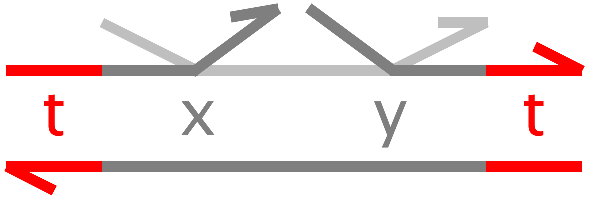


Cooperative Displacement

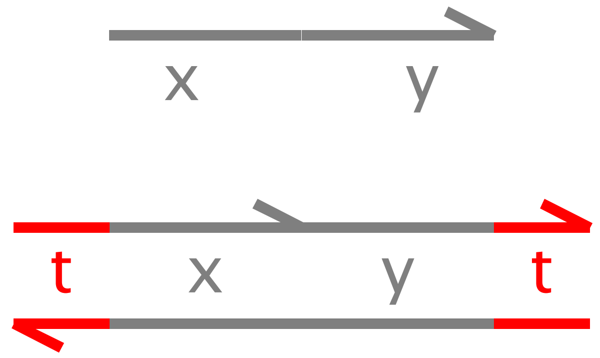


Single input
will reverse

Cooperative Displacement

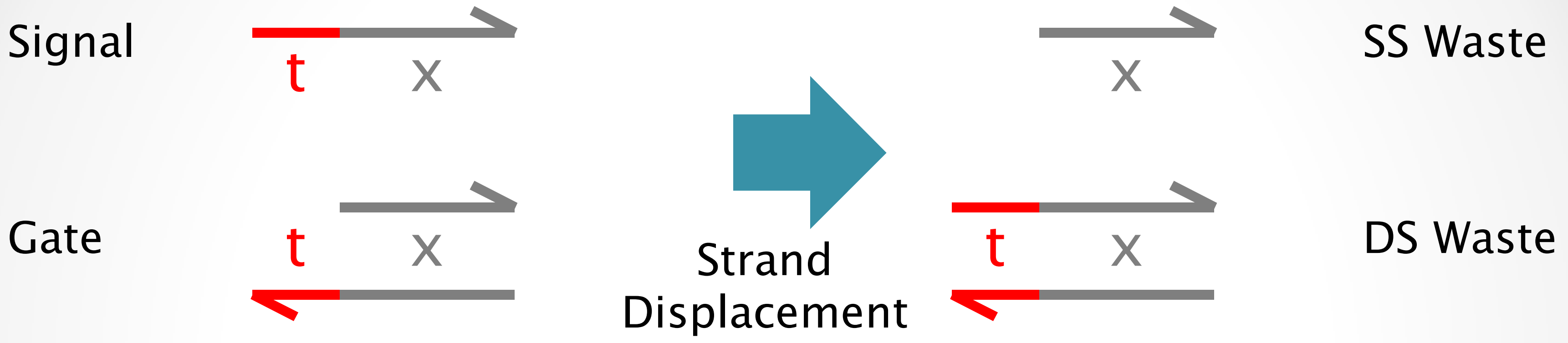


Cooperative Displacement



Double input
is irreversible

Summary (1)



Signal

t x

Gate

t x

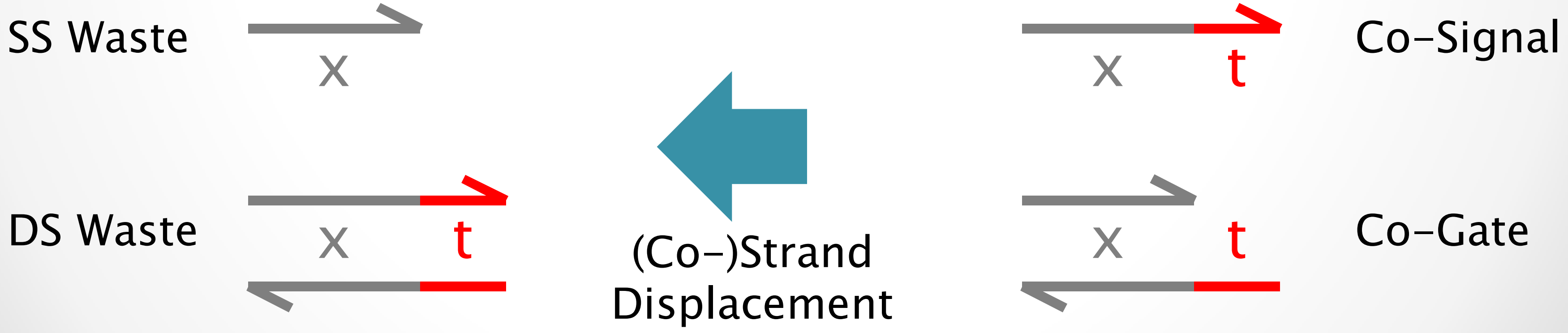
Strand Displacement

SS Waste

x

DS Waste

t x



SS Waste

x

DS Waste

x t

(Co-)Strand Displacement

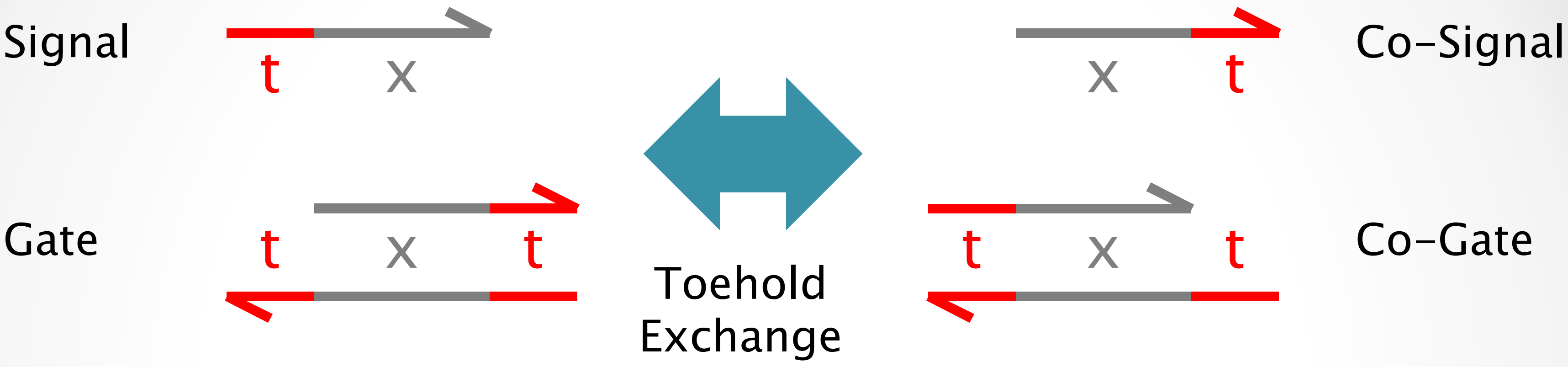
Co-Signal

x t

Co-Gate

x t

Summary (2)

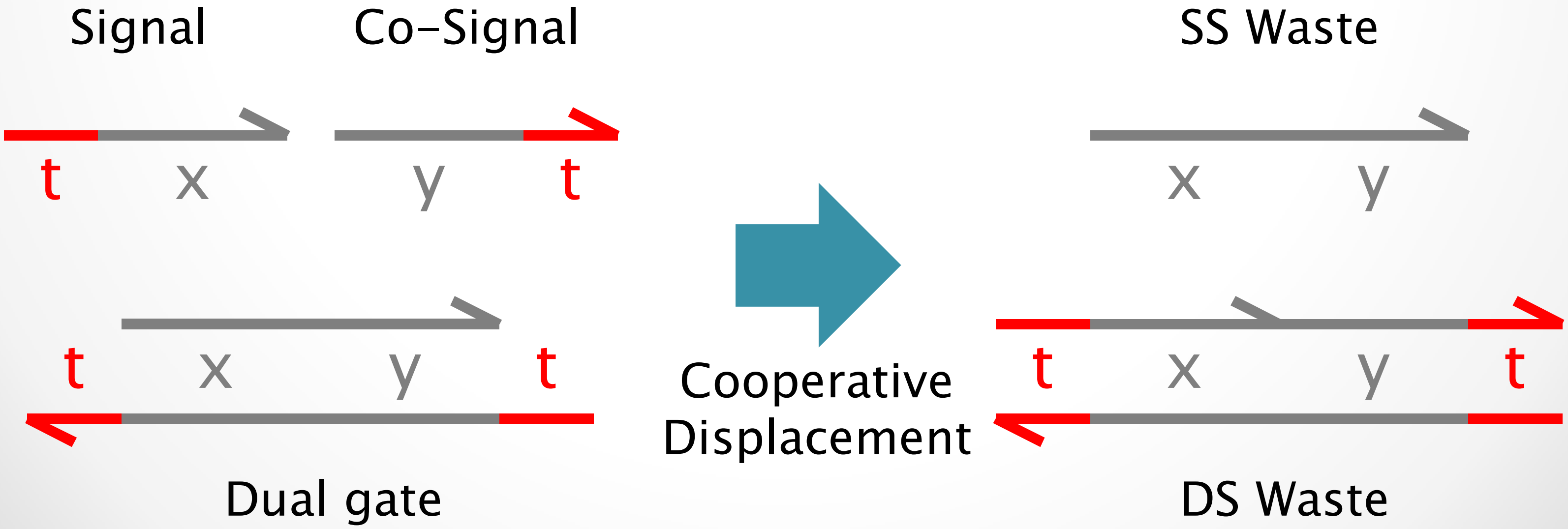

Signal

Co–Signal

Gate

Co–Gate

Toehold Exchange


Signal

Co–Signal

SS Waste

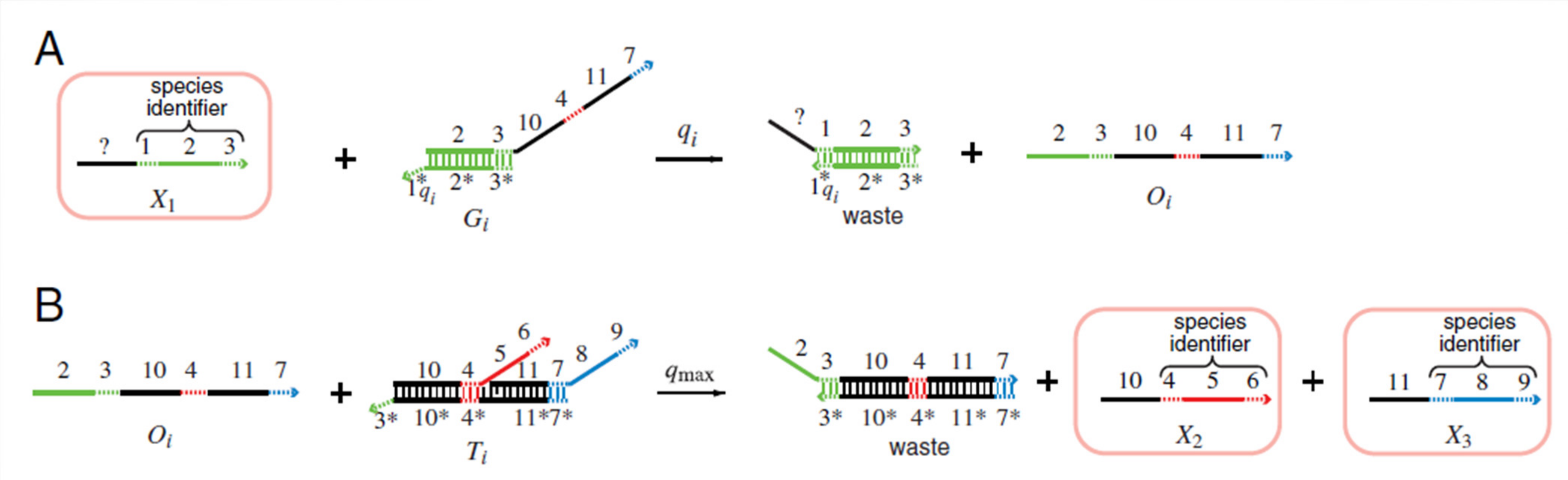
Cooperative Displacement

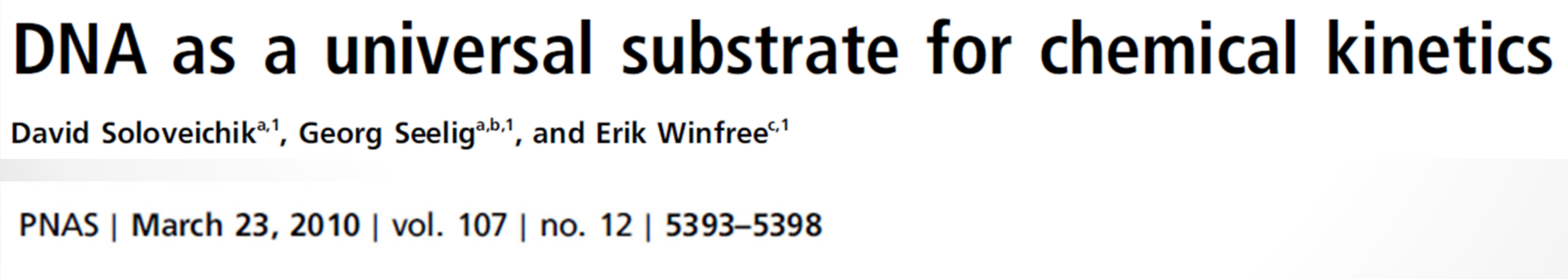
Dual gate

DS Waste

Signals & Gates

Four–Domain Signals



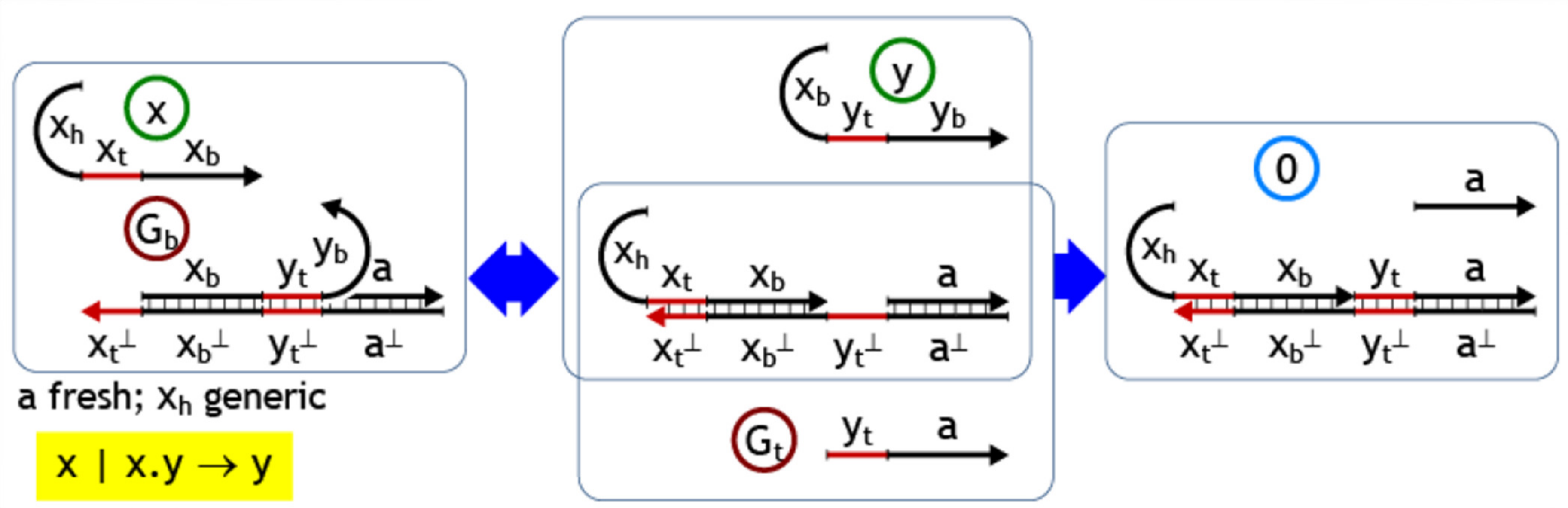


DNA as a universal substrate for chemical kinetics

David Soloveichik[a,1], Georg Seelig[a,b,1], and Erik Winfree[c,1]

PNAS | March 23, 2010 | vol. 107 | no. 12 | 5393–5398

Three-Domain Signals



a fresh; x_h generic

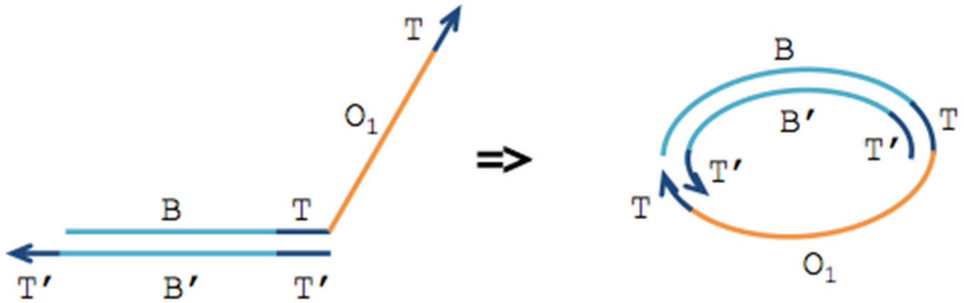
$x \mid x.y \rightarrow y$

Strand Algebras for DNA Computing

Luca Cardelli

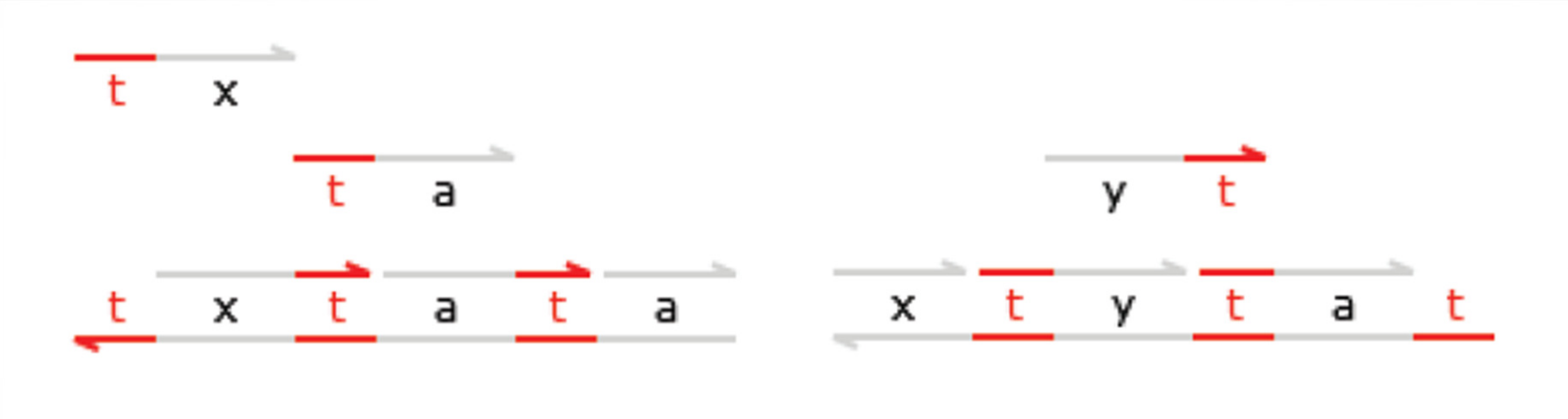
DNA Computing and Molecular Programming.
15th International Conference, DNA 15, LNCS 5877, Springer 2009, pp 12-24.

