Supporting Information: Automated leak analysis of nucleic acid circuits

Iuliia Zarubiieva[1], Carlo Spaccasassi[2], Vishwesh Kulkarni[*,3], Andrew Phillips[*,2]

[1]University of Warwick, Coventry, CV4 7AL, UK
[2]Microsoft Research, Cambridge, CB1 2FB, UK
[3]AiM Macromed Ltd, London, WC2H 9JQ, UK
Email: *vishwesh@aimacromed.com; *andrew.phillips@live.com

S1 Logic DSD method

We briefly summarise the syntax and semantics of Logic DSD, as defined previously.[1] We represent a nucleic acid system in our language as a *process*, which is defined as a multiset of nucleic acid strands (Table S1). As described in the main text, the Visual DSD language relies on the notion of a *domain*, which denotes a unique nucleic acid sequence. We represent a domain by a lower-case variable and its Watson-Crick *complement* by appending the (`*`) character to the domain. For example, `x*` denotes the complement of `x`. We assume that a domain can only bind to its complement and indicate that two complementary domains are bound using the notation `x!i` and `x*!i`, where `i` is a unique identifier called a *bond*, which can be either a variable name or an integer. A *toehold* denotes a domain that is short enough to spontaneously unbind from its complement and is represented by appending the (`^`) character to the domain. If a domain is a logical variable, indicated by an upper-case letter X, the variable will be substituted by a concrete domain during logical inference. The *wildcard* "_" is the logical variable that matches any term. We define a *site* s as a domain that is either bound or free, and a *sequence* S as a non-empty sequence of sites, ordered from the 5' end to the 3' end. A process P is a multiset of strands $<S_1> \mid \ldots \mid <S_N>$, separated by the parallel composition operator ($\mid$), where each strand is a sequence enclosed in angle brackets. In addition, we define a *species* as a set of strands bound to each other such that they form a connected component. For convenience, we define additional syntactic sugar that allows a species to be enclosed in square brackets and preceded by a constant, which denotes the species population.

The pattern $< S >$ matches a strand with exactly sequence S, while the pattern $< S$ matches a strand with sequence S at its 5' end, and the pattern $S >$ matches a strand with sequence S at its 3' end. The pattern S matches a sequence that can be present anywhere in a process, and the pattern $S_1 > \mid < S_2$ matches a nick between two adjacent strands, where $S_1 >$ and $< S_2$ represent the two ends of the strands where the nick occurs. The empty pattern $\emptyset$ does not match any strand, and is used to model the creation and deletion of strands.

A pattern π is matched with a process P using the notion of a *context* C_N (Table S1), defined as a "process with N holes",[2] where each hole $[\cdot]_i$ in C_N is associated with a number $i \in N$. The matching is performed by *applying* a context C_N to patterns $\pi_1 ... \pi_N$, written $C_N[\pi_1]...[\pi_N]$. This fills each numbered hole $[\cdot]_i$ in C_N with the corresponding pattern π_i. For example, applying the context $C_2 =< d_1 d_2 [\cdot]_2 \mid < d_4 [\cdot]_1 d_6 >$ to the patterns $\pi_1 = d_5$ and $\pi_2 = d_3 >$ is written $C_2[d_5][d_3 >]$ and results in the process $< d_1 d_2 d_3 > \mid < d_4 d_5 d_6 >$.

To allow general logic predicates to be defined, our language embeds these patterns and contexts in a general logic programming language, by extending the standard syntax of Prolog (Table S1). As in Prolog, the main data structure is a *term* T, which can be an integer *int*, floating point number $float$, a *string* enclosed in double quotes, or a logical variable X. In addition, a term can be a pattern π or a context C_N applied to N patterns, written $C_N[\pi_1]...[\pi_N]$. The context itself can also be a logical variable X, written $X[\pi_i]...[\pi_N]$, which is used to match any process that contains the patterns π_1 ... π_N. The *structure* $x(T_1, ...T_N)$ bundles together N terms under an identifier x and defines named compound terms such as $compl(d_1, d_2)$. The list $[T_1; \ldots; T_N]$ denotes a possibly empty list of terms T_i. It is also possible to define lists of undetermined length using the syntax $[T_1; \ldots; T_N \# X]$, where the logical variable X stands for the rest of the list. An *atomic predicate* or *atom* A has syntax $x(t_1, ...t_N)$, where x is an identifier. It is generally used to describe properties or assign relationships to terms. We assume that the sets of predicate identifiers and structure identifiers are disjoint. A *literal* L is either an atom A or a *negative atom* `not`A. A *Horn clause* or *clause* H has syntax $A :- L_1, ..., L_N$, and can be read "A holds *if* L_1 ... L_N hold". The list of literals L_i is possibly empty. Atom A is called the *head* of H, and the list of literals is its *body*. A clause with no literals is also known as a *fact*. Finally, a *logic program* is a set of Horn clauses.

d ::=	$x \mid x^* \mid x\hat{} \mid x\hat{}^* \mid X$	Domain
k ::=	$x \mid int \mid X$	Bond, $int \geq 0$
t ::=	$int \mid string \mid x \mid x(T_1, \ldots, T_N)$	Tag, $N \geq 1, int \geq 0$
l ::=	$x \mid int \mid X$	Location, $int \geq 0$
s ::=	$d\{t\}@l \mid d\{t\}!k@l \mid X$	Site, with $\{t\}$ and $@l$ optional
S ::=	$s_1 \ldots\ s_N$	Sequence of sites, $N \geq 1$
P ::=	$<S_1> \mid \ldots \mid <S_N>$	Process, $N \geq 0$
π ::=	$<S> \mid <S \mid S \mid S> \mid S_1>\mid<S_2 \mid \emptyset$	Pattern
C_N ::=	$[\cdot]_i \mid P \mid <S\ C_N \mid S\ C_N \mid C_N\ S> \mid C_N \mid C_N$	Context with N holes, $1 \leq i \leq N$
T ::=	$X \mid int \mid float \mid string \mid \pi \mid C_N[\pi_1]\ldots[\pi_N] \mid X[\pi_1]\ldots[\pi_N]$ $\mid x(T_1, \ldots, T_N) \mid [T_1; \ldots; T_N] \mid [T_1; \ldots; T_N \# X]$	Term
A ::=	$x(T_1, \ldots, T_N)$	Atomic predicate
L ::=	$A \mid \texttt{not}\ A$	Literal
H ::=	$A :- L_1, \ldots, L_N$	Horn clause

Table S1: Syntax of processes, patterns and logic programs in Logic DSD.

```
directive rules {
bind(P1, P2, Q, D!i) :-
  P1 = C1[D], P2 = C2[D'], compl(D, D'),
  Q = C1[D!i] | C2[D'!i], freshBond(D!i,P1|P2).
slow([P1;P2], "kt",Q) :- bind(P1,P2,Q,_).
displace(P, Q, E!j, D!i) :-
  P = C [E!j D] [D!i] [D'!i E'!j],
  Q = C [E!j D!i] [D] [D'!i E'!j].
fast([P],"displace",Q) :- displace(P,Q,_,_).
displaceL(P,Q,E!j,D!i) :-
  P = C [E'!j D'!i] [D E!j] [D!i],
  Q = C [E'!j D'!i] [D!i E!j] [D].
fast([P],"displace",Q) :- displaceL(P,Q,_,_).
unbind(P, Q, D!i) :-
  P = C [D!i] [D'!i], toehold(D),
  Q = C [D]  [D'], not adjacent(D!i,_,P).
adjacent(D!i, E!j, P) :- P = C [D!i E!j] [E'!j D'!i].
adjacent(D!i, E!j, P) :- P = C [E!j D!i] [D'!i E'!j].
fast([P],"unbind",Q) :- unbind(P,Q,_).
```

```
reaction([P1;P2],Rate,R) :-
  slow([P1;P2],Rate,Q), merge(Q,R,[(P1|P2);Q]).
reaction([P],Rate,R) :-
  slow([P],Rate,Q), merge(Q,R,[P;Q]).
merge(P,P,V) :- not fast([P],_,_).
merge(P,R,V) :-
  fast([P],_,Q), not member(Q,V), merge(Q,R,[Q#V]).
}
directive parameters [kt = 0.001]
def X() = <t^ x>
def Y() = <t^ y>
def Catalysis() =
( 1000* [<x!1 u^!2> | <y!3 u^!4>
      | <u^*!4 y*!3 u^*!2 x*!1 t^*>]
| 1000*[<t^!1 x!2> | <u^* x*!2 t^*!1>]
| 1000*[<t^!1 y!2> | <u^* y*!2 t^*!1>]
| 1000*<u^ y> )
( X() | Catalysis() )
```

Figure S1: Visual DSD program code for the Catalysis circuit from Figure 1.