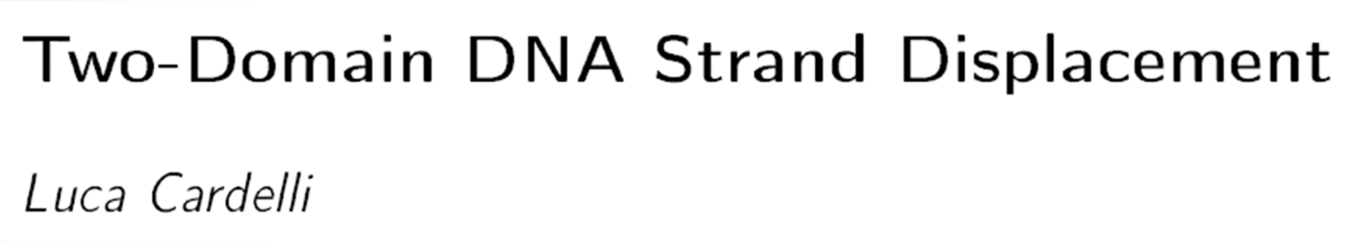
“Lulu’s Trouble”



(from D.Soloveichik)

Two–Domain Signals





Two-Domain DNA Strand Displacement

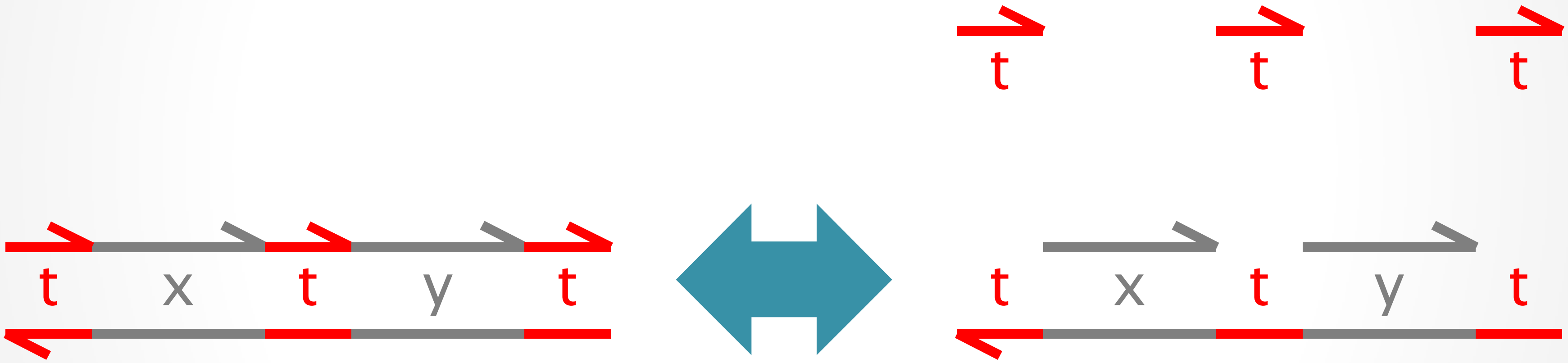
Luca Cardelli

In S. B. Cooper, E. Kashefi, P. Panangaden (Eds.):
Developments in Computational Models (DCM 2010).
EPTCS 25, 2010, pp. 33–47. May 2010.

Top-Nicked Double Strands

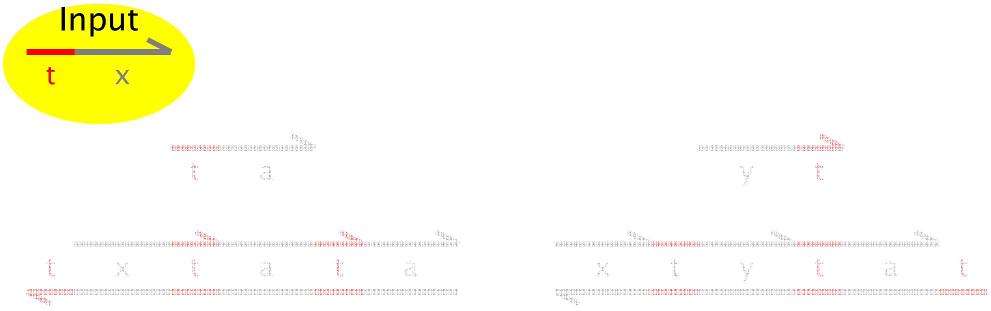
Signals have a simple structure: just two domains.

Gates have a simple structure:
'top-nicked' double-stranded DNA with no 'frills'.

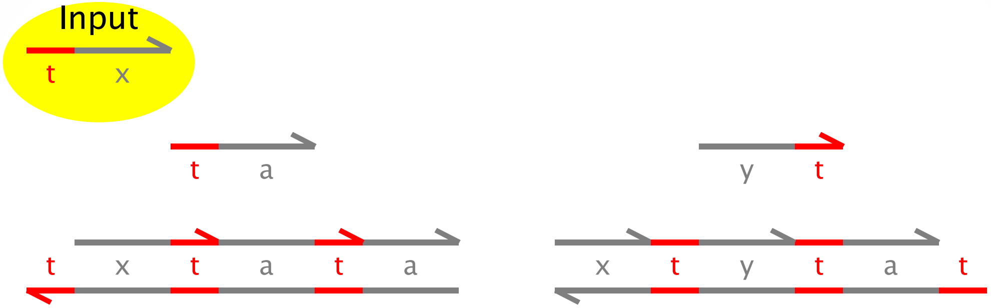


A top-nicked double-strand is 'equivalent'
to a double strand with open toeholds.
These situations shall not be distinguished.

Transducer x→y

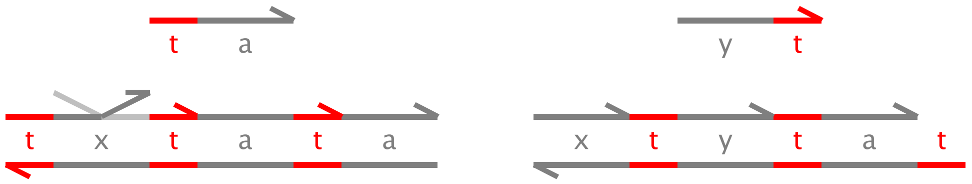


Transducer x→y

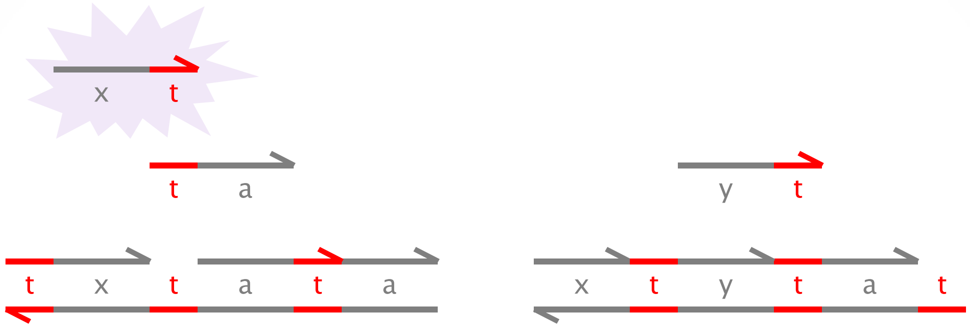


ta is a *private* signal (a different ‘a’ for each xy pair)

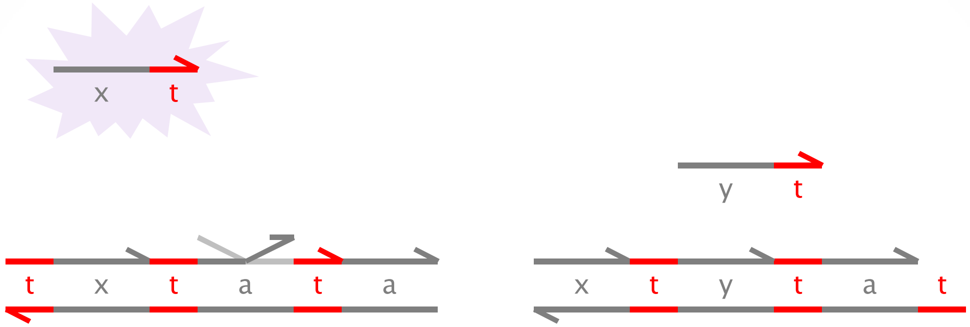
Transducer x→y



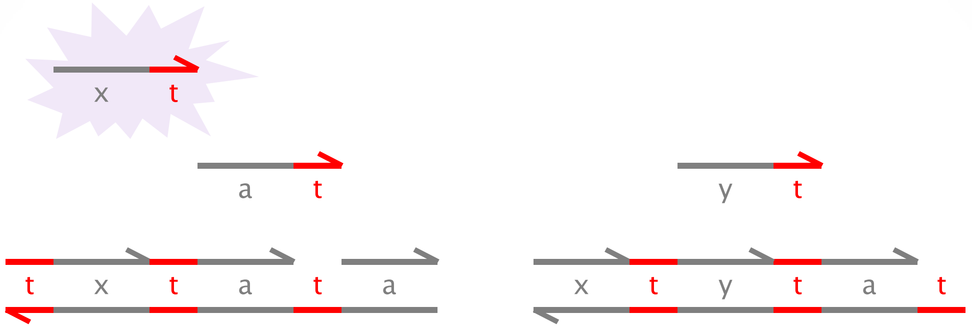
Transducer x→y



Transducer x→y

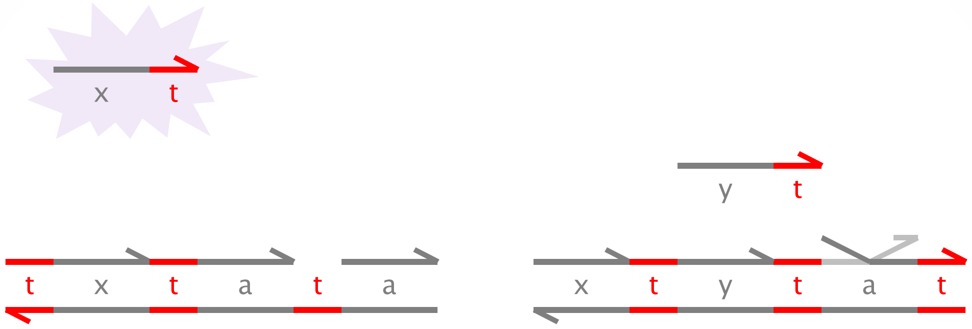


Transducer x→y

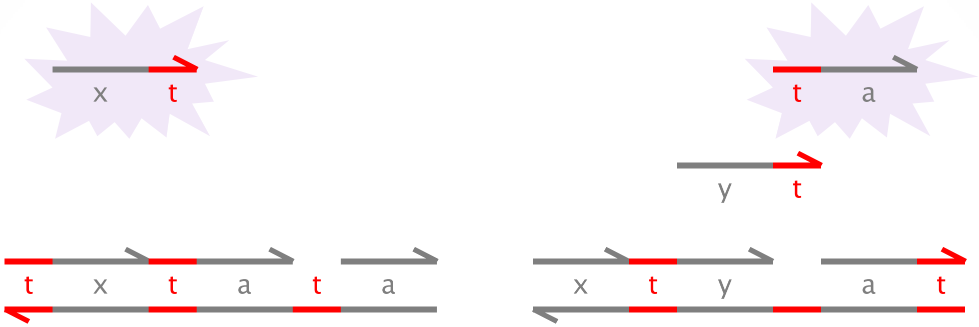


So far, a **tx** *signal* has produced an **at** *cosignal.*
But we want signals as output, not cosignals.

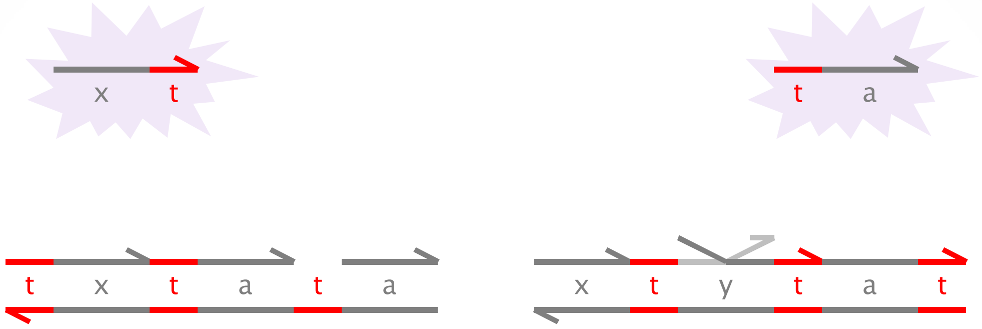
Transducer x→y



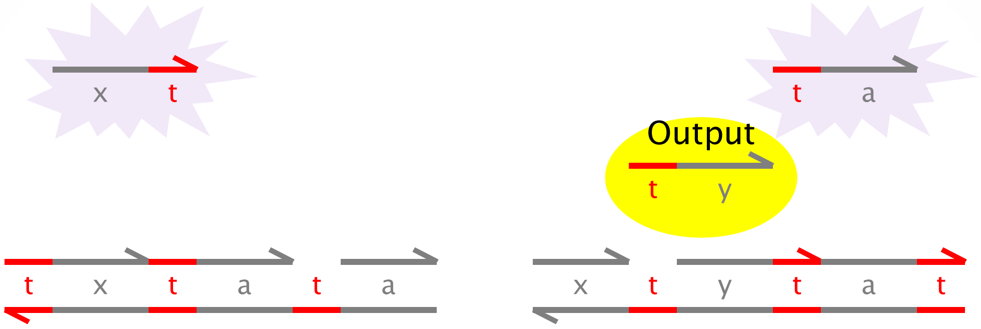
Transducer x→y



Transducer x→y



Transducer x→y



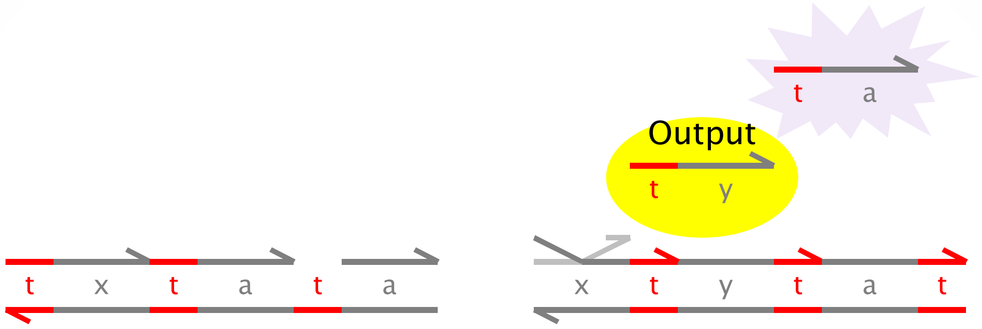
Here is our output ty *signal*.

But we are not done yet:

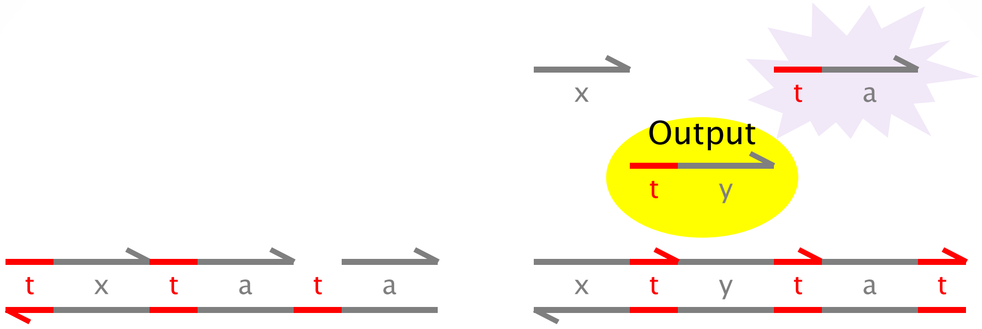
1) We need to make the output irreversible.
2) We need to remove the garbage.

We can use (2) to achieve (1).

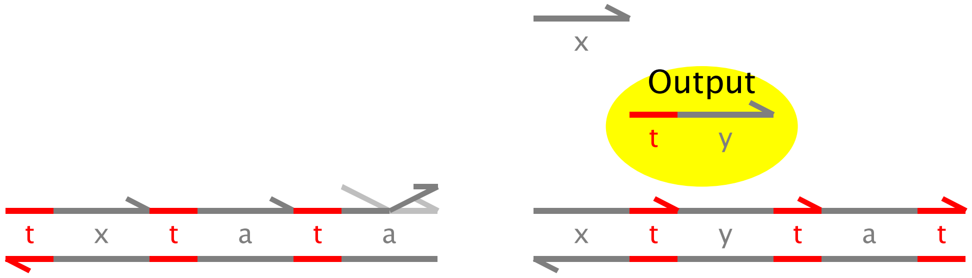
Transducer x→y



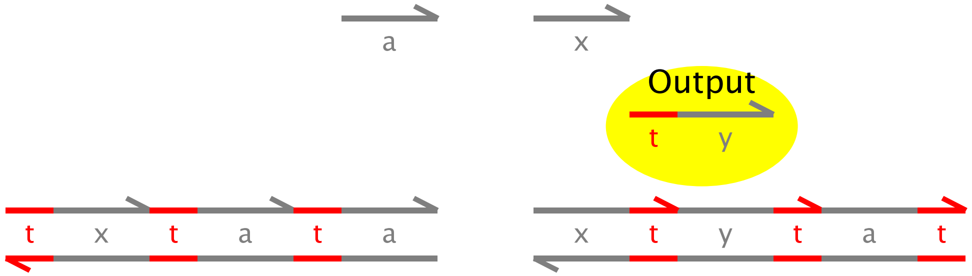
Transducer x→y



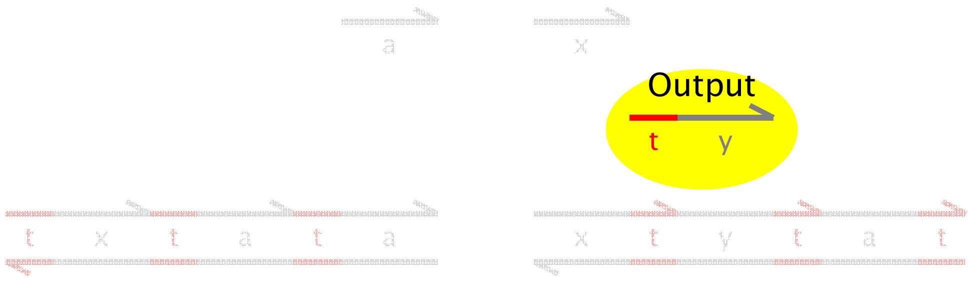
Transducer x→y



Transducer x→y



Transducer x→y

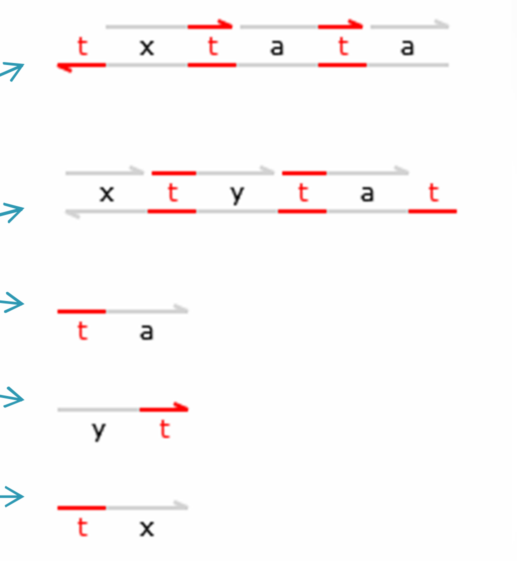


Done.

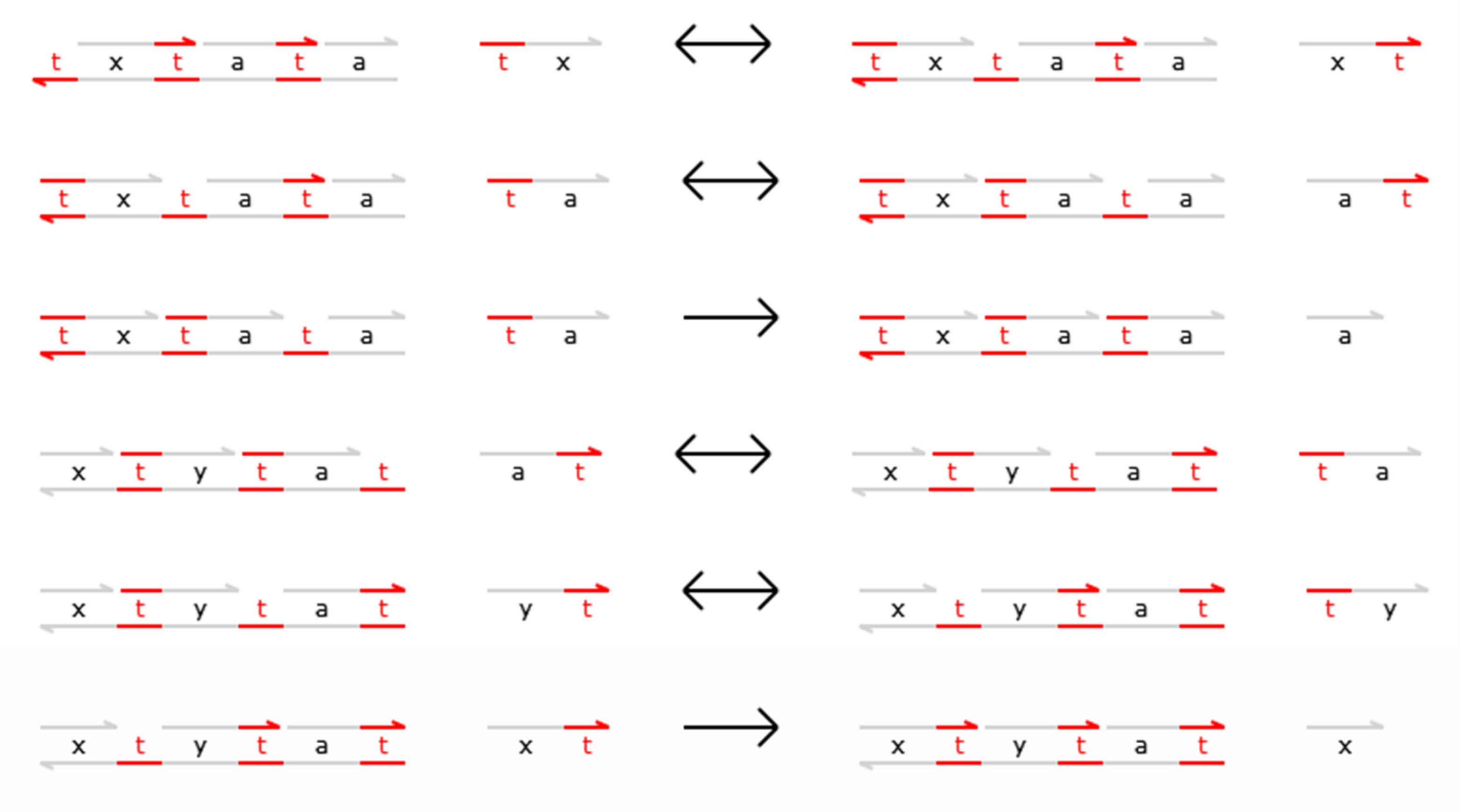
Note the **tata** motif and how it helps in collection.

The Transducer in DSD

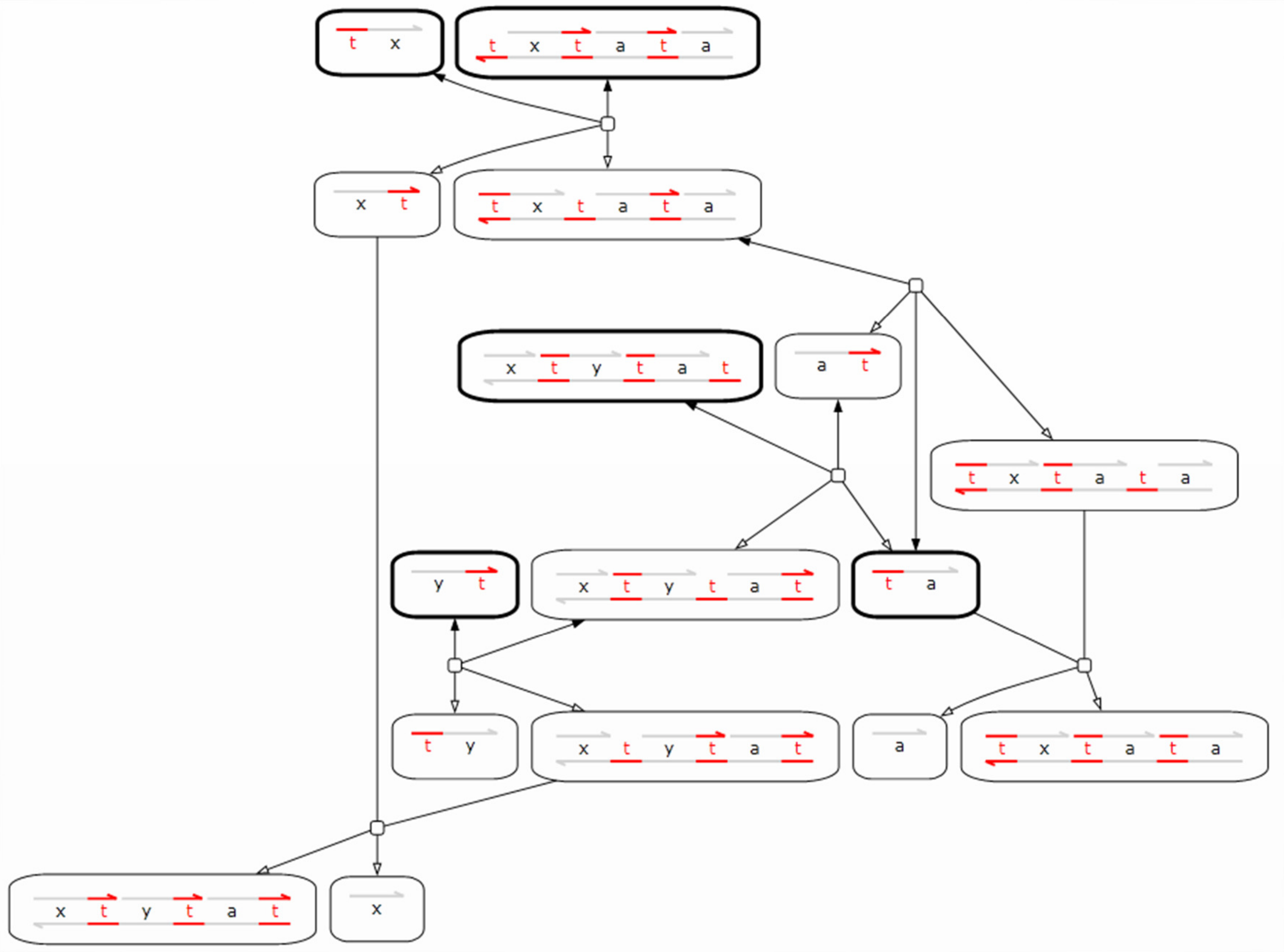
```
directive sample 50.0
directive plot <t^ x>; <t^ y>
directive scale 1.0
new t@1.0,1.0

def Tr(N, x, y) =
new a
( N* <t^ a>
| N* <y t^>
| N* t^:[x t^]:[a t^]:[a]
| N* [x]:[t^ y]:[t^ a]:t^
)

( Tr(10, x, y)
| 1* <t^ x>
)
```