S2 DSD Leaks methodology

```
directive rules {
nonLeaked(P) :-
  catalysis2D(L),
  member(P,L).
catalysis2D([
<t^ x>
; <u^*!0 y*!1 u^*!2 x*!3 t^*{complementarity("c")}>
| <x!3 u^!2> | <y!1 u^!0>
; <u^* x*!0 t^*!1> | <t^!1 x!0>
; <x u^>
; <u^ y>
; <t^!0 x!1> | <u^*!2 y*!3 u^* x*!1
   t^*{complementarity("c")}!0> | <y!3 u^!2>
; <u^* y*!0 t^*!1> | <t^!1 y!0>
; <u^*!0 x*!1 t^*> | <x!1 u^!0>
; <u^* y*!0 u^*!1 x*!2 t^*{complementarity("c")}!3>
| <t^!3 x!2> | <u^!1 y!0>
; <y u^>
; <u^*!0 y*!1 t^*> | <y!1 u^!0>
; <t^ y>
]).

bind(P1, P2, Q, D!i) :-
  P1 = C1[D], P2 = C2[D'], compl(D, D'),
  Q = C1[D!i] | C2[D'!i], freshBond(D!i,P1|P2).
slow([P1;P2], Rate,Q) :-
  bind(P1,P2,Q,D!i), find(D, "bind", Rate).
displace(P, Q, E!j, D!i) :-
  P = C [E!j D] [D!i] [D'!i E'!j],
  Q = C [E!j D!i] [D] [D'!i E'!j].
fast([P],"displace",Q) :- displace(P,Q,_,_).
displaceL(P,Q,E!j,D!i) :-
  P = C [E'!j D'!i] [D E!j] [D!i],
  Q = C [E'!j D'!i] [D!i E!j] [D].
fast([P],"displace",Q) :- displaceL(P,Q,_,_).
unbind(P, Q, D!i) :-
  P = C [D!i] [D'!i], toehold(D),
  Q = C [D]  [D'], not adjacent(D!i,_,P).
adjacent(D!i, E!j, P) :-
  P = C [D!i E!j] [E'!j D'!i].
adjacent(D!i, E!j, P) :-
  P = C [E!j D!i] [D'!i E'!j].
fast([P],"unbind",Q) :- unbind(P,Q,_).

leak(P1,P2,Q,D!i) :- leak2(P1, P2, Q, D!i).
leak(P1,P2,Q,D!i) :- leak2(P2, P1, Q, D!i).
leak2(P1,P2,Q,D!i) :-
  P1 = C1[D!i][D'!i], P2 = C2[D],
  not clamped(D!i,P1),
  Q = C1[D][D'!i] | C2[D!i].
clamped(D!i,P):-
  P = C [D1!i1 D!i D2!i2] [D2'!i2 D'!i D1'!i1].
//First order leaks
//slow([P1;P2], "leak",Q) :-
//  nonLeaked(P1), nonLeaked(P2), leak(P1,P2,Q,_).
slow([P1;P2], "leak",Q) :- leak(P1,P2,Q,_).

cover(P,Q,E!j,D!i) :-
  P = C [E!j D] [D' E'!j], compl(D, D'),
  Q = C [E!j D!i] [D'!i E'!j], freshBond(D!i,P).
fast([P],"cover",Q) :- cover(P,Q,_,_).
coverL(P,Q,E!j,D!i) :-
  P = C [D E!j] [E'!j D'], compl(D, D'),
  Q = C [D!i E!j] [E'!j D'!i], freshBond(D!i,P).
fast([P],"cover",Q) :- coverL(P,Q,_,_).

migrate(P, Q, E!j, D!i1) :-
  P = C [E!j D!i1] [D'!i2 E'!j] [D'!i1] [D!i2],
  Q = C [E!j D!i1] [D'!i1 E'!j] [D'!i2] [D!i2].
fast([P],"migrate",Q) :- migrate(P,Q,_,_).

normalize(R, R) :- not displaceL(R,_,_,_).
normalize(P, R) :-
  displaceL(P,Q,D!i,E!j), normalize(Q,R).

reaction([P1; P2], Rate, Rn) :-
  slow([P1;P2], Rate, Q), merge(Q, R), normalize(R, Rn).
reaction([P], Rate, Rn) :-
  slow([P], Rate, Q), merge(Q, R), normalize(R, Rn).

condense(Q, G) :- condense2([Q], [], G).
condense2([], _, []).
condense2([Q # ToVisit], V, [[Q; Rs] # G']) :-
  findAll(R, fast([Q], _, R), Rs),
  difference(Rs,[Q # V], Rs'),
  difference(Rs', ToVisit, Rs''),
  append(ToVisit, Rs'', ToVisit'),
  condense2(ToVisit', [Q # V], G').

merge(Q, R) :-
  condense(Q,G), tscc(G,TS), member([R#_],TS).
merge(Q, Q) :- not fast([Q], _,_).

rate(_{complementarity(X)}, "bind", X).
rate(t^, "bind", "kt").
rate(u^, "bind", "kt").

find(D,Type,Rate) :- rate(D,Type,Rate).
find(D,Type,Rate) :-
  not rate(D,Type,_), compl(D, D'), rate(D',Type,Rate).
}

directive parameters [kt = 0.001;
     c = 0.0000008; leak = 1E-09;]
def X() = <t^ x>
def Y() = <t^ y>
def Catalysis() =
( 1000* [<x!1 u^!2> | <y!3 u^!4>
| <u^*!4 y*!3 u^*!2 x*!1 t^*{complementarity("c")}>]
| 1000*[<t^!1 x!2> | <u^* x*!2 t^*!1>]
| 1000*[<t^!1 y!2> | <u^* y*!2 t^*!1>]
| 1000*<u^ y> )
( X() | Catalysis() )
```

Figure S2: Visual DSD program code for the Catalysis circuit from Figure 2 with leaks.

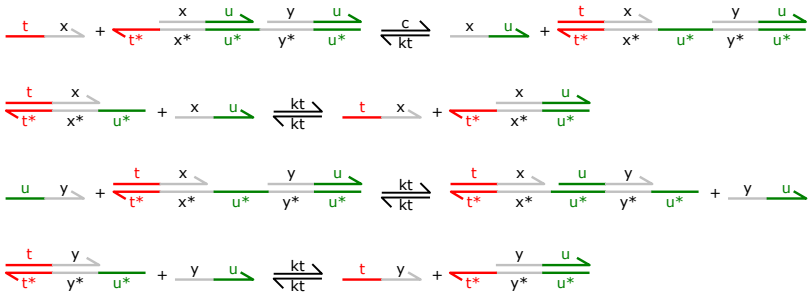
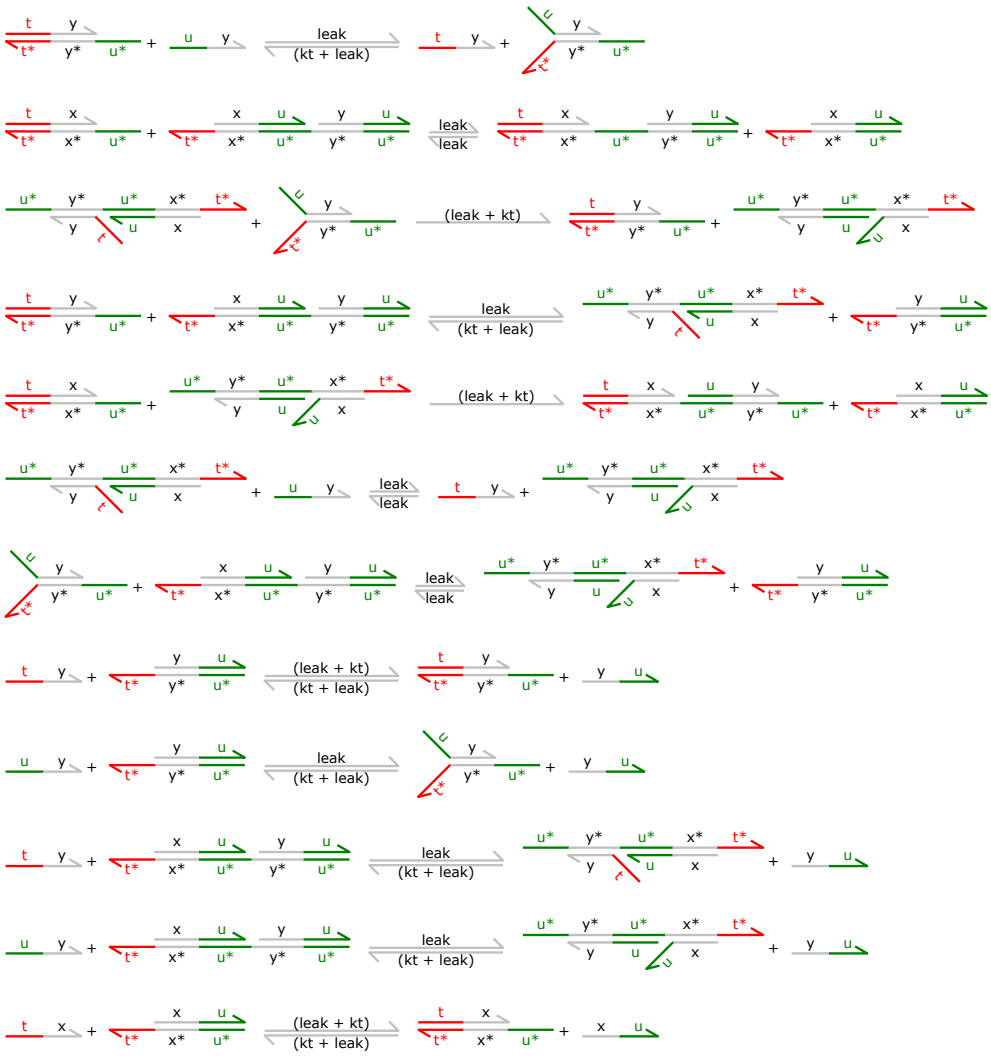
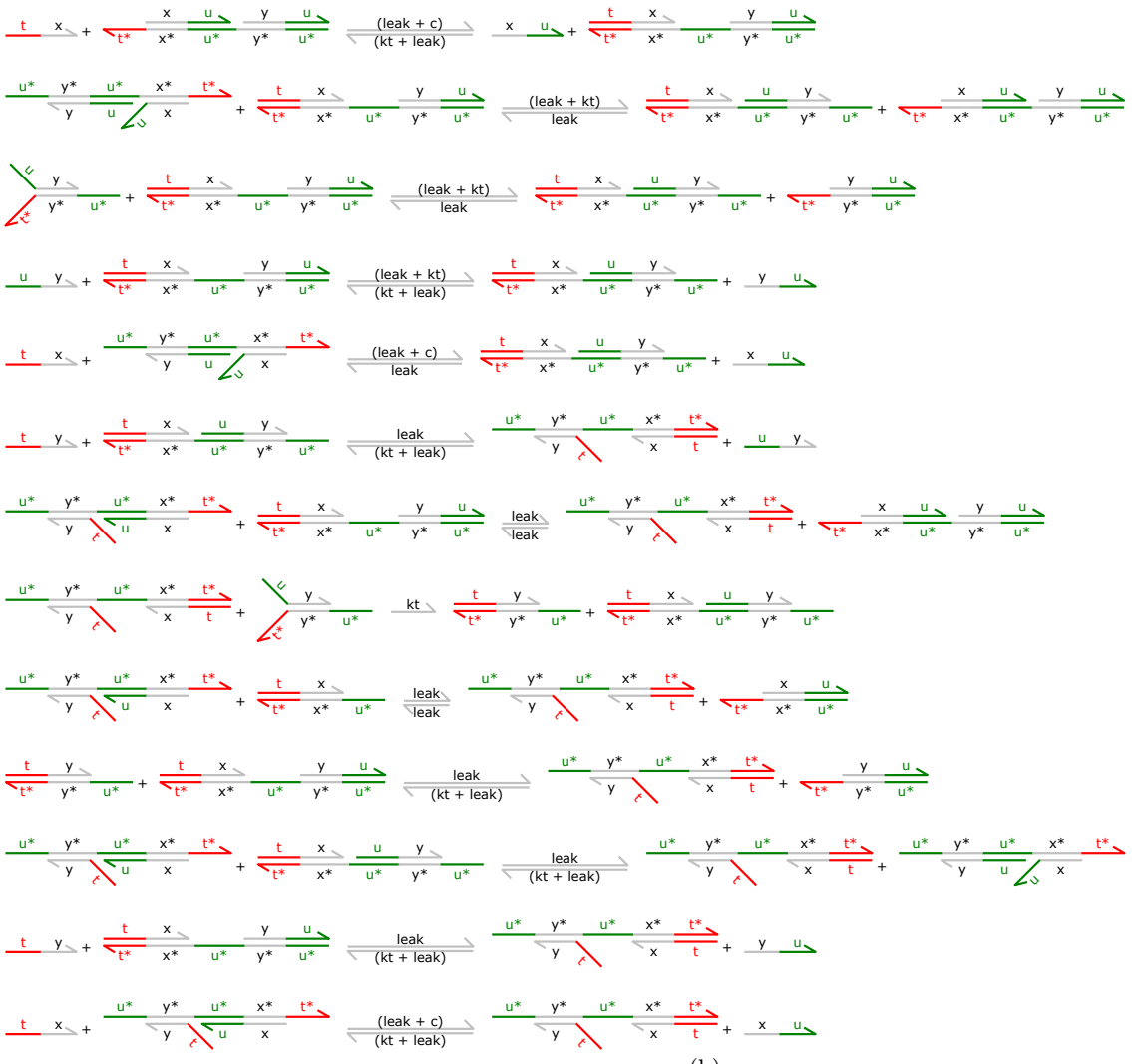