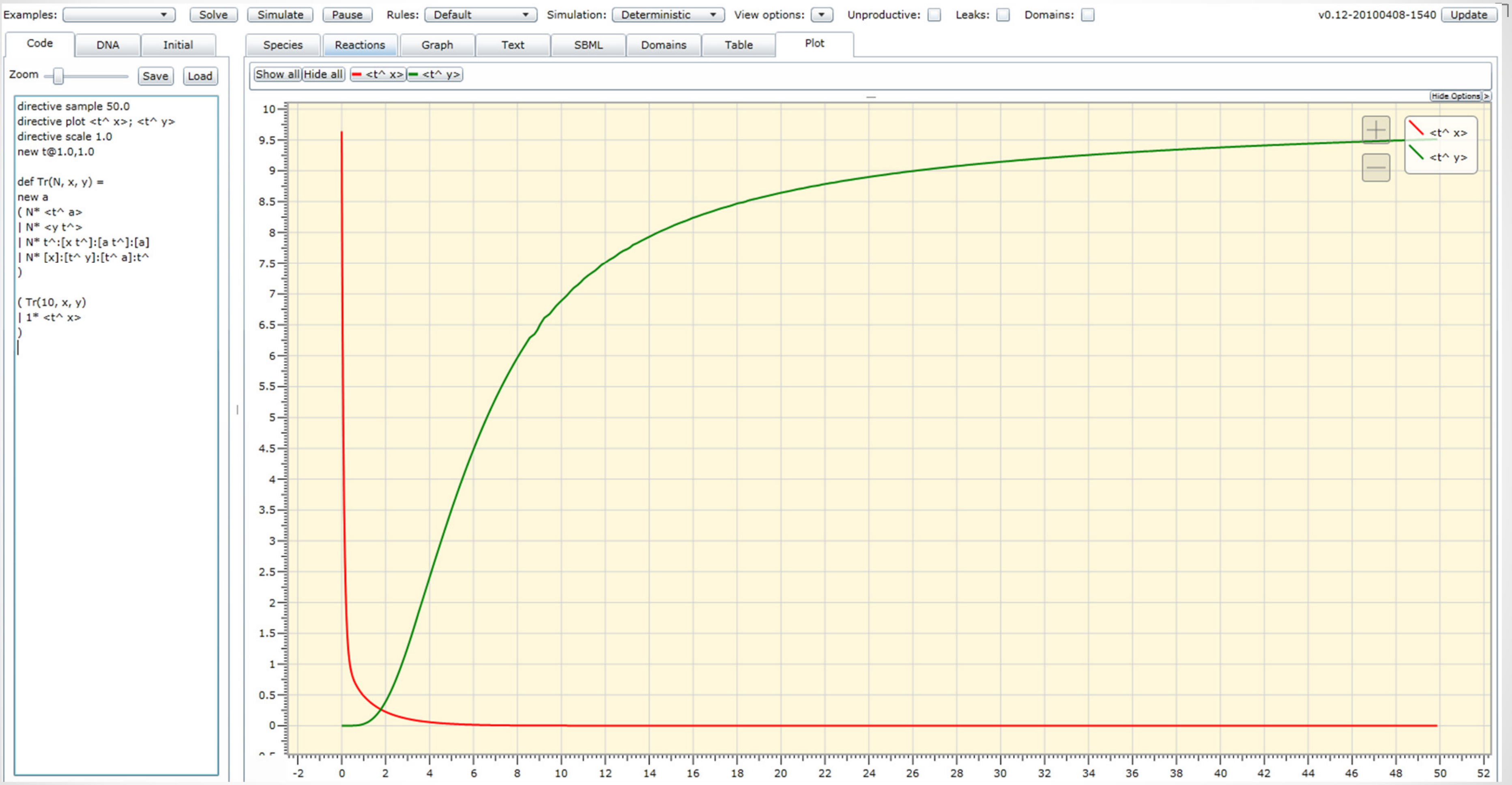
Transducer Reactions



Transducer Reaction Graph



Transducer Simulation



Rules: Default

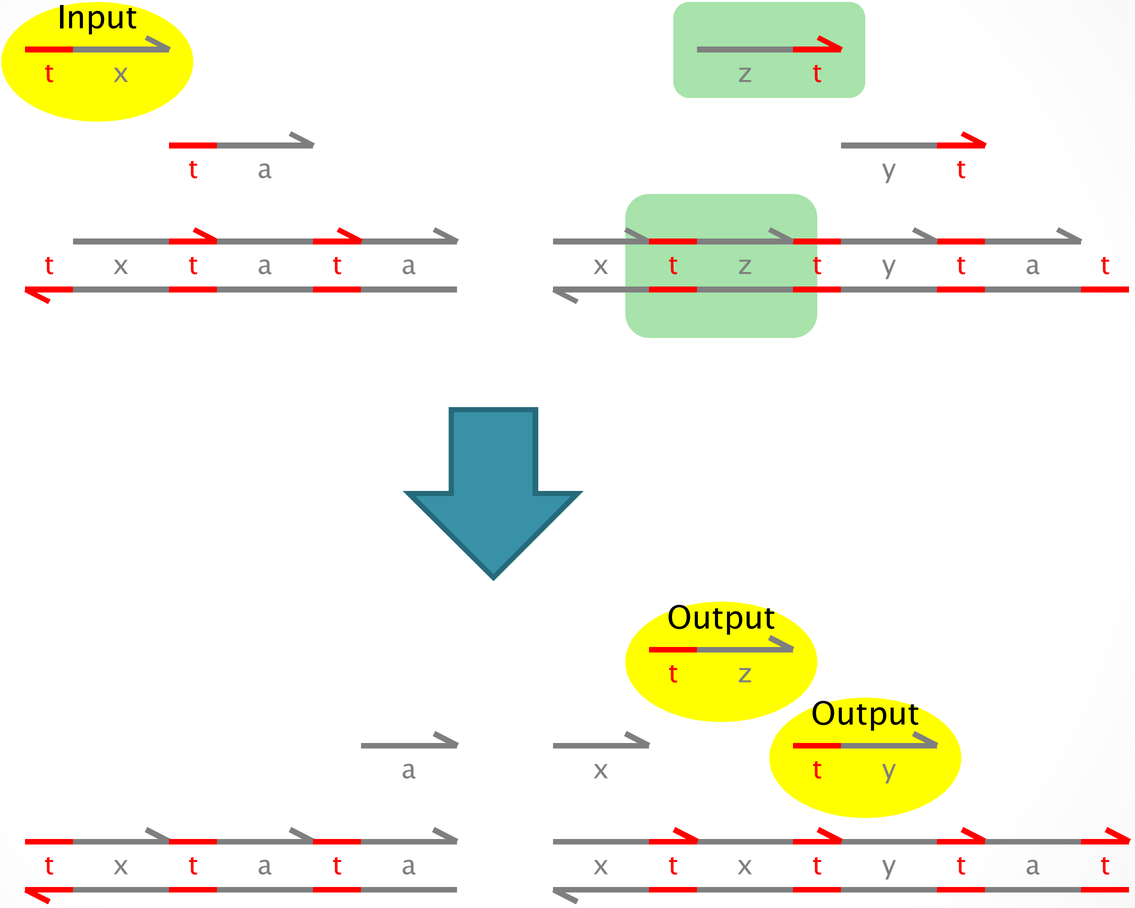
Simulation: Deterministic

Unproductive: □ Leaks: □ Domains: □

v0.12-20100408-1540

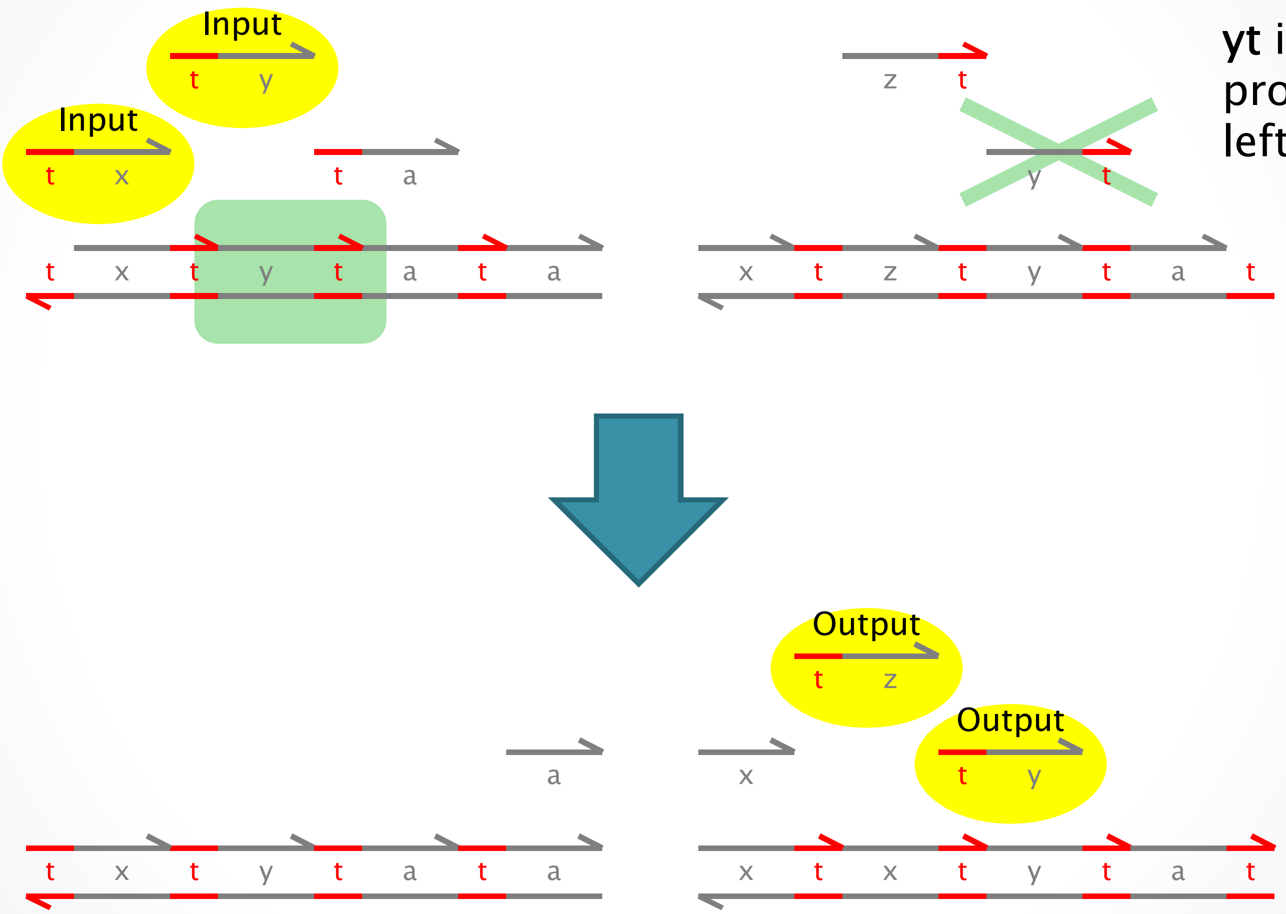
```
directive sample 50.0
directive plot <t^ x>; <t^ y>
directive scale 1.0
new t@1.0,1.0

def Tr(N, x, y) =
new a
( N* <t^ a>
| N* <y t^>
| N* t^:[x t^]:[a t^]:[a]
| N* [x]:[t^ y]:[t^ a]:t^
)

( Tr(10, x, y)
| 1* <t^ x>
)
```

Fork x→y+z



(Amplifier: x→x+x)

Catalyst x+y→y+z

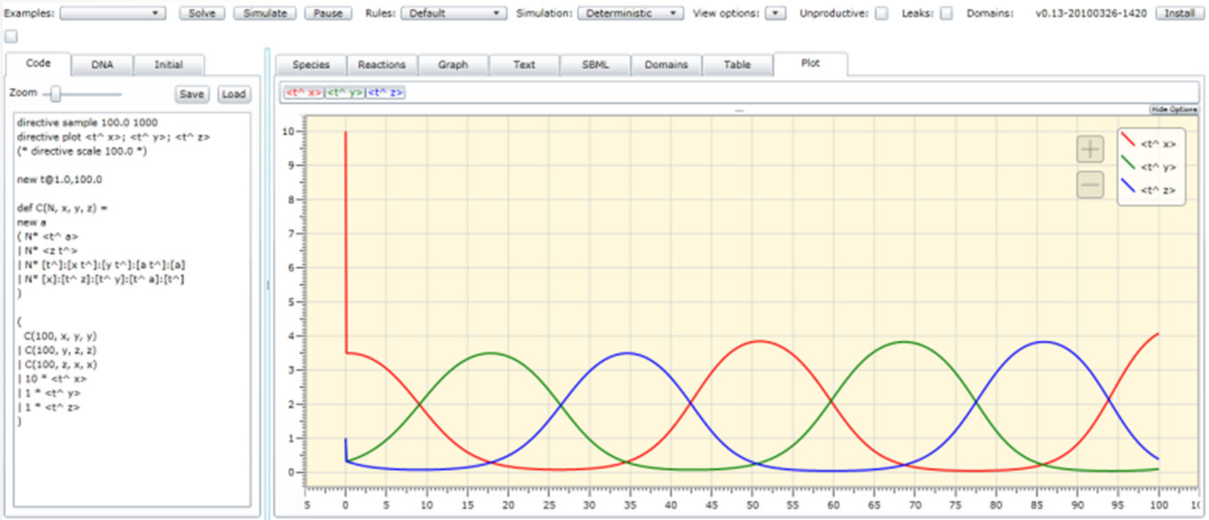


yt is kindly provided by the left hand side.

(Autocatalyst: x+y→y+y)

Autocatalytic Oscillator

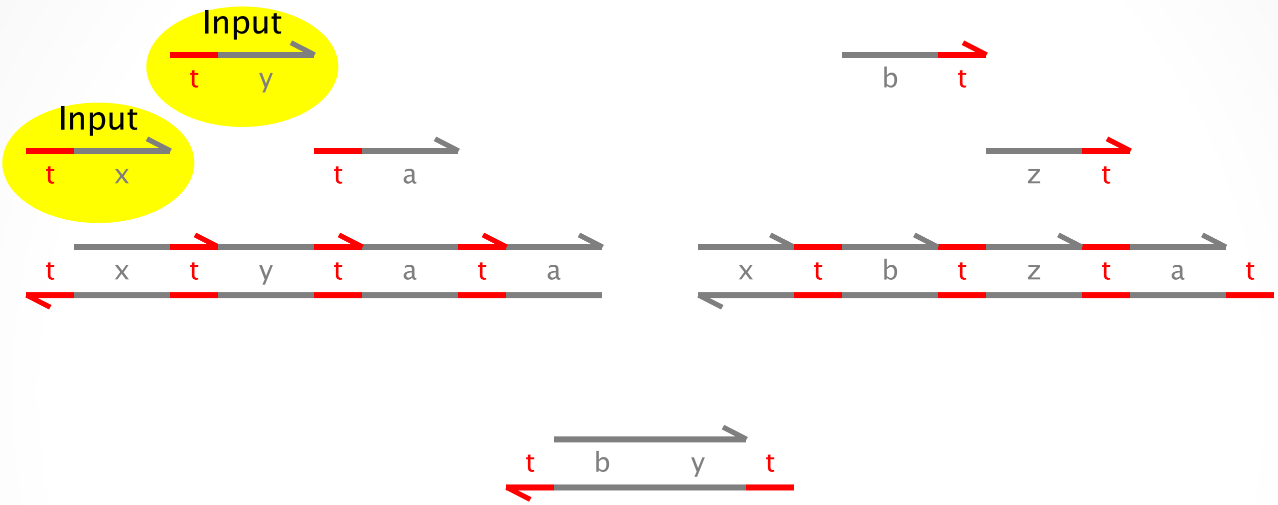
x+y→y+y
y+z→z+z
z+x→x+x



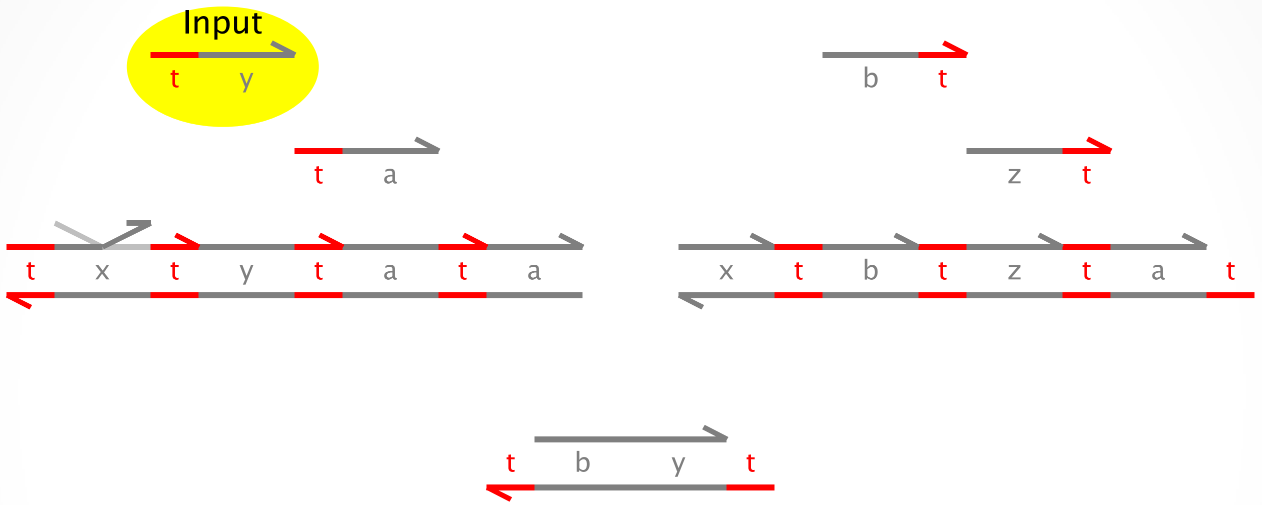
```
directive sample 100.0 1000
directive plot <t^ x>; <t^ y>; <t^ z>
(* directive scale 100.0 *)

new t@1.0,100.0

def C(N, x, y, z) =
new a
( N* <t^ a>
| N* <z t^>
| N* [t^]:[x t^]:[y t^]:[a t^]:[a]
| N* [x]:[t^ z]:[t^ y]:[t^ a]:[t^]
)

(
  C(100, x, y, y)
| C(100, y, z, z)
| C(100, z, x, x)
| 10 * <t^ x>
| 1 * <t^ y>
| 1 * <t^ z>
)
```

```
directive sample 100.0 1000
directive plot <t^ x>; <t^ y>;
<t^ z>
(* directive scale 100.0 *)

new t@1.0,100.0

def C(N, x, y, z) =
new a
( N* <t^ a>
| N* <z t^>
| N* [t^]:[x t^]:[y t^]:[a t^]:[a]
| N* [x]:[t^ z]:[t^ y]:[t^ a]:[t^]
)

(
  C(100, x, y, y)
| C(100, y, z, z)
| C(100, z, x, x)
| 10 * <t^ x>
| 1 * <t^ y>
| 1 * <t^ z>
)
```