Figure S3: CRN for no-leak 2-domain catalysis circuit without caps from Figure 2.



(a) CRN for full leak 2-domain catalysis circuit without caps from Figure 2 (continued on next page).



(b)

Figure S4: CRN for full leak 2-domain catalysis circuit without caps from Figure 2.

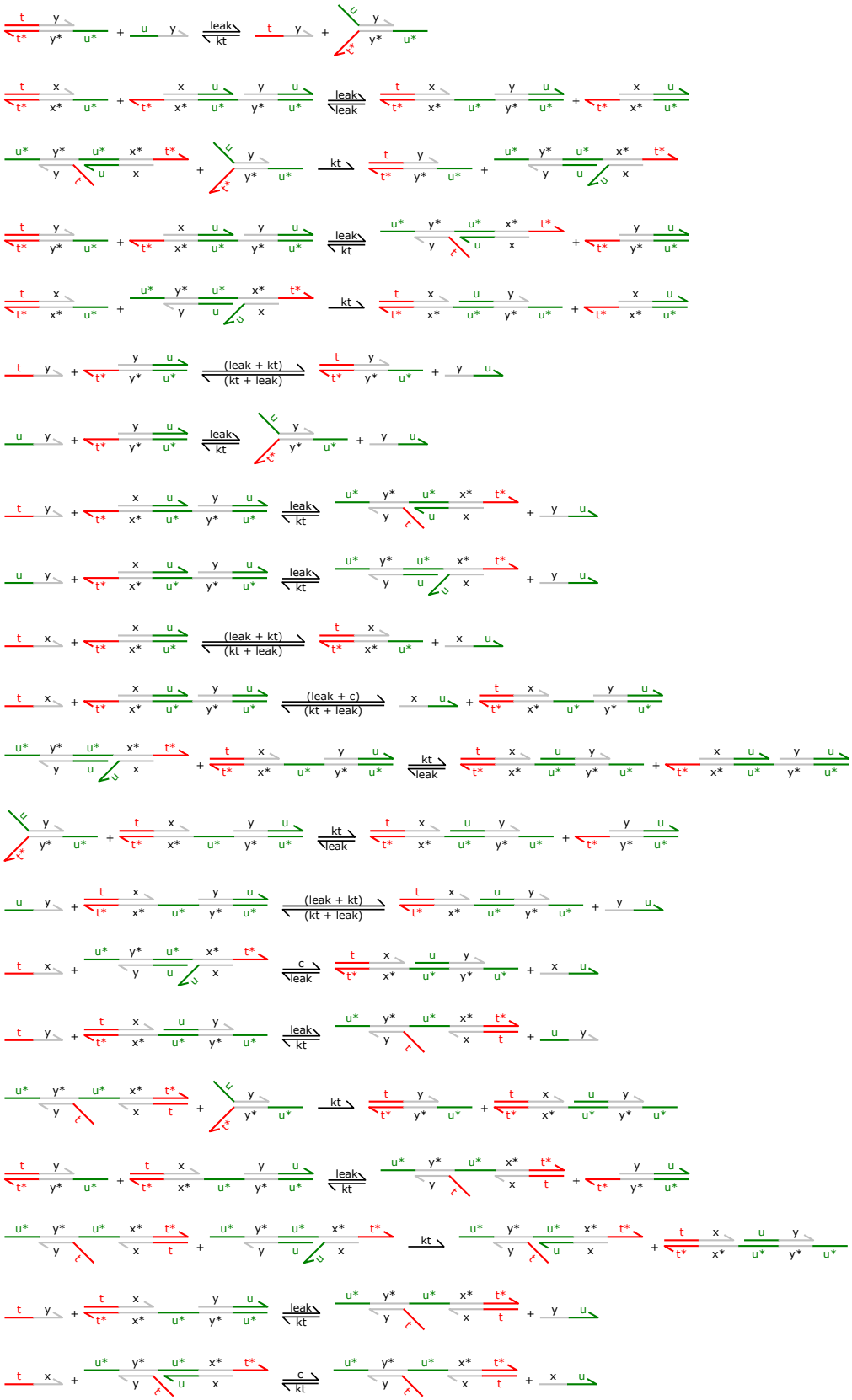


Figure S5: CRN for first order leak 2-domain catalysis circuit without caps from Figure 2.

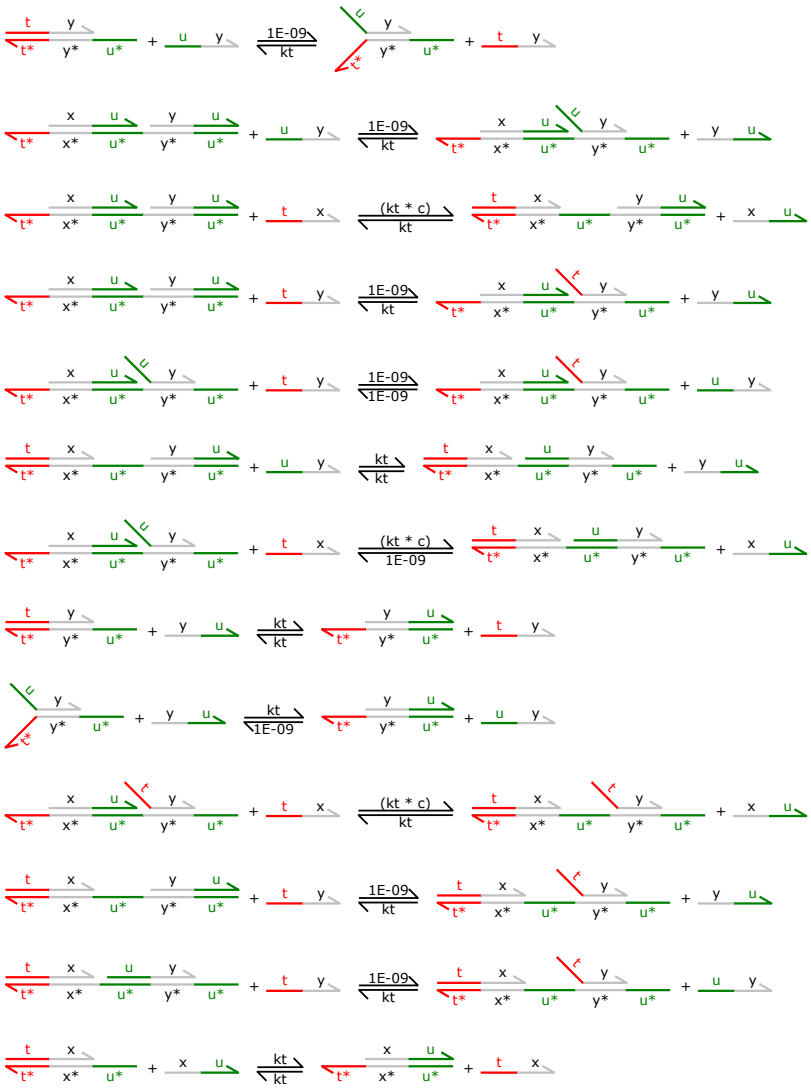


Figure S6: CRN for classic leaks 2-domain catalysis circuit without caps from Figure 2.

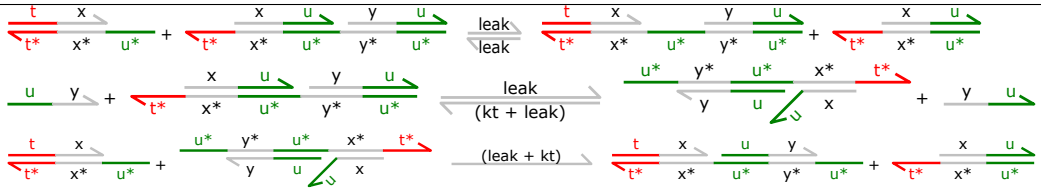
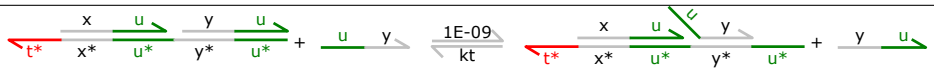
	Critical reactions	Contribution
A		33.6%
		33.6%
		23.9%
B		40.9%

Figure S7: **Critical leak reactions for the Catalysis circuit from Figure 2 implemented with different hypotheses:** (**A**) full leak and first order leak, and (**B**) classic leak. Critical leaks for the full leak and first order leak approaches are the same. The effect of disabling each reaction is shown as "Contribution". Other reactions present in the circuits have the contribution below 1% and therefore are considered insignificant.

S2.1 Use of DSD leak semantics with various secondary structures

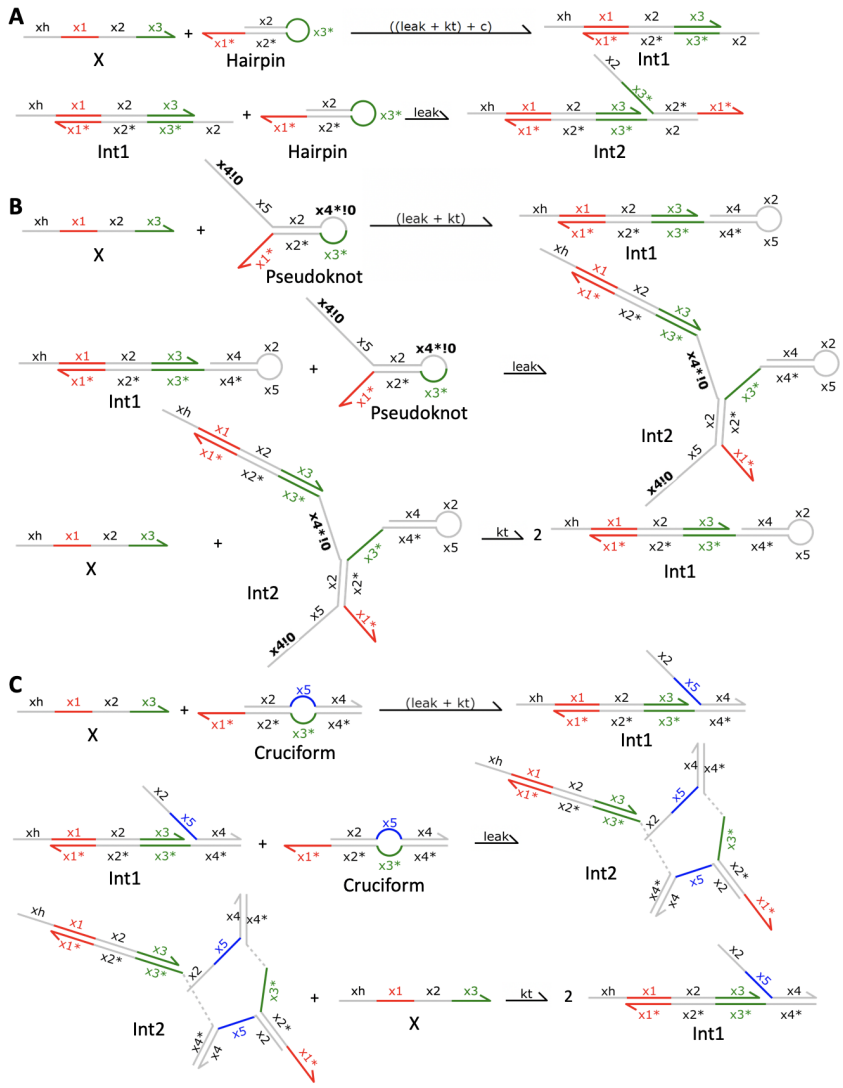


Figure S8: **Illustration of automated leak generation for different DNA conformations.** (**A**) hairpin. (**B**) pseudoknot, where the bond `!0` signifies the binding of corresponding domains `x4` and `x4*`. (**C**) cruciform.

S2.2 Automated detection of the most critical leak reactions

The leak semantics can generate a considerable number of reactions even from a relatively small circuit. Including only first order leaks can significantly reduce the number of low probability reactions. However, even such reduced CRNs may still contain a number of reactions whose effects are negligible. In order to analyse the resulting CRN and evaluate a given leak mitigation strategy, it is important to determine the leaks that are most significant. This can be achieved by manually disabling one or several leak reactions at a time and observing the change in output. However, doing this manually is time-consuming and unsuitable for larger circuits. Here, we present a technique to perform such sensitivity analysis as an automated workflow designed in MATLAB and the SimBiology toolbox.

The workflow uses the CRN file generated by the leak semantics with a few modifications. Firstly, the file should contain only the reactions without the constants or other additional lines of code. Secondly, it is preferable to remove the separator "|" before each reaction, which can be easily done using the "Replace" function. After this, the constants and initial conditions can be specified directly in the .mlx file. The CRN code saved as a .txt file is read and pre-processed. Next, the reaction rates, reactions and species are extracted and saved as separate variables. Using the specific commands of the SimBiology toolbox, it is possible to construct a model object and then associate the reactions, species and rates. The toolbox can construct the ODE model where the equation for each species consists of the reaction fluxes in which this species participates. If the reaction contributes to the species production, i.e. the species is on the right-hand side of the reaction, it will appear with a "+" sign if it contributes to its consumption, i.e. the species is on the left-hand side, – it will appear with "-". For example, consider species sp_0 from the leak model of the 4-domain catalysis circuit, illustrated in Figure S13B:

$$d(sp\_0)/dt = Reaction\_1 + Reaction\_3 + Reaction\_4 + Reaction\_10 + Reaction\_12 \tag{1}$$

Each "Reaction" component comprises a rate and species respective to this reaction:

$$Reaction\_1 = leak * sp\_3 * sp\_2 \tag{2}$$

where sp_3 and sp_2 are the reactants and $leak$ is the reaction rate. If a reaction is bidirectional, it is presented as the difference between forward and reverse reactions:

$$Reaction\_9 = kt * sp\_9 * sp\_7 - leak * sp\_5 * sp\_2 \tag{3}$$

It is worth noting that reaction numbers for this example may not always match with those specified in Figure S13B since the algorithm numbers the reactions consecutively. For example, $Reaction\_5$ corresponds to reaction #10 of Figure S13B.

After the ODE model is obtained, the entire model can now be simulated. Following this, each reaction is deactivated one by one, and the results are saved to a 3-D matrix. By deactivating a reaction, its contribution in every ODE will be set to 0. It is possible to avoid deactivating the intended reactions since we are only interested in the effect of leaks. The results can be presented graphically and numerically, sorted by the contribution of each reaction. If the intended reactions were not disabled, hence no difference with the full model, they would appear at the end of the list.

S3 Leak analysis of control circuits

S3.1 Representing signals in DSD circuits

We represent a reversible bimolecular chemical reaction as $R_1 + R_2 \underset{\delta_2}{\overset{\delta_1}{\rightleftharpoons}} P_1 + P_2$. Here, R_1 and R_2 are the reactants, P_1 and P_2 are the products while δ_1 and δ_2 denote the forward reaction rate and backward reaction rate, respectively. As suggested in[3] and,[4] we define a signal x as the difference in concentration of two chemical species, x^+ and x^-, which represent respectively the positive and negative components of x such that $x = x^+ - x^-$. In practice, x^+ and x^- can be realised as single strand DNA molecules.[4,5] We use the superscript '$\pm$' and '$\mp$' as the shorthand notations to represent the reactions or equations with '+' and '−' components – for example, $x_i^{\pm} \xrightarrow{K} x_i^{\pm} + x_o^{\pm}$ should be understood as the set of two reactions: $x_i^{+} \xrightarrow{K} x_i^{+} + x_o^{+}$ and $x_i^{-} \xrightarrow{K} x_i^{-} + x_o^{-}$. The catalysis reaction is of the form $x_i^{\pm} \xrightarrow{\gamma K} x_i^{\pm} + x_o^{\pm}$. The degradation reaction is of the form $x_o^{\pm} \xrightarrow{\gamma} \emptyset$ where $\emptyset$ denotes a waste product or an inert product. The annihilation reaction is of the form $x_o^{+} + x_o^{-} \xrightarrow{\eta} \emptyset$. Gain, summation junction, and integrator can be implemented in CRN using the results of[3] and[4] as follows. Let $x_o = Kx_i$ where x_i is the input, x_o is the output and K is the gain. This operation is implemented using the following CRN: $x_i^{\pm} \xrightarrow{\gamma K} x_i^{\pm} + x_o^{\pm}$, $x_o^{\pm} \xrightarrow{\gamma} \emptyset$, $x_o^{+} + x_o^{-} \xrightarrow{\eta} \emptyset$, where γK, γ and η are the kinetic rates associated with catalysis, degradation and annihilation respectively. The summation operation $x_o = x_i + x_d$, where x_i and x_d are the inputs and x_o is the output, is implemented using the following CRN: $x_i^{\pm} \xrightarrow{\gamma} x_i^{\pm} + x_o^{\pm}$, $x_d^{\pm} \xrightarrow{\gamma} x_d^{\pm} + x_o^{\pm}$, $x_o^{\pm} \xrightarrow{\gamma} \emptyset$, $x_o^{+} + x_o^{-} \xrightarrow{\eta} \emptyset$. Consider the integrator $x_o = K \int x_i \, dt$ where x_i is the input, x_o is the output, and K is the DC gain. This integrator is implemented using the following CRN: $x_i^{\pm} \xrightarrow{K} x_i^{\pm} + x_o^{\pm}$, $x_o^{+} + x_o^{-} \xrightarrow{\eta} \emptyset$. Nonlinearities can also be synthesized using DSD and CRN:a ratio computation circuit using DSD and CRN was derived in.[6]

S3.2 DSD Implementation of CRNs

Three idealized chemical reactions – catalysis, degradation and annihilation – can serve as elementary building components of larger circuits.[4] We now illustrate how to implement annihilation and degradation via DSD using two different translation architectures: 2-domain and 4-domain. The catalysis reactions are described in the main text in detail.