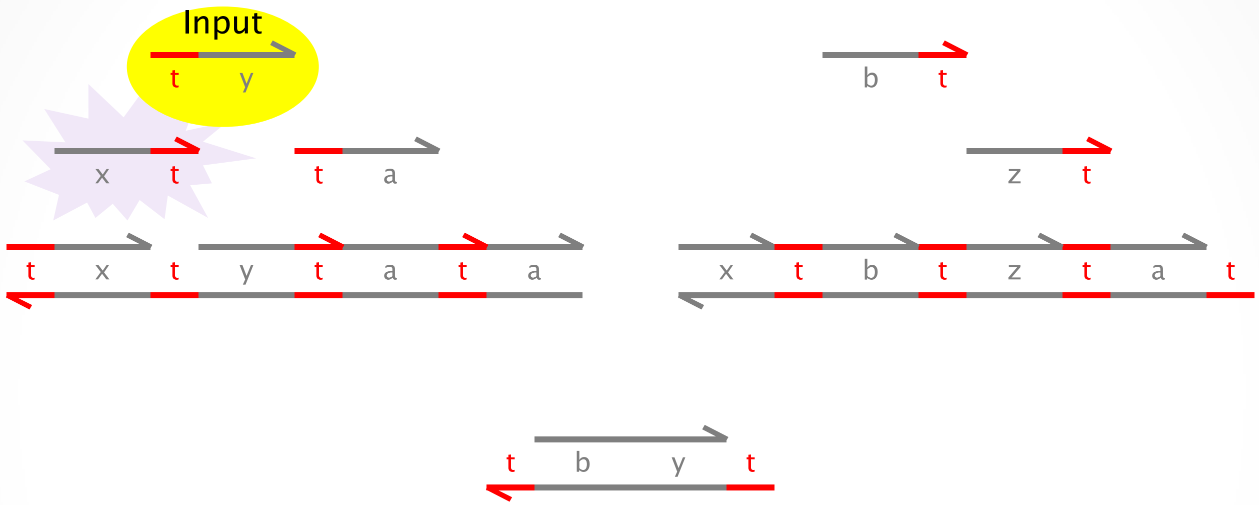
Join x+y→z



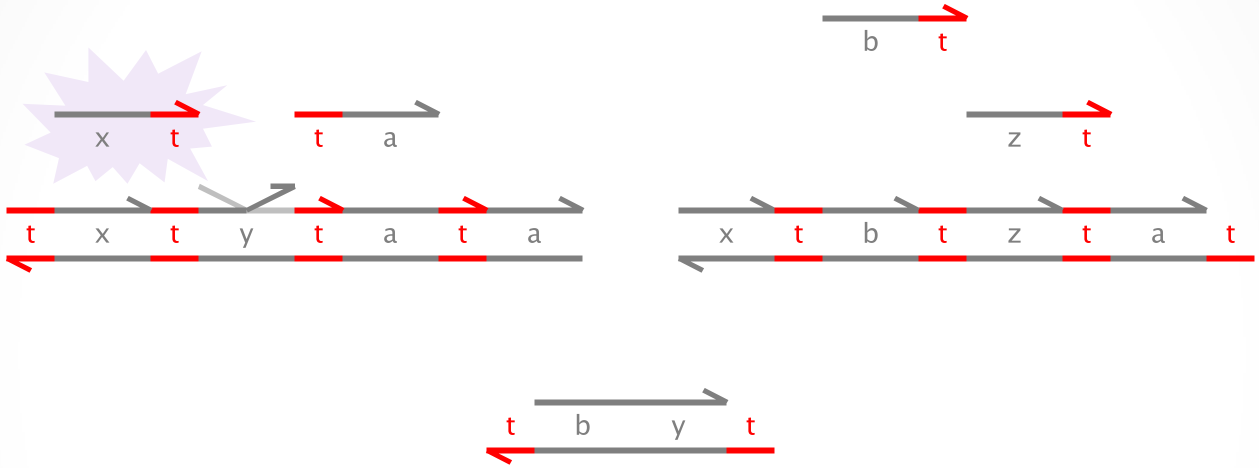
Join $x+y \rightarrow z$



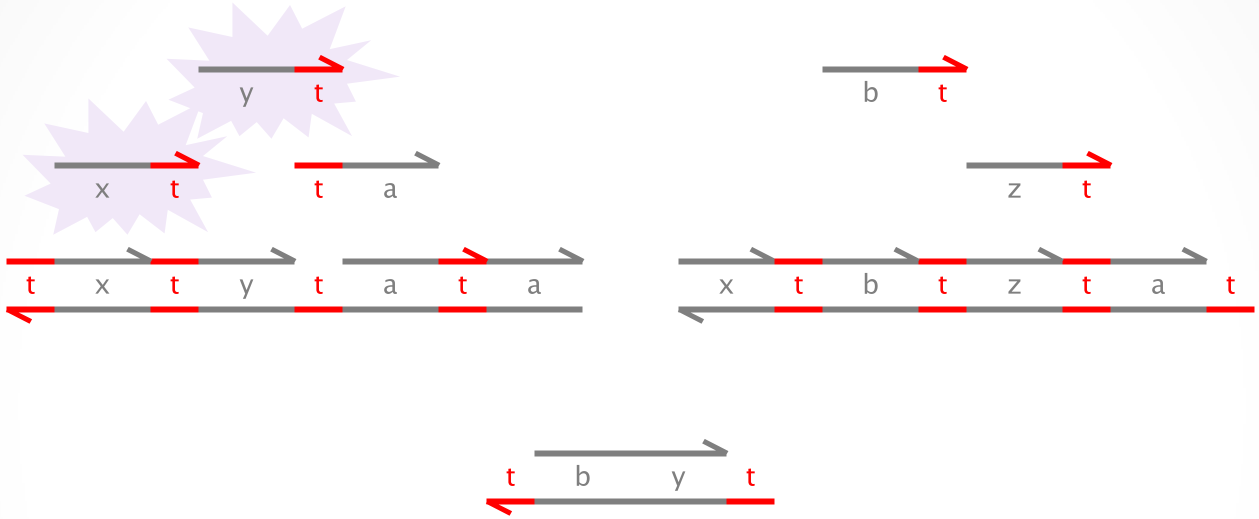
Join x+y→z



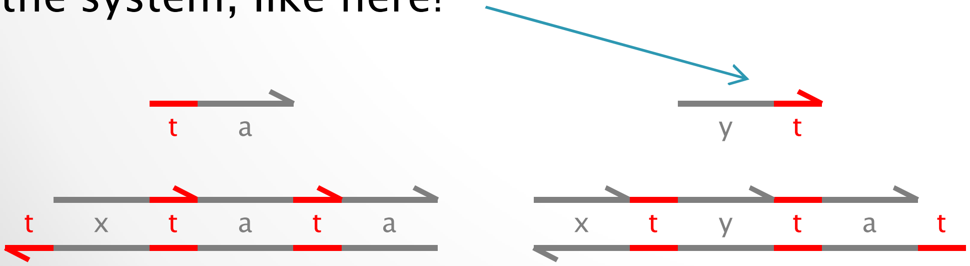
Join x+y→z



Join x+y→z



We cannot have a collector just waiting for **yt**, because there may be innocent **yt** elsewhere in the system, like here!

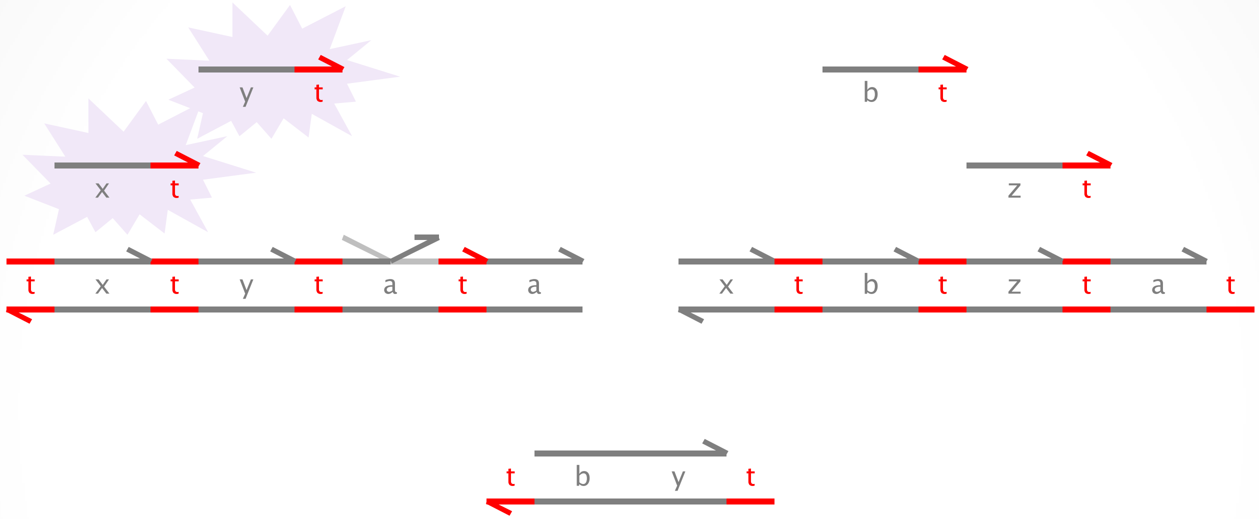


Transducer x→y

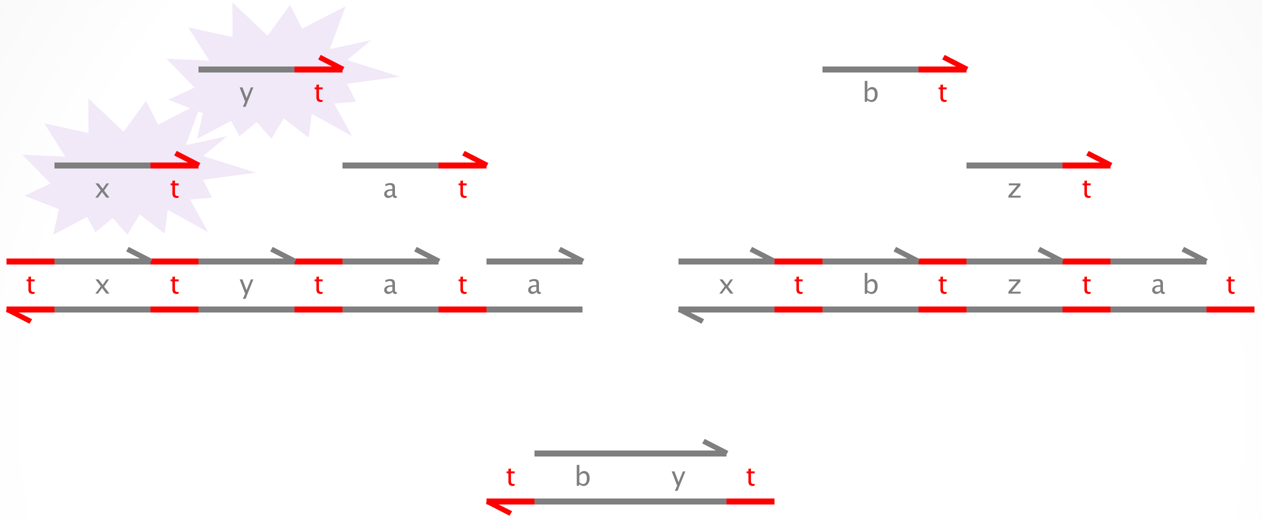
Instead, the collection of **yt** must be triggered only by a signal signifying that an x+y→z gate has fired. That signal is **tb**, which will trigger the collection of **yt** after output **tz** is produced.

bt is a *private* signal
(a different ‘b’ for each xyz triple)