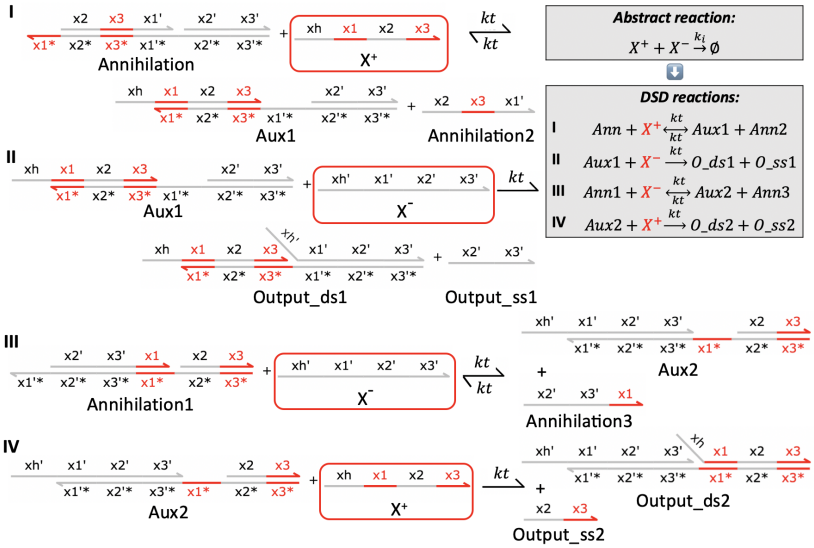


Figure S9: **Implementation of a 4-domain annihilation DNA strand displacement circuit.**

Figures S9 and S10 illustrate the Annihilation reaction $X^+ + X^- \rightarrow \emptyset$ under 4-domain and 2-domain strategies, respectively. In the first DSD reaction, the signal strand X^+ is used to produce an auxiliary product and a waste. This reaction must be reversible so that in the absence of X^-, X^+ does not get consumed and is free to participate in other reactions. In the second DSD reaction, X^- reacts with an auxiliary species to produce an inert waste. All reactions proceed at the same rate kt. The next two DSD reactions implement the same annihilation reaction but with the order of species reversed, so that X^- binds first, and X^+ reacts with a product of the first reaction. This ensures that the process is symmetrical. Even though the first reaction is reversible, it still results in a fraction of signal species being bound for a short time. The symmetric reaction ensures that equal parts of signal species X^+ and X^- are bound simultaneously.

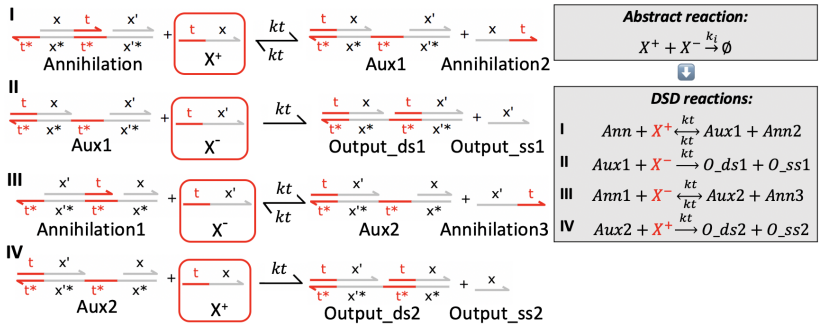


Figure S10: **Implementation of a 2-domain annihilation DNA strand displacement circuit.**

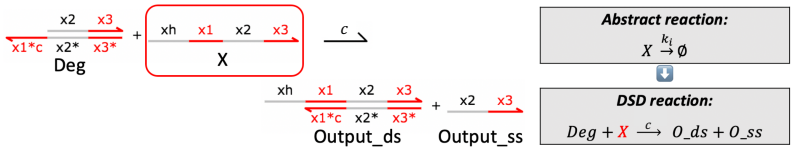


Figure S11: **Implementation of a 4-domain degradation DNA strand displacement circuit.**

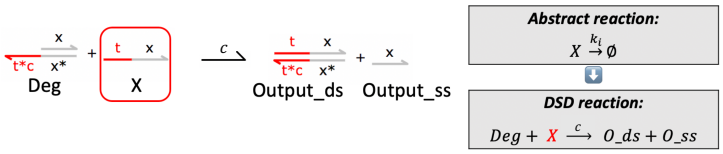


Figure S12: **Implementation of a 2-domain degradation DNA strand displacement circuit.**

For the degradation reaction $X \to \emptyset$, 4-domain and 2-domain schemes are shown in Figures S11 and S12, respectively. This is a one-step reaction which consumes signal X and produces inert waste. The rate c for this reaction is scaled. It should be noted that to implement these DSD reactions in Visual DSD, one must substitute X^+ with X and X^- with X', since the "+" and "−" symbols are reserved in Visual DSD for arithmetic operations.

Table S2 illustrates the mismatch in performance between full leak circuit and the first order leak approach simulated for basic reaction types under two different translation schemes. The average error was calculated from 1000 points every 10 s over the total simulation time of 10000 s. In this simulation, the scaled rate c in catalysis and degradation circuits was set to $8 \cdot 10^{-7} nM^{-1}s^{-1}$ while the toehold binding rate kt is $10^{-3} nM^{-1}s^{-1}$. This was done to enable combining the basic reactions into a larger circuit to ensure its correct operation for a longer time.

Although the annihilation and degradation reactions produce only waste products, for the purpose of our analysis we will treat these products as outputs since they can also be prone to leaks. Thus, in the degradation circuit the outputs comprise a single-stranded DNA (ssDNA) species and a double-stranded DNA (dsDNA) species with no exposed toeholds (Figures S11 and S12). The annihilation circuit has two inputs: X^+ and X^-. Because of a symmetrical duplication of the reactions, there are four outputs: two ssDNA species and two dsDNA species (Figures S9 and S10).

For all simulations in this section, the input X for the catalysis and degradation circuits was set to 1 nM. For the annihilation circuit, the input X^+ was set to 2 nM and the input X^- to 1 nM. The initial concentration of the auxiliary species C_{max} was set to 1000 nM and the maximum complex size $N = N1 + N2$ was set to 8. Such limit was chosen to ensure that all originally present species can interact with one another while guaranteeing the finite simulation.

Time	A	B	C	D	E
1000	0.80	3.58	3.59	349%	0.06%
2000	1.60	8.72	8.72	446%	0.09%
5000	3.99	32.78	32.83	722%	0.14%
10000	7.96	96.24	96.39	1109%	0.16%
15000	11.92	177.01	177.28	1385%	0.15%
20000	15.86	262.80	263.17	1557%	0.14%
25000	19.78	345.47	345.92	1646%	0.13%
30000	23.69	420.92	421.39	1676%	0.11%
40000	31.47	545.88	546.34	1635%	0.09%
50000	39.18	639.56	639.98	1532%	0.07%

(a) 4-domain catalysis

Time	A	B	C	D	E
1000	0.80	3.78	3.78	374%	0.13%
2000	1.60	7.53	7.55	372%	0.26%
5000	3.99	18.57	18.69	366%	0.63%
10000	7.96	36.35	36.79	357%	1.21%
15000	11.92	53.38	54.32	348%	1.74%
20000	15.86	69.72	71.31	340%	2.23%
25000	19.78	85.41	87.77	332%	2.68%
30000	23.69	100.51	103.72	324%	3.10%
40000	31.47	129.05	134.18	310%	3.83%
50000	39.18	155.59	162.83	297%	4.45%

(b) 2-domain catalysis

Time	A	B	C	D	E
1000	0.28	1.28	1.28	354%	0%
2000	0.39	2.39	2.39	516%	0%
5000	0.48	5.46	5.46	1041%	0%
10000	0.50	10.42	10.42	1990%	0%
15000	0.50	15.31	15.31	2961%	-0.01%
20000	0.50	20.15	20.14	3927%	-0.01%
25000	0.50	24.94	24.93	4885%	-0.02%
30000	0.50	29.68	29.67	5833%	-0.03%
40000	0.50	39.03	39.01	7701%	-0.05%
50000	0.50	48.19	48.15	9533%	-0.08%

(c) 4-domain annihilation

Time	A	B	C	D	E
1000	0.28	1.28	1.28	354%	0%
2000	0.39	2.39	2.39	516%	0%
5000	0.48	5.46	5.46	1041%	0%
10000	0.50	10.42	10.42	1990%	0%
15000	0.50	15.31	15.31	2962%	-0.01%
20000	0.50	20.15	20.15	3929%	-0.01%
25000	0.50	24.95	24.94	4887%	-0.02%
30000	0.50	29.69	29.68	5836%	-0.03%
40000	0.50	39.04	39.02	7705%	-0.05%
50000	0.50	48.21	48.17	9536%	-0.08%

(d) 2-domain annihilation

Table S2: **Comparison of performance of catalysis (a, b) and annihilation (c, d) reactions under different translation schemes using first order leak approximation.** For catalysis reaction, the reaction rate for the first step is set to $8 \cdot 10^{-7} nM^{-1}s^{-1}$, for the subsequent steps - $10^{-3} nM^{-1}s^{-1}$; for annihilation reaction the rate is set to $10^{-3} nM^{-1}s^{-1}$. Such rates are chosen as they will be used for simulation of the Gain circuit. Column **A** signifies the Output value in no-leak mode; column **B** marks the Output value obtained using first order leaks approach; column **C** – the Output value obtained using full leak mode; column **D** – computational error between no-leak and first order leak simulations, $D = (B - A)/A$; and column **E** – percentage mismatch between full leak and first order leak approaches, $E = (C - B)/C$. For annihilation reaction, the error specified is for a single-stranded product only (presented for Output_ss2); the error for double-stranded products is insignificant.

Hereby, we present the analysis of 4-domain catalysis circuit simulated with and without leaks. Figure S13 illustrate 4 simulations: full leaks (A), first order leaks (B), leaks generated by classic Visual DSD (C) and no-leak simulation (D). The corresponding reactions are marked with the same numbers. Six reactions from the full leak CRN are not included in the first order leak CRN: #3, 11, 13, 15, 16, 18. In these reactions, one of the reactants is a species that was not originally present in the system and was formed in a course of a leak reaction – *sp_4*, *sp_7* and *sp_8*. Therefore, such reactions are considered second-order leaks and are excluded from the first order leak CRN. Reaction #17 has the same reactants as reaction #15, which was excluded, however, it can occur via TMSD, therefore it is included in the first order leak CRN.

A. Full leak CRN generated by Logic DSD leak semantics

```
1  | sp_3 + sp_2 ->{leak} sp_1 + sp_0 + sp_7
2  | sp_9 + sp_3 ->{(leak + kt)} sp_5 + sp_1 + sp_0
3  | sp_8 + sp_2 ->{leak} sp_6 + sp_0 + sp_7
4  | sp_3 + sp_2 <->{leak}{leak} sp_9 + sp_8
5  | sp_9 + sp_8 ->{(leak + kt)} sp_5 + sp_6 + sp_0
6  | sp_1 + sp_7 ->{((leak + kt) + c)} sp_5 + sp_6
7  | sp_1 + sp_8 <->{(leak + c)}{leak} sp_6 + sp_3
8  | sp_1 + sp_2 <->{(leak + c)}{leak} sp_9 + sp_6
9  | sp_9 + sp_7 <->{(leak + kt)}{leak} sp_5 + sp_2
10| sp_7 + sp_3 <->{(leak + kt)}{leak} sp_5 + sp_8
11| sp_8 + sp_2 <->{leak}{leak} sp_4 + sp_2
12| sp_4 + sp_2 ->{(leak + kt)} sp_6 + sp_0 + sp_7
13| sp_1 + sp_8 <->{leak}{leak} sp_4 + sp_1
14| sp_6 + sp_3 <->{leak}{(kt + leak)} sp_4 + sp_1
15| sp_4 + sp_9 ->{leak} sp_3 + sp_2
16| sp_9 + sp_8 <->{leak}{leak} sp_4 + sp_9
17| sp_4 + sp_9 ->{(leak + kt)} sp_5 + sp_6 + sp_0
18| sp_4 + sp_7 ->{leak} sp_7 + sp_8
```

B. First-order leak CRN by Logic DSD leak semantics

```
1  | sp_3 + sp_2 ->{leak} sp_1 + sp_0 + sp_7
4  | sp_3 + sp_2 ->{leak} sp_9 + sp_8
2  | sp_9 + sp_3 ->{(leak + kt)} sp_5 + sp_1 + sp_0
5  | sp_9 + sp_8 ->{kt} sp_5 + sp_6 + sp_0
10| sp_7 + sp_3 ->{kt} sp_5 + sp_8
6  | sp_1 + sp_7 ->{(kt + c)} sp_5 + sp_6
7  | sp_1 + sp_8 <->{c}{leak} sp_6 + sp_3
8  | sp_1 + sp_2 <->{(leak + c)}{leak} sp_9 + sp_6
9  | sp_9 + sp_7 <->{kt}{leak} sp_5 + sp_2
12| sp_4 + sp_2 ->{kt} sp_6 + sp_0 + sp_7
14| sp_6 + sp_3 <->{leak}{kt} sp_4 + sp_1
17| sp_4 + sp_9 ->{kt} sp_5 + sp_6 + sp_0
```

C. Leak CRN generated by Classic Visual DSD

```
8  | sp_1 + sp_2 <->{kt*c}{leak} sp_6 + sp_5
2  | sp_6 + sp_3 ->{kt} sp_4 + sp_1 + sp_0
```

D. No-leak CRN

```
8  | sp_1 + sp_2 ->{c} sp_6 + sp_5
2  | sp_6 + sp_3 ->{kt} sp_4 + sp_1 + sp_0
```

Figure S13: **CRNs for the 4-domain Catalysis circuit: the input is** *sp_1* **and the output is** *sp_0*. (**A**) illustrates the full leak CRN produced by Logic DSD semantics; (**B**) contains first order leak reactions; (**C**) presents the leak CRN produced by classic Visual DSD; and (**D**) shows the minimal CRN of a no-leak circuit. Reactions are numbered to illustrate the corresponding reactions in different CRNs. Intended reactions are highlighted in bold red. Rate c in panels (**A**), (**B**) and (**D**) is compatible with the rate $kt * c$ from panel (**C**).

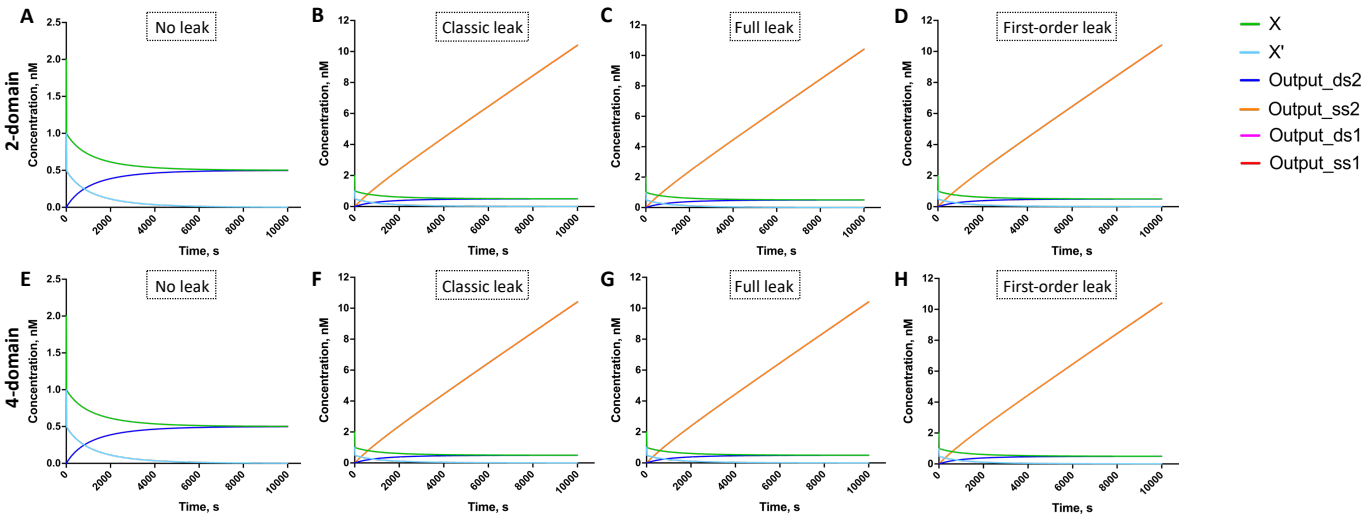


Figure S14: **Simulations of the annihilation circuit under the two implementation schemes with different leak hypotheses.** The circuit is implemented without leaks **(A, E)**, using classic Visual DSD leak functionality **(B, F)**, with full leaks generated by leak semantics **(C, G)** and using first order leak approximation **(D, H)**. The initial conditions are as follows: initial concentration of input X^+ is set to 2 nM, X^- is set to 1 nM, fuel species are set to $C_{max} = 1000$ nM, toehold binding rate $kt = 10^{-3} nM^{-1} s^{-1}$, maximum size limit N is set to 8.

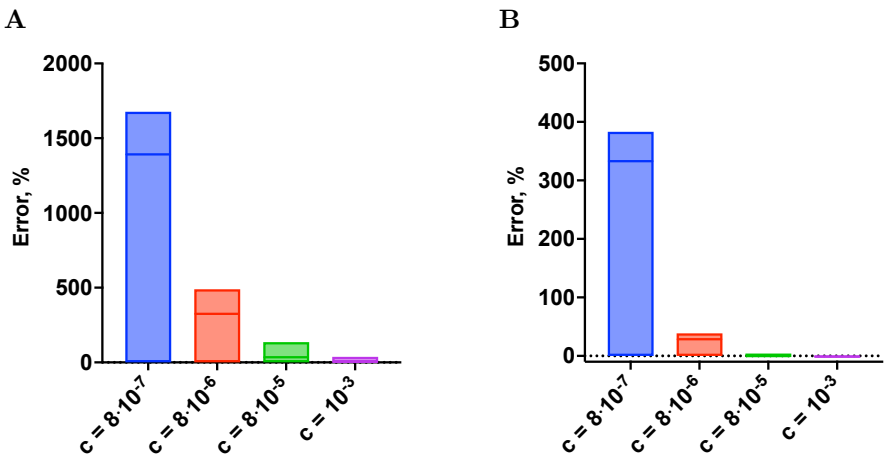


Figure S15: **Performance comparison of the 4-domain (A) and 2-domain (B) catalysis circuits between first order leak and no-leak implementations for different scaled rates *c*.** The bars represent minimum to maximum errors with a line at mean. The error is calculated as a difference of the output value in the first order leak and no-leak modes. The toehold binding rate is set to $kt = 10^{-3}M^{-1}s^{-1}$ and the maximum complex size $N1 + N2$ is set to 8, with simulations run for 50000 s. For most simulations, the computational error of 2-domain scheme is significantly lower than for the 4-domain scheme.