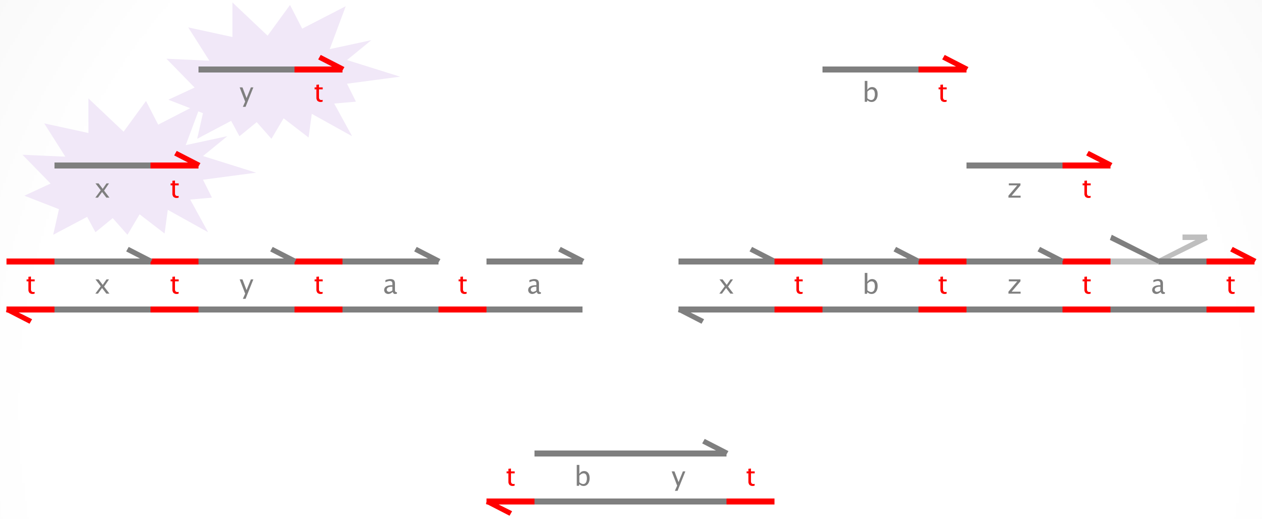
Join x+y→z



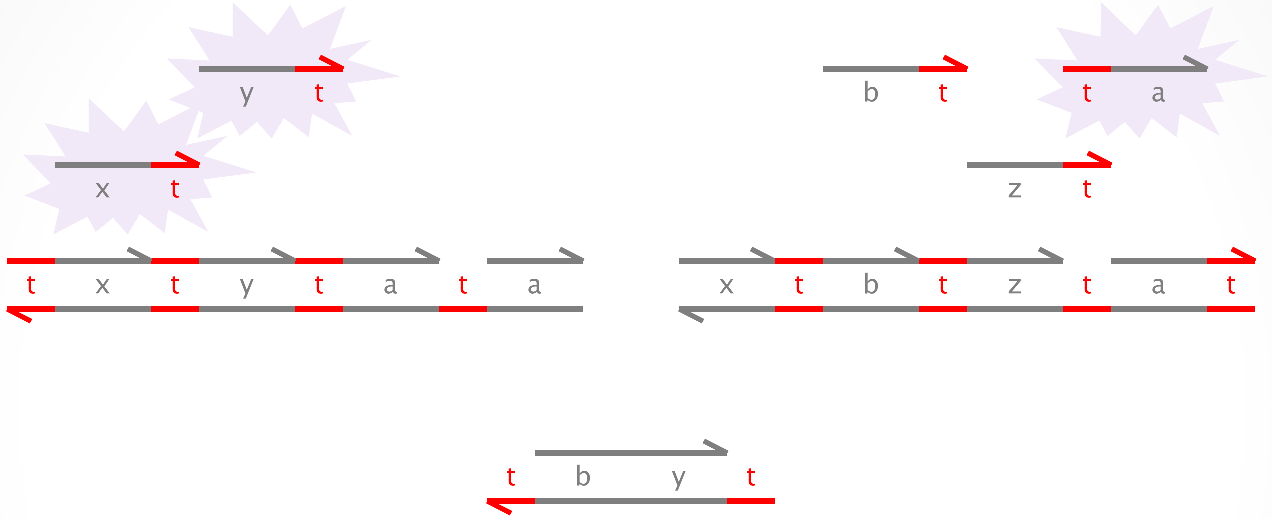
Join x+y→z



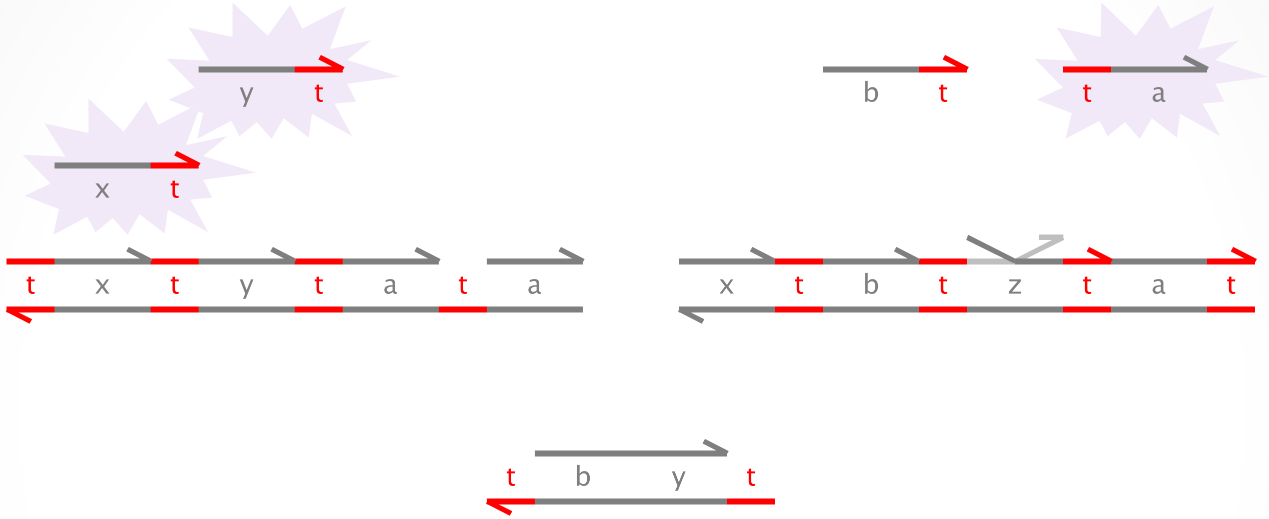
Join x+y→z



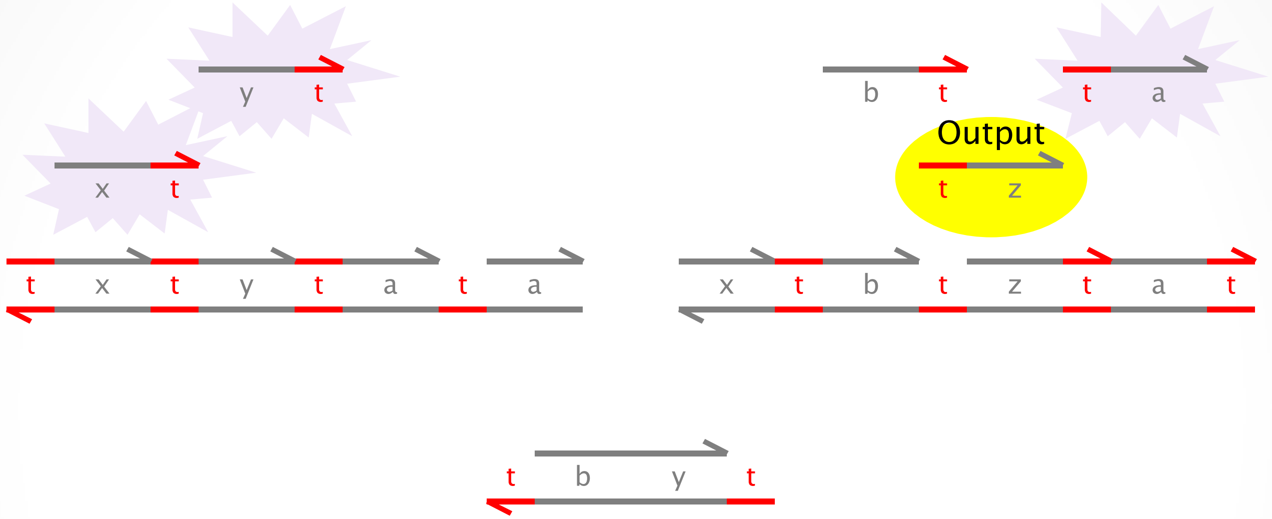
Join x+y→z



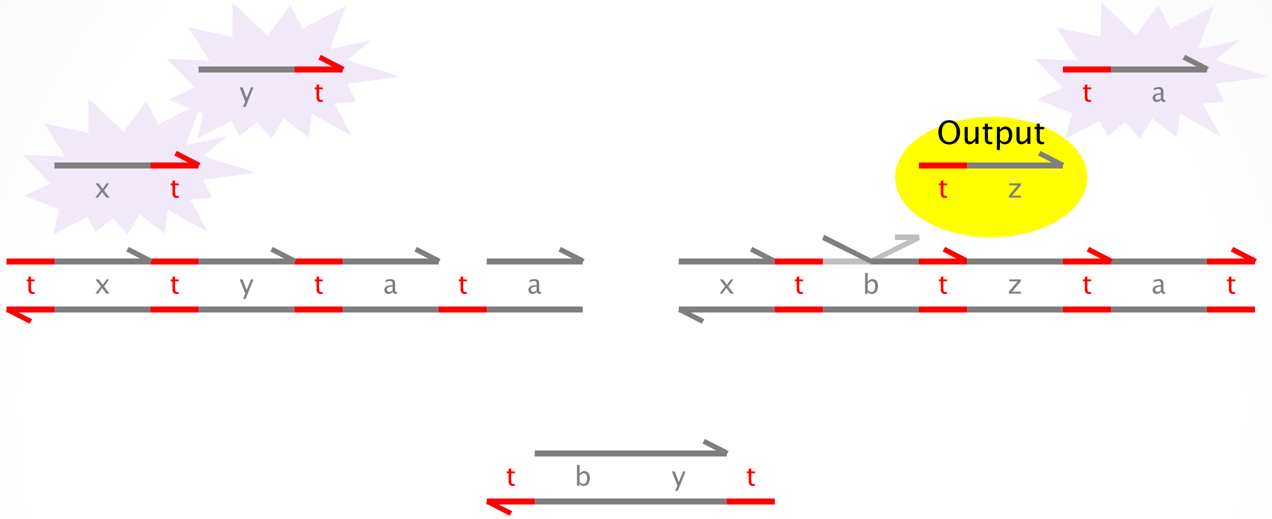
Join x+y→z



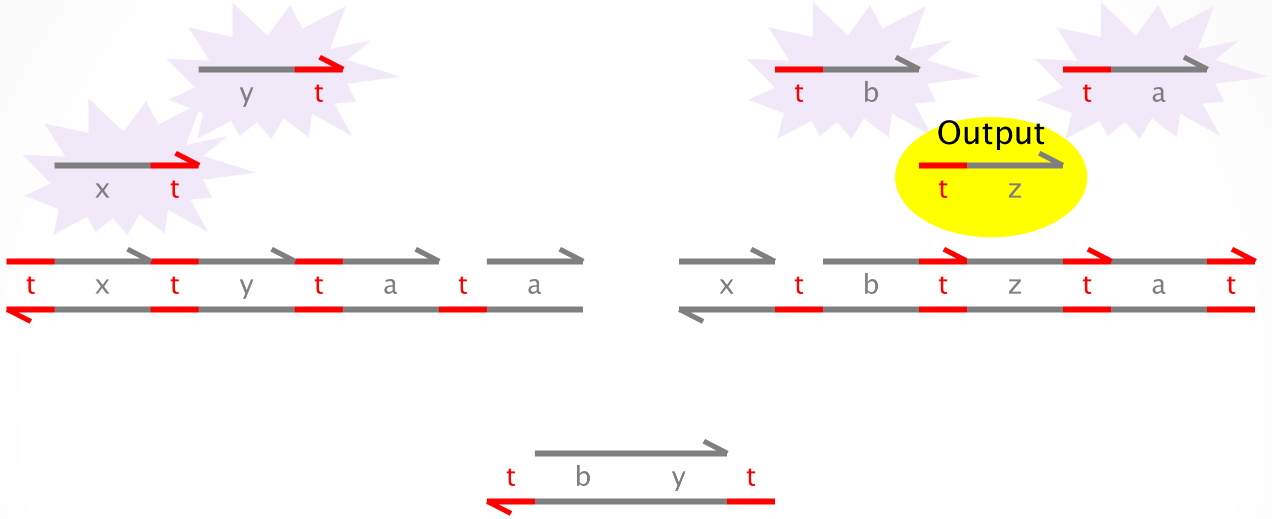
Join x+y→z



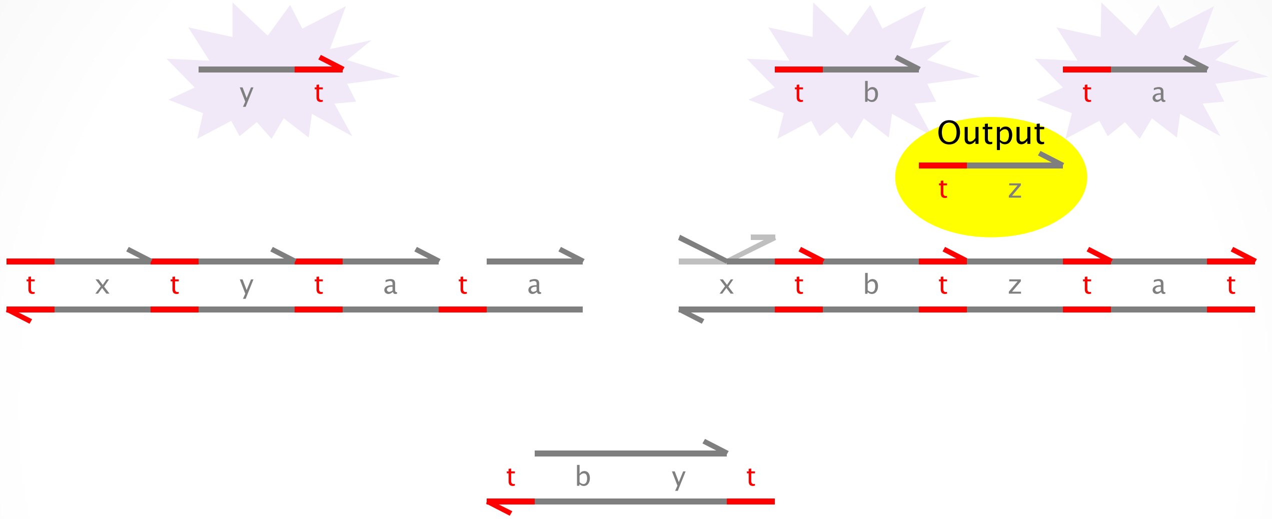
Join x+y→z



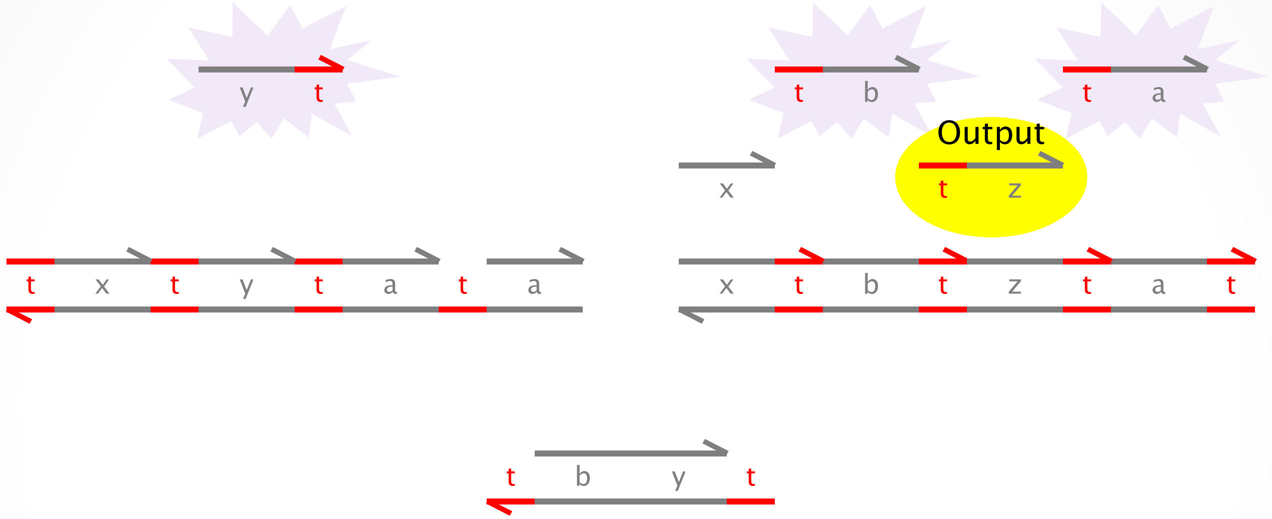
Join x+y→z



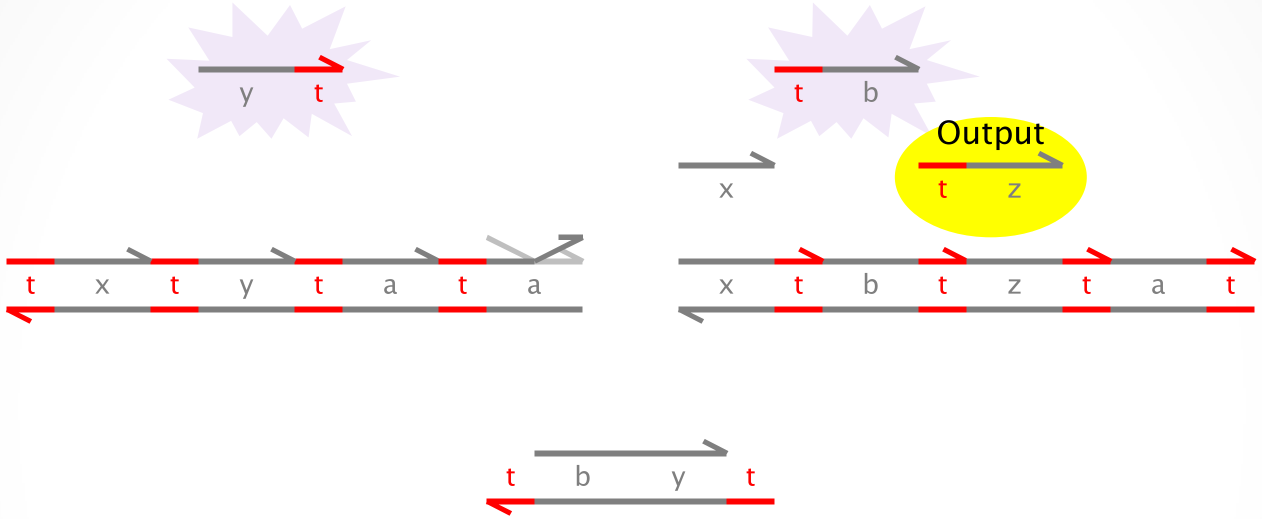
Join x+y→z



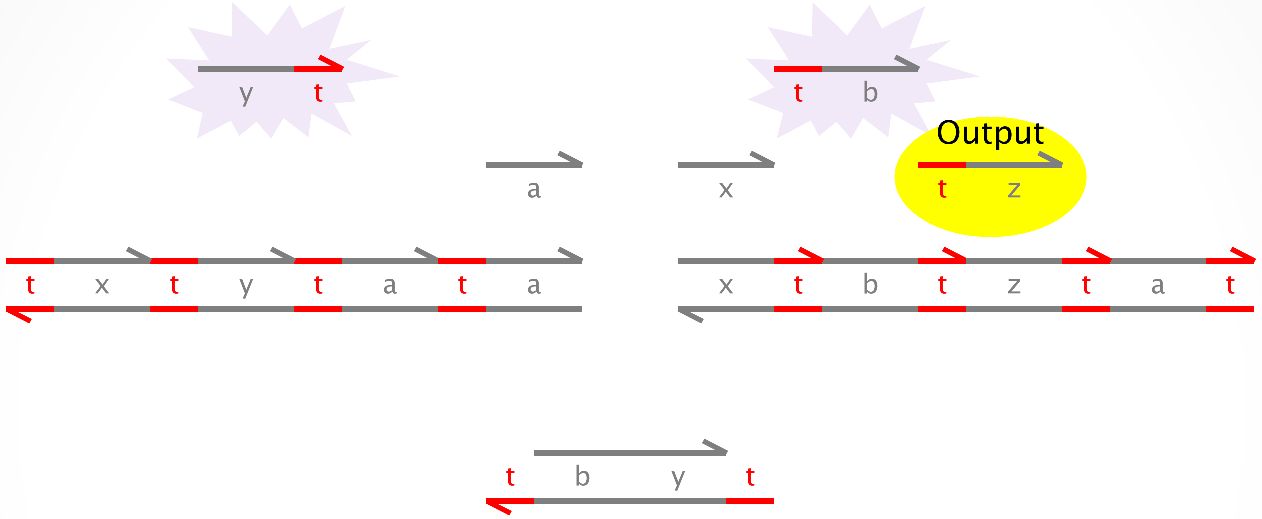
Join x+y→z



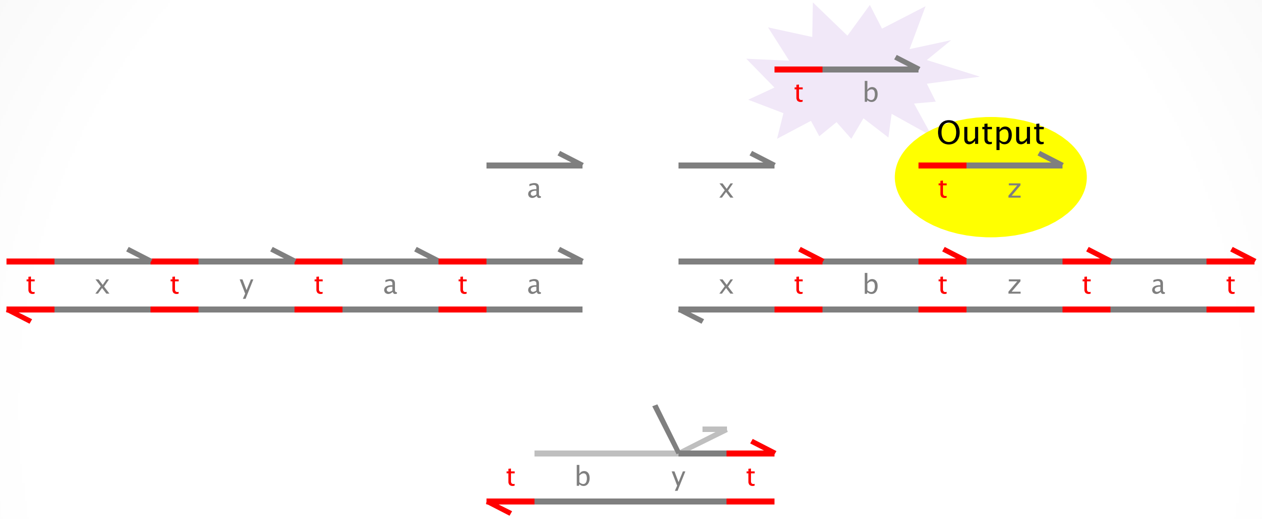
Join x+y→z



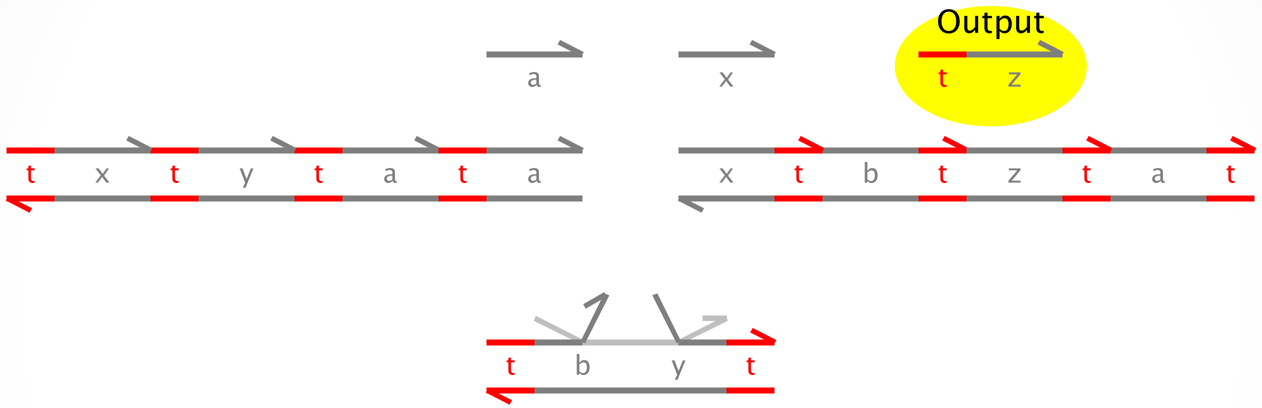
Join x+y→z



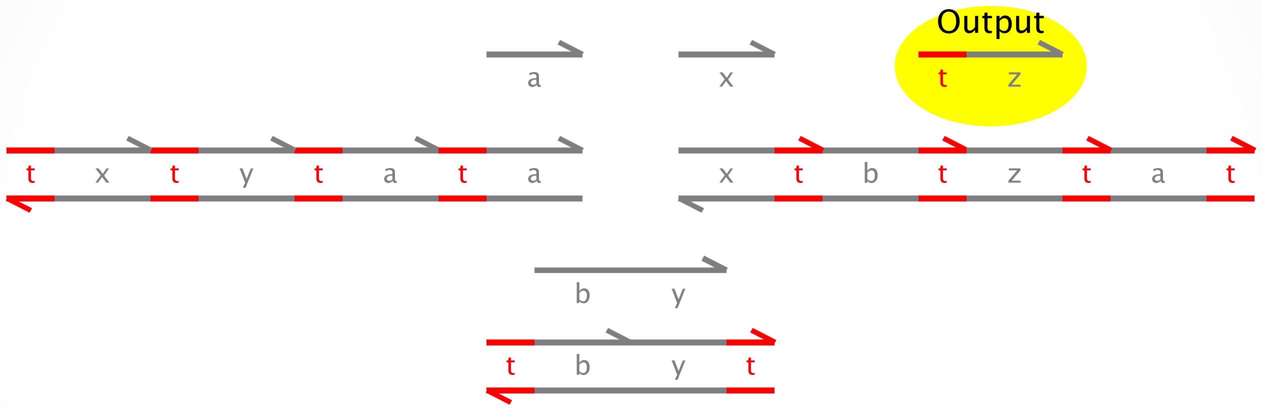
Join x+y→z



Join x+y→z



Join x+y→z



General n×m Join–Fork

- Easily generalized to 3+ inputs (with 2+ collectors) etc.
- Easily generalized to 2+ outputs (like Fork) etc.