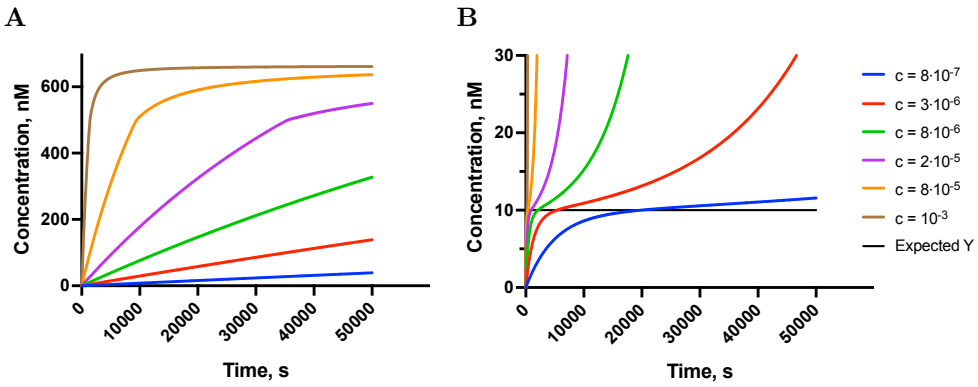


Figure S16: **Illustration of change of performance for different circuits as the scaled rate *c* is increased:** (**A**) 2-domain catalysis, (**B**) 2-domain gain. As the scaled rate c approaches the rate $kt = 10^{-3}M^{-1}s^{-1}$, the function of the circuit gets more and more disrupted.

S3.3 Application of the first order leak approximation to a gain circuit

Species	Average % error	Average difference, nM
Y^-	0.85%	0.03
Y^+	-0.51%	-0.01
X^-	0.3%	0.10
X^+	0.36%	0.12
$Degradation^-$	-0.13%	-0.56
$Degradation^+$	0.02%	-0.11
$Catalysis^-$	-0.77%	-2.87
$Catalysis1^-$	-0.78%	-2.61
$Catalysis^+$	-0.62%	-2.37
$Catalysis1^+$	-0.60%	-2.10
$Annihilation$	0.18%	-1.24
$Annihilation1$	0.11%	1.31
$Annihilation2$	-0.11%	-0.79
$Annihilation3$	0.08%	0.93

(a) 4-domain Gain circuit

Species	Average % error	Average difference, nM
Y^-	31.13%	0.17
Y^+	-25.30%	-1.65
X^-	0.0%	0.0
X^+	0.0%	0.0
$Degradation^-$	-0.33%	-1.51
$Degradation^+$	3.12%	12.07
$Catalysis^-$	-2.55%	-20.51
$Catalysis1^-$	-9.47%	-82.31
$Catalysis2^-$	-2.88%	-19.52
$Catalysis3^-$	-2.71%	-23.47
$Catalysis4$	-3.65%	-51.99
$Catalysis5$	-7.66%	-72.84
$Catalysis^+$	-0.14%	-0.93
$Catalysis1^+$	0.14%	1.03
$Catalysis2^+$	-0.45%	-2.59
$Catalysis3^+$	-0.57%	-4.27
$Annihilation$	-1.15%	-9.18
$Annihilation1$	-1.36%	-10.98
$Annihilation2$	0.80%	7.67
$Annihilation3$	1.32%	12.24

(b) 2-domain Gain circuit

Table S3: **Comparison of full leaks and first order leak approaches for Gain circuit under different translation schemes in terms of computational error.** Subtable (**a**) illustrates 4-domain implementation. Here, the mismatch between full leak and first order leak approaches is minimal for all species. Subtable (**b**) illustrates 2-domain implementation. Here, some of the species show rather large average % error, e.g. species Y^-. However, such error is caused by comparing two very small values, as can be observed from the Average difference column. For species Y^+, the Average difference is more significant, which, given the high average error, suggests that the first order leak approach must be used with caution for larger circuits under 2-domain scheme. Average error was calculated from 1000 points every 50 s over the total time of 50000 s. The simulations were made under the following conditions: C_{max} = 1000 nM, c = $8 \cdot 10^{-7} nM^{-1}s^{-1}$, kt = $10^{-3} nM^{-1}s^{-1}$, $N1 + N2 = 8$.

S4 Analysis of leak reduction strategies

S4.1 Leakless circuit encoding

We encoded previous leak reduction strategies[7] in Visual DSD using the following minor syntactic modifications. Firstly, previously[7] four different domain types were used: full domains, almost full domains (with the symbol Δ), toehold domains (with the symbol δ), and clamp domains. In our design, toehold domains are named using the prefix t and labelled with the symbol ^ (e.g. `<tx1^>`); almost full domains are represented as regular long domains (e.g. `<x1>`), and full domains are represented as the concatenation of the corresponding toehold and long domain (e.g. `<tx1^ x1>`). This modification is required since the Δ and δ symbols cannot be used in Visual DSD. Secondly, previously[7] the strands had 5'- 3' orientation from right to left. In Visual DSD, this representation is reversed. For example, in the SLD scheme the signal X was previously[7] represented as $<\delta$x1 x2$>$ with the 3' end on the left, where δx1 is a toehold and the full domain x2 consists of a toehold and a long domain. Here we represent the signal X as `<tx2^ x2 tx1^>` with the 3' end on the right.

Note that both schemes also made use of 2-nucleotide *clamps* to further reduce leaks. However, since these clamps were short enough to still allow fraying, albeit with lower probability, we did not represent them explicitly to enumerate the leaks. An explicit representation of clamps is also possible with our approach, but would lead to a much less concise representation of the same leak pathways.