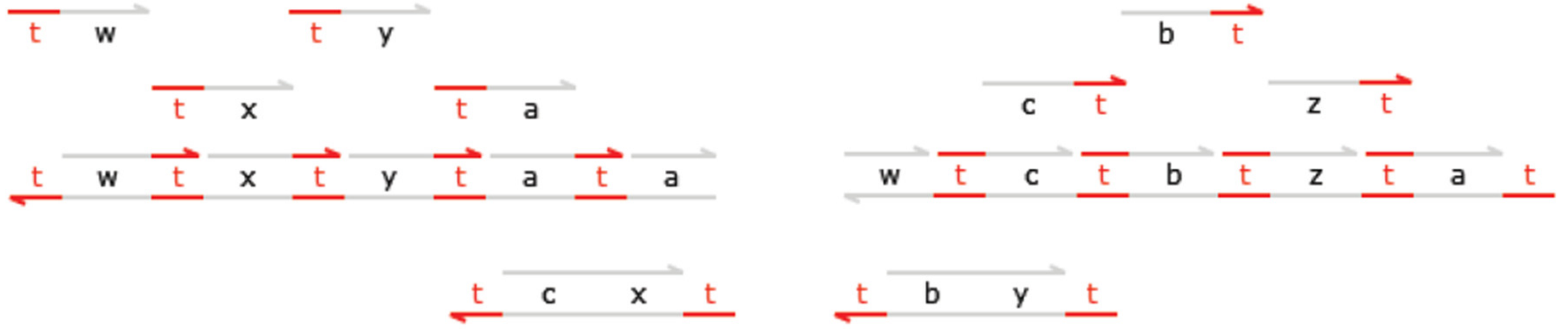


Figure 9: 3-Join $J_{wxyz} \mid tw \mid tx \mid ty \longrightarrow tz$: initial state plus inputs tw, tx, ty.

Strand Algebra

- An abstract description of signal–gate interactions:

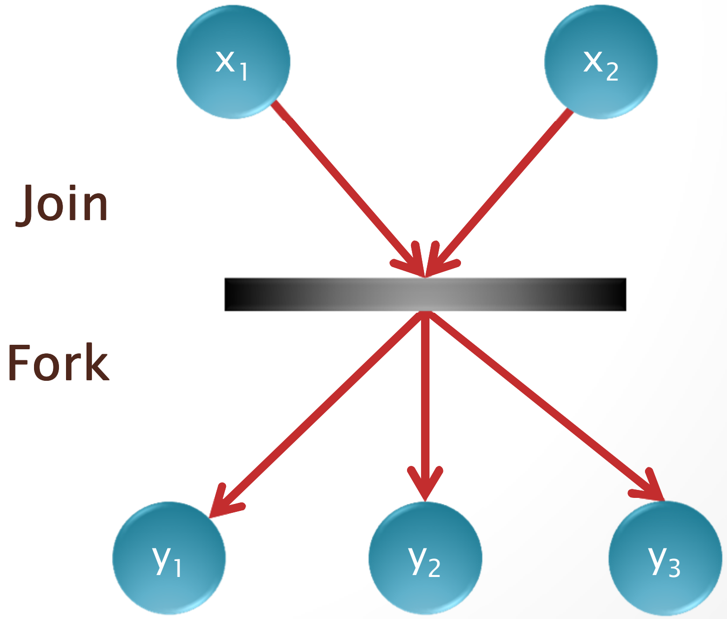
$$x_1 \mid .. \mid x_n \mid [x_1,..,x_n].[y_1,..,y_m] \rightarrow y_1 \mid .. \mid y_m$$

- Strand Algebra is an ‘intermediate language’
  - Four–three–two domain gates implement Strand Algebra.
  - Strand Algebra implements Boolean circuits, Petri Nets, FSA, Linear I/O Systems, Interacting Automata, etc.

- Two–domain gates implement Strand Algebra
  - N.B. this is a *conjecture*.

Petri Net Transitions

- Computing power equivalent to Petri Nets (not Turing complete).
- Not completely trivial: gates are consumed by activation, hence a persistent Petri net transition requires a stable population of gates.



Verification

Verification Issues

- Individual Components
  - Reversible reactions (infinite traces)
  - Interferences (deadlocks etc.) between copies of the same gate
  - Interferences (deadlocks etc.) between copies of different gates
  - Removal of active byproducts (garbage collection) is tricky

- Populations
  - Gates come in (large) populations
  - Each population *shares private domains* (technologically unavoidable)
  - Correctness of populations means proofs with large state spaces
  - Proofs about *arbitrary* population size?

- Environment
  - The nano-environment is stochastic (noise, failures, etc.)
  - Biology is messy
  - But we should al least make sure our designs are *logically correct*

Correctness

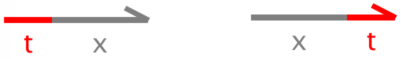
- The spec of a transducer: $T_{xy} + tx \rightarrow ty$
  - Is it true at all?
  - Is it true *possibly*, or *necessarily*, or *probabilistically (measure 1)?*
  - Is it true in the context of other *identical transducers?*
  - Is it true *in all possible contexts?*
  - Is it *(more)* true for large populations?
  - Is it true for infinite populations (continuous limit)?

Nick Algebra

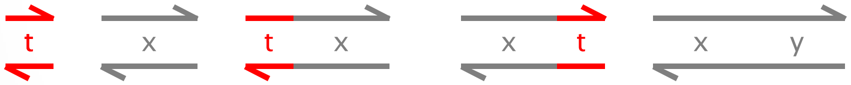
Nick Algebra

$S \ ::= \ t.x \ ⋮ \ x.t$	single strand
$\underline{D} \ ::= \ ø \ ⋮ \ \underline{t} \ ⋮ \ \underline{x} \ ⋮ \ \underline{t.x} \ ⋮ \ \underline{x.t} \ ⋮ \ \underline{x.x} \ ⋮ \ \underline{D^{\dagger}D}$	double strand
$U \ ::= \ S \ ⋮ \ \underline{D} \ ⋮ \ U \vert U \ ⋮ \ (\nu x)U$	soup

S



D



Algebraic Equality

= is an equivalence relation,
and a congruence over the term syntax

$$\underline{D}_1{}^{\underline{\dagger}}(\underline{D}_2\underline{{}^{\dagger}D}_3) \;=\; (\underline{D}_1\underline{{}^{\dagger}D}_2)\underline{{}^{\dagger}D}_3$$
$$\varnothing\underline{{}^{\dagger}D} \;=\; \underline{D{}^{\dagger}}\varnothing \;=\; \underline{D}$$

$$U_1|(U_2|U_3) \;=\; (U_1|U_2)|U_3$$
$$U_1|U_2 \;=\; U_2|U_1$$
$$\varnothing|U \;=\; U|\varnothing \;=\; U$$

$$(\nu x)U \;=\; (\nu y)(U\{y/x\}) \qquad \text{if } y \notin \mathit{pd}(U)$$
$$(\nu x)\varnothing \;=\; \varnothing$$
$$(\nu x)(U_1|U_2) \;=\; U_1|(\nu x)U_2 \qquad \text{if } x \notin \mathit{pd}(U_1)$$
$$(\nu x)(\nu y)U \;=\; (\nu y)(\nu x)U$$

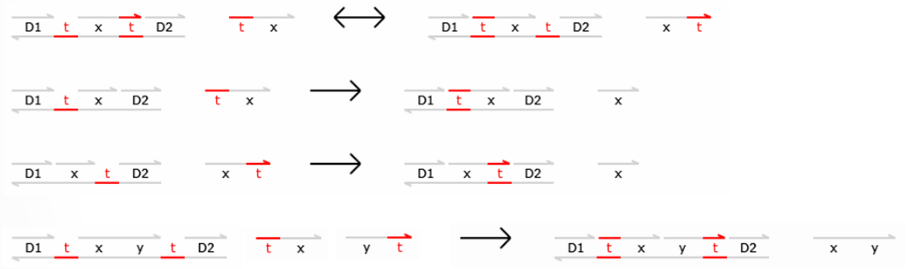
Reduction

$\underline{D_1{}^{\dagger}t^{\dagger}xt^{\dagger}D_2} \mid tx \;\leftrightarrow\; \underline{D_1{}^{\dagger}tx^{\dagger}t^{\dagger}D_2} \mid xt$ exchange

$\underline{D_1{}^{\dagger}t^{\dagger}x^{\dagger}D_2} \mid tx \;\rightarrow\; \underline{D_1{}^{\dagger}tx^{\dagger}D_2}$ left coverage

$\underline{D_1{}^{\dagger}x^{\dagger}t^{\dagger}D_2} \mid xt \;\rightarrow\; \underline{D_1{}^{\dagger}xt^{\dagger}D_2}$ right coverage

$\underline{D_1{}^{\dagger}t^{\dagger}xy^{\dagger}t^{\dagger}D_2} \mid tx \mid yt \;\rightarrow\; \underline{D_1{}^{\dagger}tx^{\dagger}yt^{\dagger}D_2}$ cooperation



$\underline{D} \;\rightarrow\; \varnothing$ if $\underline{D}$ not reactive waste

$U_1 \;\rightarrow\; U_2 \;\Rightarrow\; U_1 \mid U \;\rightarrow\; U_2 \mid U$ dilution

$U_1 \;\rightarrow\; U_2 \;\Rightarrow\; (\nu x)U_1 \;\rightarrow\; (\nu x)U_2$ isolation

$U_1 = U_2,\; U_2 \;\rightarrow\; U_3,\; U_3 = U_4 \;\Rightarrow\; U_1 \;\rightarrow\; U_4$ mixing

Reachability

- $U_1 \rightarrow^* U_2$ iff $U_1 \rightarrow \ldots \rightarrow U_2$
  - That is, U_1 *may* reduce to U_2.

- $U_1 \rightarrow^\forall U_2$ iff $\forall U, U_1 \rightarrow^* U \Rightarrow U \rightarrow^* U_2$
  - That is, U_1 *will* reduce to U_2. (It cannot avoid the possibility of reducing to U_2).

Gate Definitions

- $T_{xay} = \underline{t^{\dagger}xt^{\dagger}at^{\dagger}a} \mid ta \mid \underline{x^{\dagger}ty^{\dagger}ta^{\dagger}t} \mid yt$
- $T^{n}{}_{xy} = (\nu a)((T_{xay})^{n})$

- $F_{xayz} = \ldots$
- $F^{n}{}_{xyz} = (\nu a)((F_{xayz})^{n})$

- $J_{xyaz} = \ldots$
- $J^{n}{}_{xyz} = (\nu a)((J_{xyaz})^{n})$

Correctness

- Proposition: May-Correctness

  $T^n_{xy}|tx^n \rightarrow^* ty^n$

  $F^n_{xyz}|tx^n \rightarrow^* ty^n|tz^n$

  $J^n_{xyz}|tx^n|ty^n \rightarrow^* tz^n$

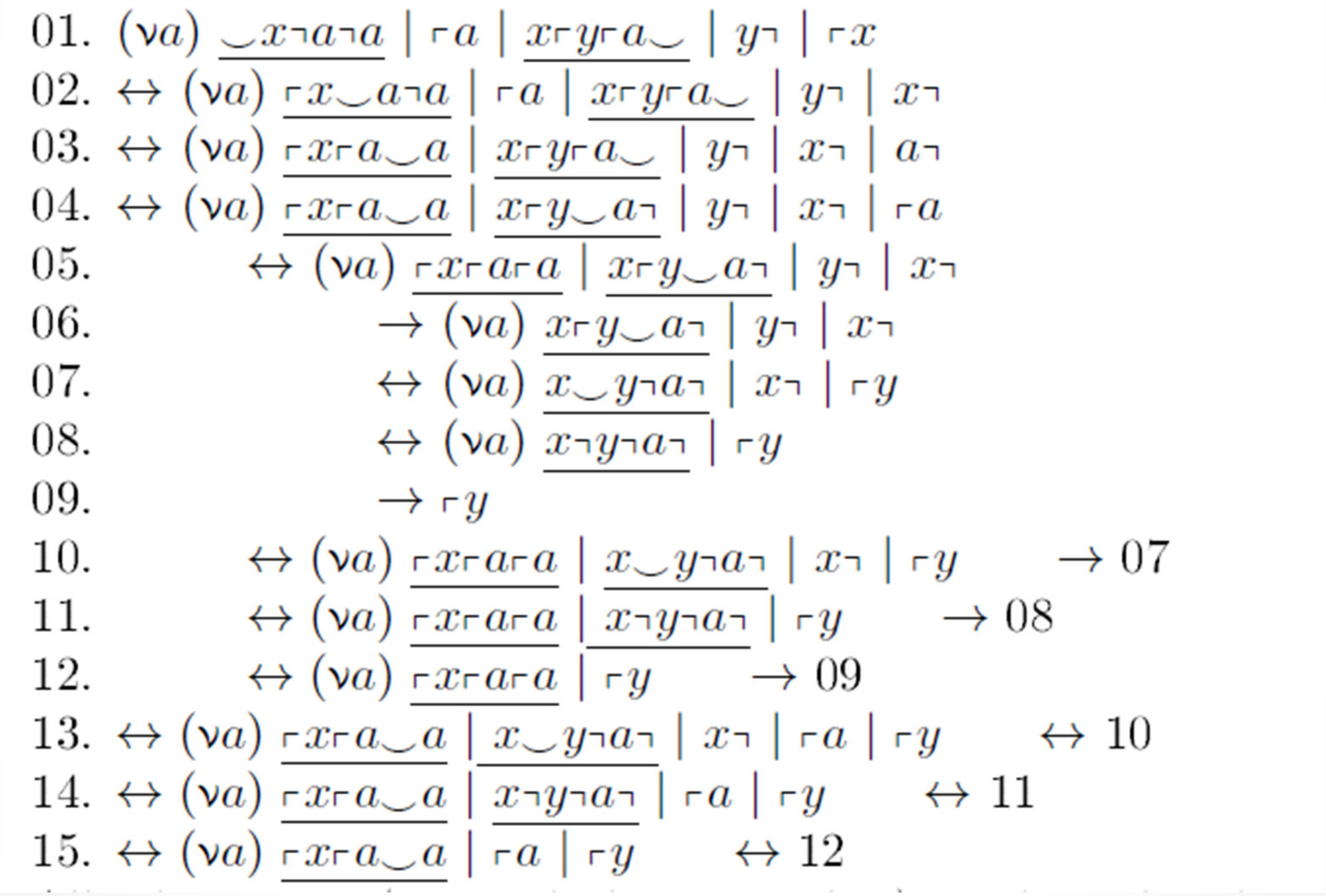
  - Easy case analysis and induction on n.

- Proposition: T^1_{xy} Will-Correctness

  $T^1_{xy} \mid tx \rightarrow^{\forall} ty$

  - Exhaustive case analysis enumerating all states of the system.
  - Can be done by hand for T^1_{xy}, and maybe T^2_{xy}, but not really for T^3_{xy} etc.
  - Will-correctness for fork/join is harder (more states).
  - Will-correctness for combinations of gates is harder
    (does not compose and requires analysis of joint state space).
  - We are using modelchecking to verify some of these properties.
    [Andrew Phillips & David Parker in PRISM]

$\mathrm{T}^1{}_{xy}$ Will-Correctness



01. $(\nu a)\ \underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \ulcorner x$

02. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\smile a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid x\urcorner$

03. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid x\urcorner \mid a\urcorner$

04. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner \mid \ulcorner a$

05. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner$

06. $\rightarrow (\nu a)\ \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner$

07. $\leftrightarrow (\nu a)\ \underline{x\smile y\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner y$

08. $\leftrightarrow (\nu a)\ \underline{x\urcorner y\urcorner a\urcorner} \mid \ulcorner y$

09. $\rightarrow \ulcorner y$

10. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\smile y\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner y \qquad \rightarrow 07$

11. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\urcorner y\urcorner a\urcorner} \mid \ulcorner y \qquad \rightarrow 08$

12. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \ulcorner y \qquad \rightarrow 09$

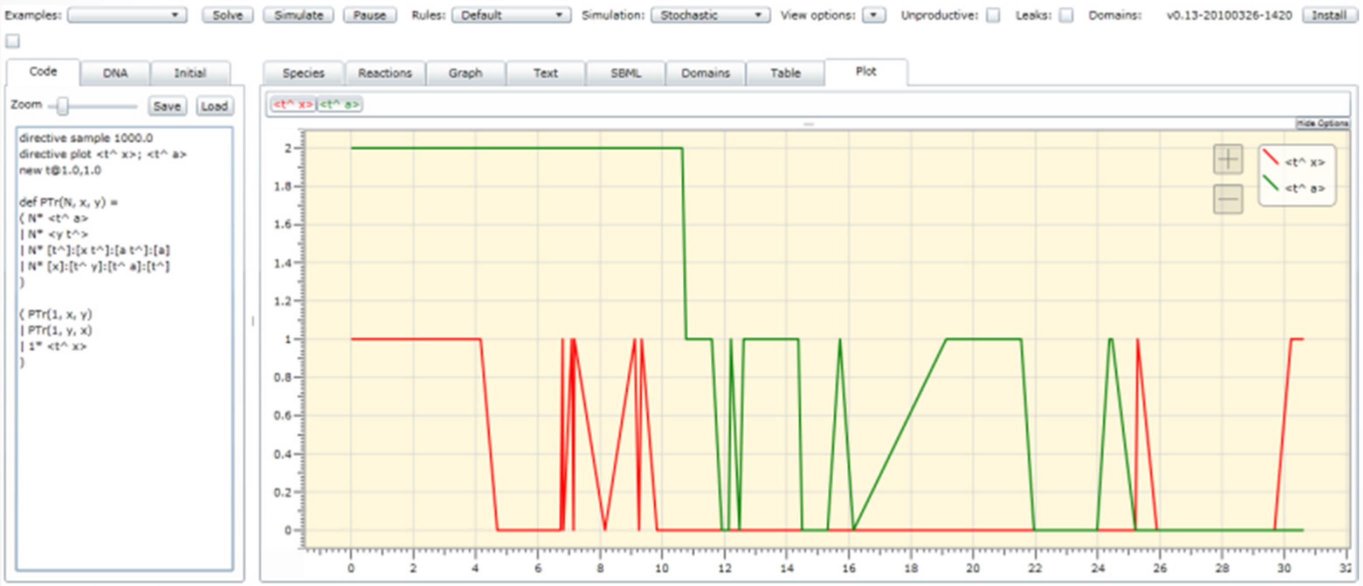
13. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\smile y\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner a \mid \ulcorner y \qquad \leftrightarrow 10$

14. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\urcorner y\urcorner a\urcorner} \mid \ulcorner a \mid \ulcorner y \qquad \leftrightarrow 11$

15. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \ulcorner a \mid \ulcorner y \qquad \leftrightarrow 12$

Interfering Transducers

- T_{xay} | T_{yax} *sharing the same 'a'*
    - ○ T^1_{xay} | T^1_{yax} | tx^1 Correct Run



Examples: Rules: Default Simulation: Stochastic View options: Unproductive: □ Leaks: □ Domains: v0.13-20100326-1420

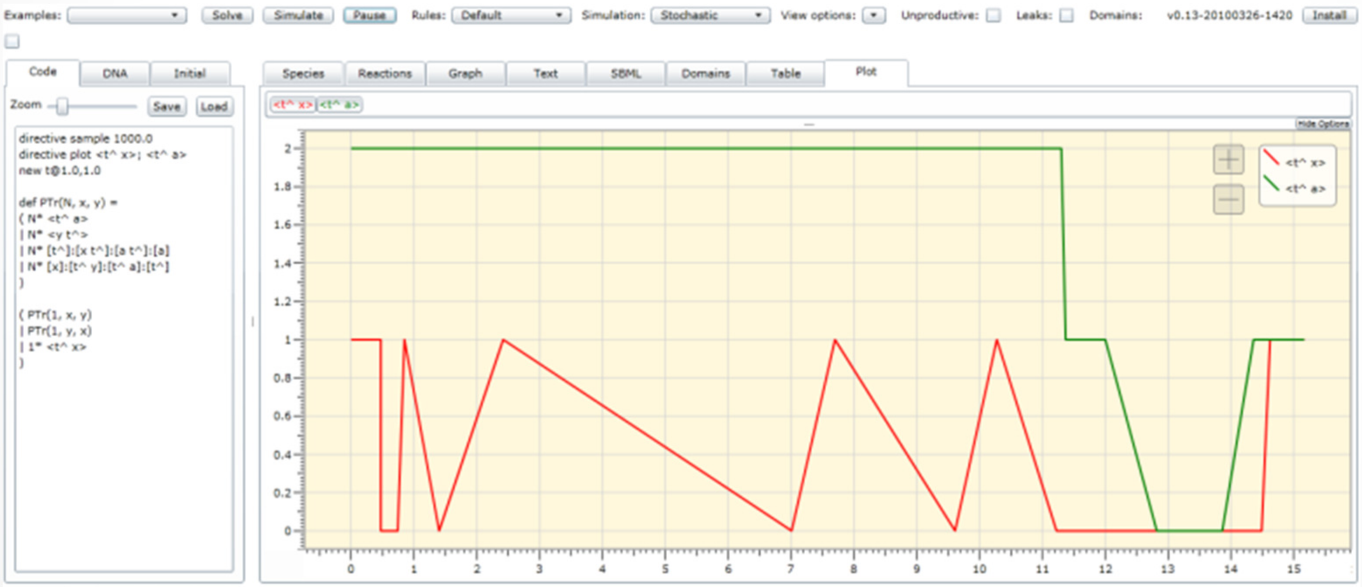
```
directive sample 1000.0
directive plot <t^ x>; <t^ a>
new t@1.0,1.0

def PTr(N, x, y) =
( N* <t^ a>
| N* <y t^>
| N* [t^]:[x t^]:[a t^]:[a]
| N* [x]:[t^ y]:[t^ a]:[t^]
)

( PTr(1, x, y)
| PTr(1, y, x)
| 1* <t^ x>
)
```

tx^1

ta^0

Interfering Transducers

- T_{xay} | T_{yax} *sharing the same 'a'*
  - ○ T^1_{xay} | T^1_{yax} | tx^1 Incorrect Run



Rules: Default

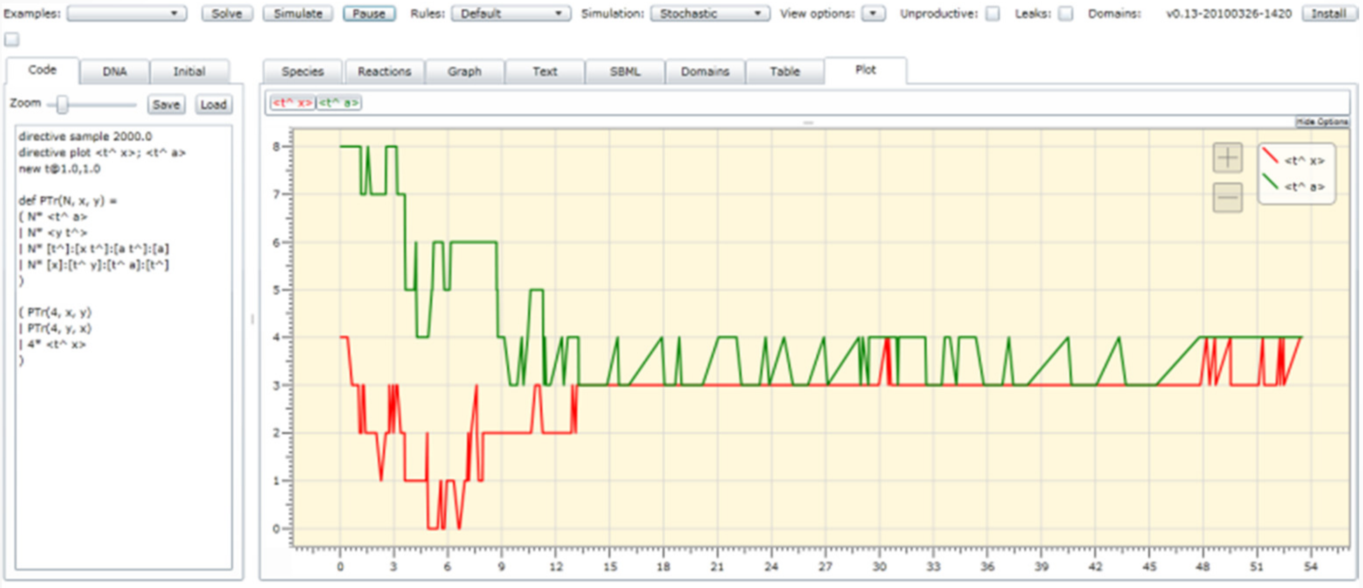
Simulation: Stochastic

```
directive sample 1000.0
directive plot <t^ x>; <t^ a>
new t@1.0,1.0

def PTr(N, x, y) =
( N* <t^ a>
| N* <y t^>
| N* [t^]:[x t^]:[a t^]:[a]
| N* [x]:[t^ y]:[t^ a]:[t^]
)

( PTr(1, x, y)
| PTr(1, y, x)
| 1* <t^ x>
)
```

tx^1 ta^1

Interfering Transducers

- T_{xay} | T_{yax} *sharing the same 'a'*
  - ○ T^4_{xay} | T^4_{yax} | tx^4 4 copies all 'badly interfering' (rare case)



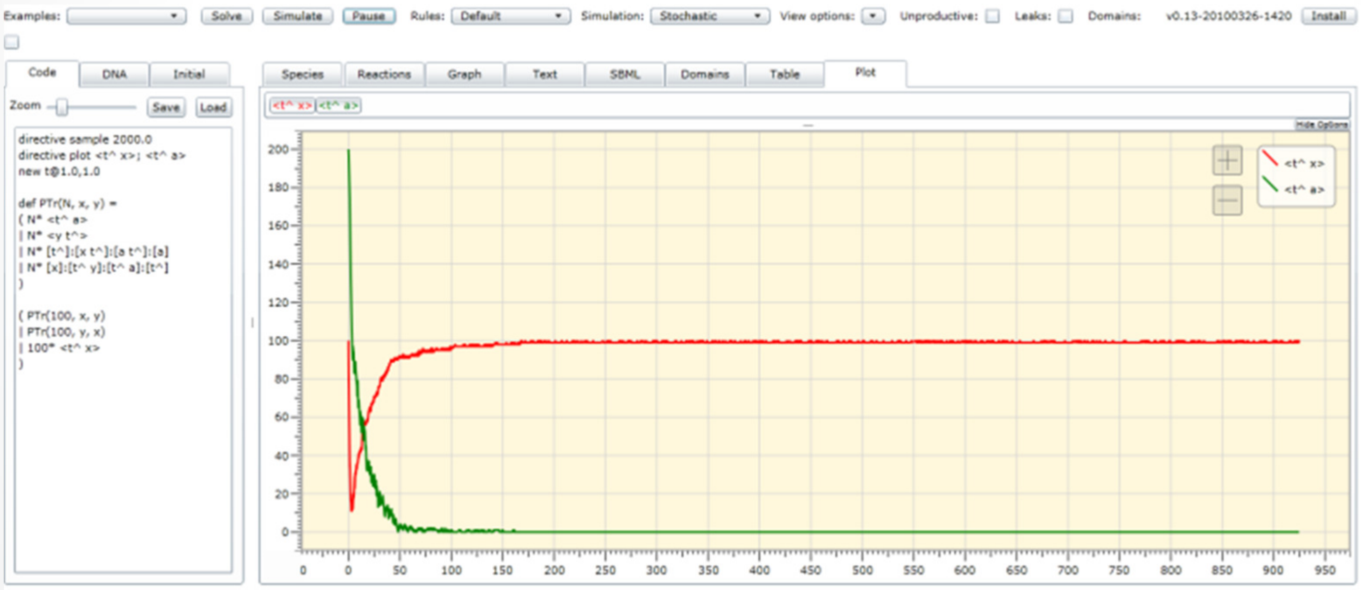
Rules: Default
Simulation: Stochastic
v0.13-20100326-1420

```
directive sample 2000.0
directive plot <t^ x>; <t^ a>
new t@1.0,1.0

def PTr(N, x, y) =
( N* <t^ a>
| N* <y t^>
| N* [t^]:[x t^]:[a t^]:[a]
| N* [x]:[t^ y]:[t^ a]:[t^]
)

( PTr(4, x, y)
| PTr(4, y, x)
| 4* <t^ x>
)
```

tx^4 ta^4

Interfering Transducers

- T_{xay} | T_{yax} *sharing the same 'a'*
  - ○ T^{100}_{xay} | T^{100}_{yax} | tx^{100} 100 copies



Rules: Default Simulation: Stochastic Unproductive: □ Leaks: □ Domains: v0.13-20100326-1420

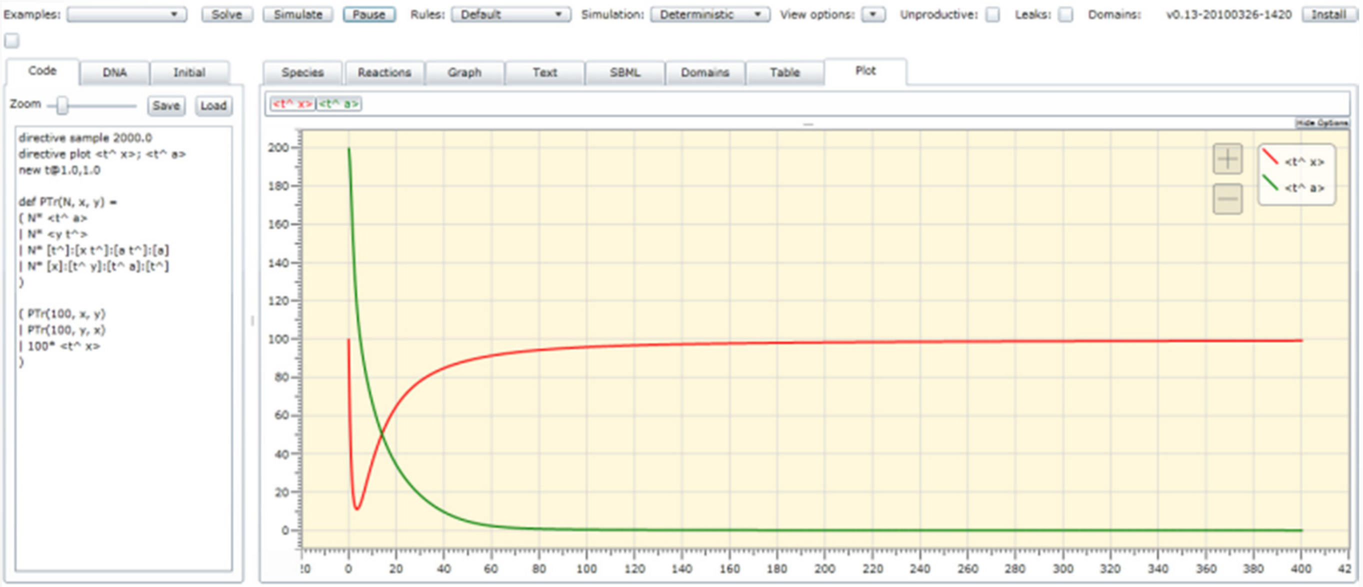
```
directive sample 2000.0
directive plot <t^ x>; <t^ a>
new t@1.0,1.0

def PTr(N, x, y) =
( N* <t^ a>
| N* <y t^>
| N* [t^]:[x t^]:[a t^]:[a]
| N* [x]:[t^ y]:[t^ a]:[t^]
)

( PTr(100, x, y)
| PTr(100, y, x)
| 100* <t^ x>
)
```

tx^{100}

ta^{0}

Interfering Transducers

- T_{xay} | T_{yax} *sharing the same 'a'*
  - ○ T^{100}_{xay} | T^{100}_{yax} | tx^{100} continuous limit (ODE simulation)



Rules: Default

Simulation: Deterministic

v0.13-20100326-1420

```
directive sample 2000.0
directive plot <t^ x>; <t^ a>
new t@1.0,1.0

def PTr(N, x, y) =
( N* <t^ a>
| N* <y t^>
| N* [t^]:[x t^]:[a t^]:[a]
| N* [x]:[t^ y]:[t^ a]:[t^]
)

( PTr(100, x, y)
| PTr(100, y, x)
| 100* <t^ x>
)
```

tx^{100}

ta^{0}

Interfering Transducers

- Although $\mathrm{T}_{\mathrm{xay}} \mid \mathrm{T}_{\mathrm{yax}} \mid \mathrm{tx} \not\rightarrow^{\forall} \mathrm{tx}$
- We have $\mathrm{T}_{\mathrm{xay}} \mid \mathrm{T}_{\mathrm{yax}} \mid \mathrm{tx} \mid \mathrm{ty} \rightarrow^{\forall} \mathrm{tx} \mid \mathrm{ty}$
- That means that a large population of such gates in practice does not deadlock easily: each pair of deadlocked gates can be unblocked by another pair correctly producing a ty as an intermediate product.
- Wisdom of the masses: individuals can be wrong, but the population is right. It is very unlikely that a significant fraction of gates ends up being deadlocked.

Conclusions

- A new architecture for general DNA gates
  - Simple signals, simple gate structures.
  - Self-cleaning: no garbage left by operation (except inert).
  - Enabling new ways of assembling gates.
  - Some experimental evidence that it works.

- A correspondingly simple algebra
  - For verifying gate designs mechanically.

- Verification issues
  - Verification techniques for gate populations.
  - Are the fork/join gates in Nick Algebra a correct implementation of (Strand Algebra and) Petri nets?