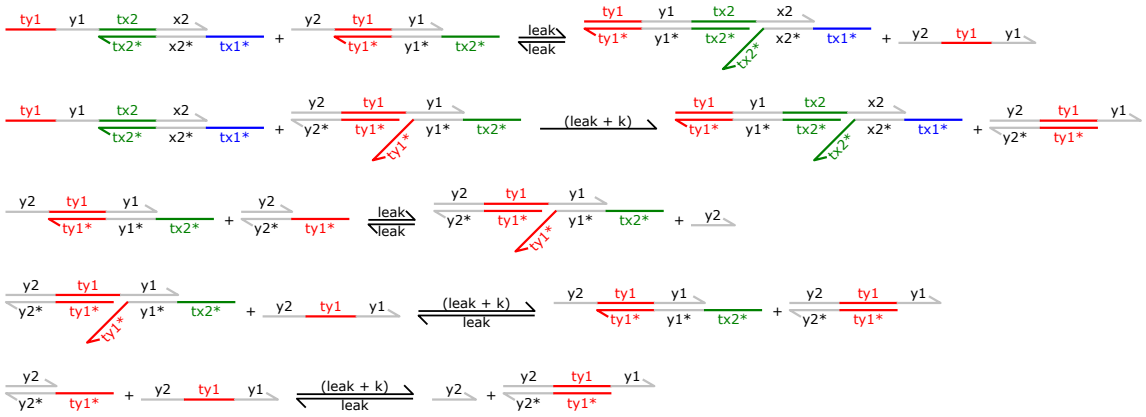
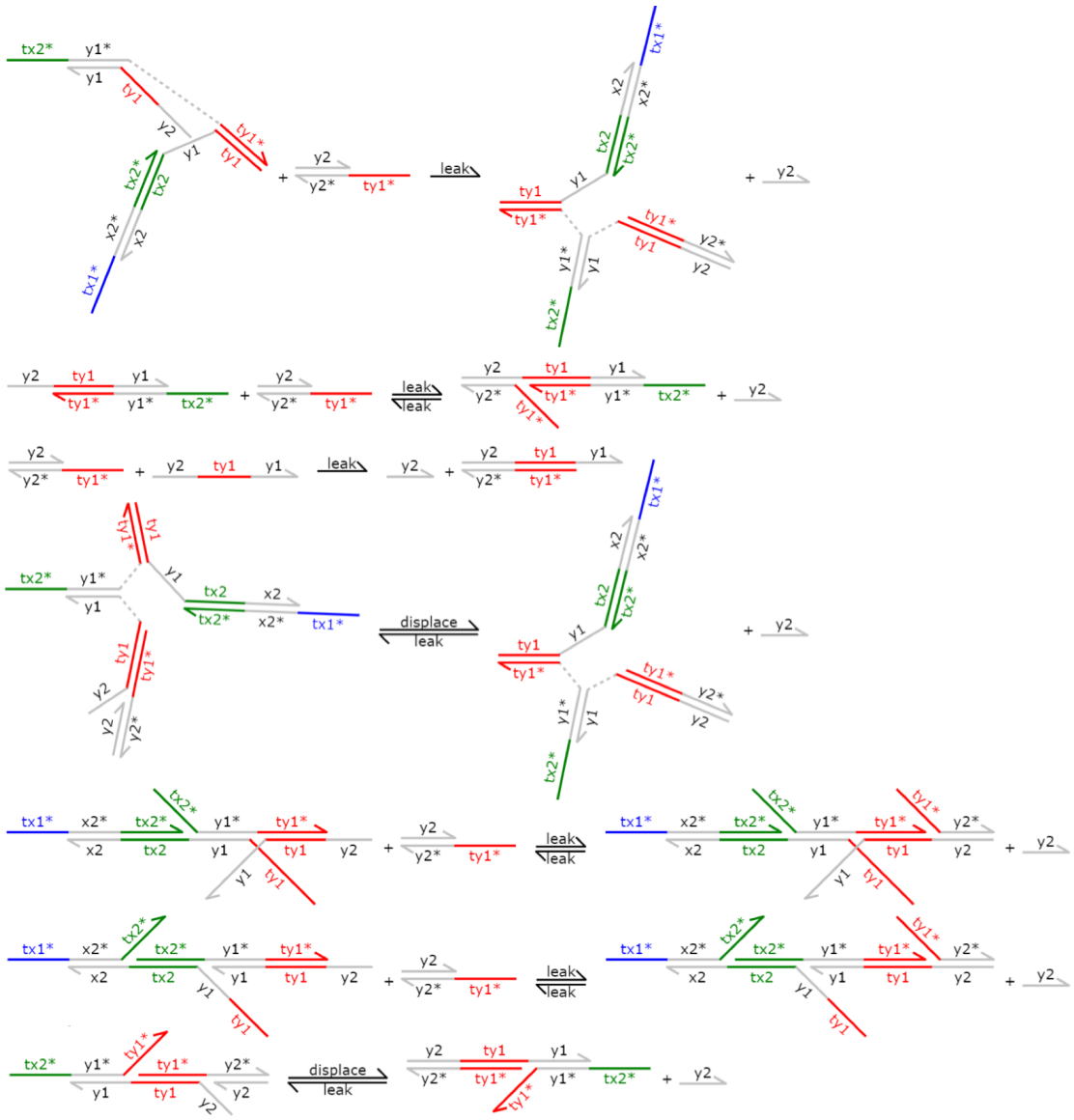
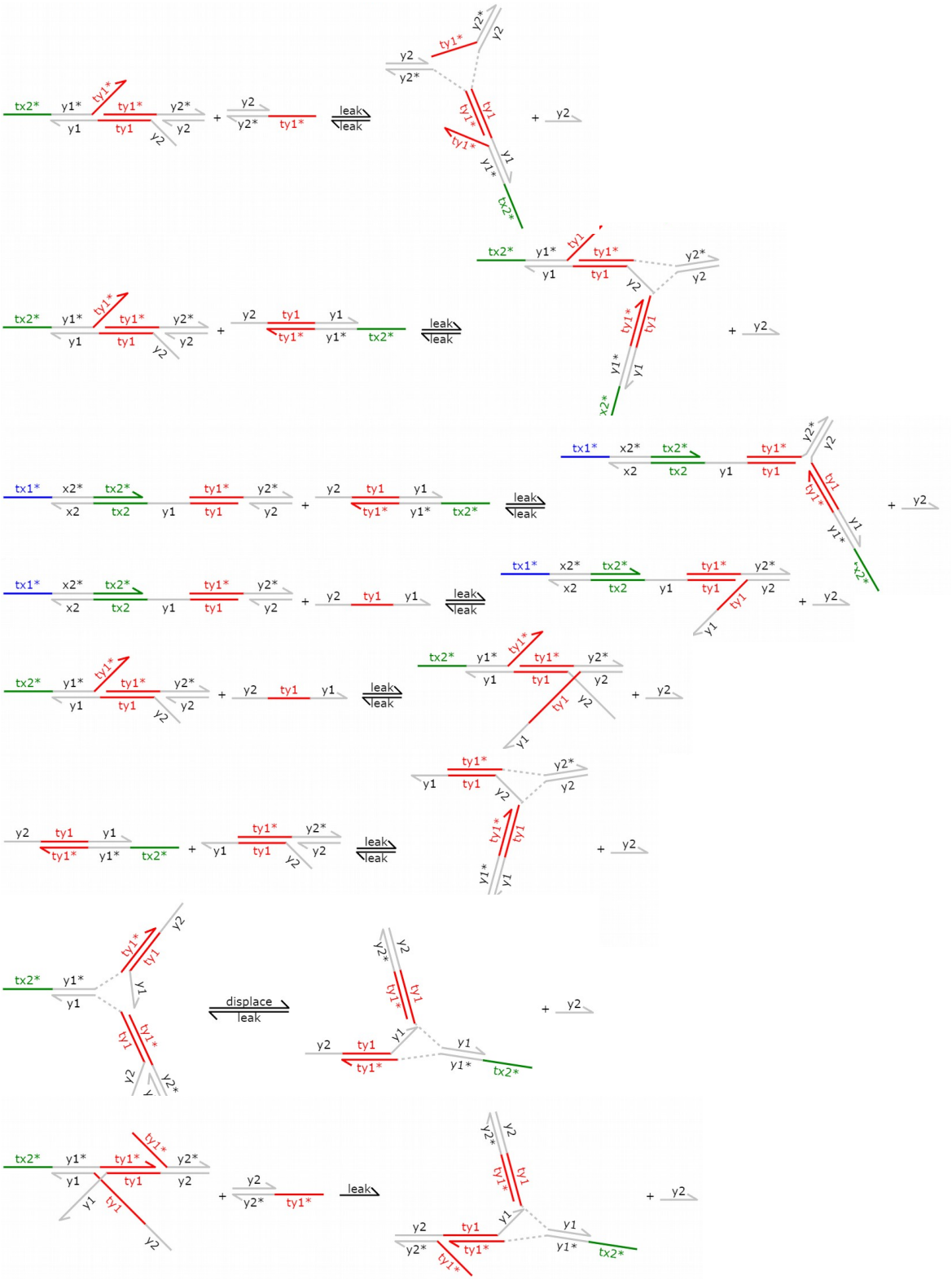


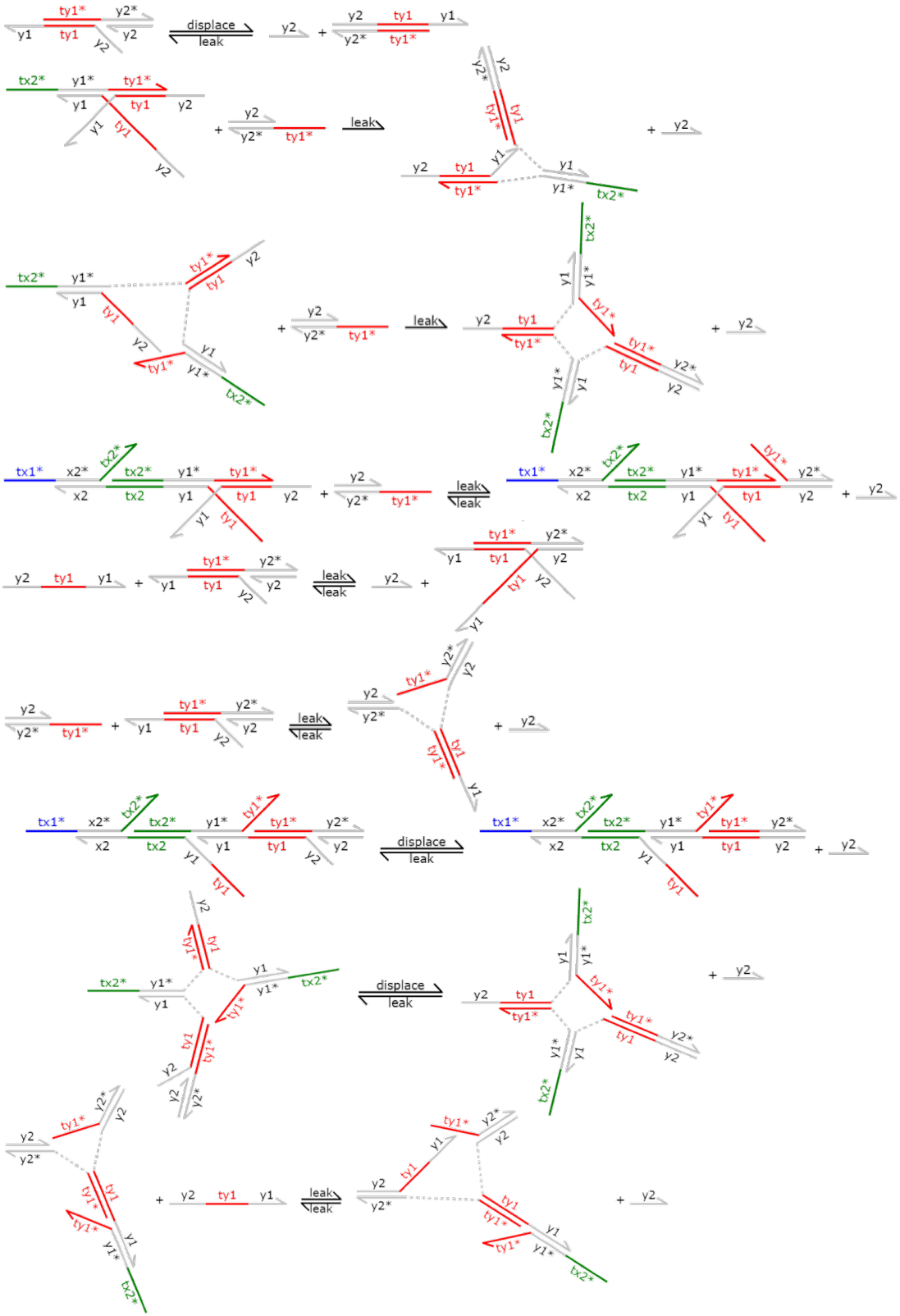
Figure S17: Leak reactions of the SLD circuit detected by the leak semantics in the infinite mode.



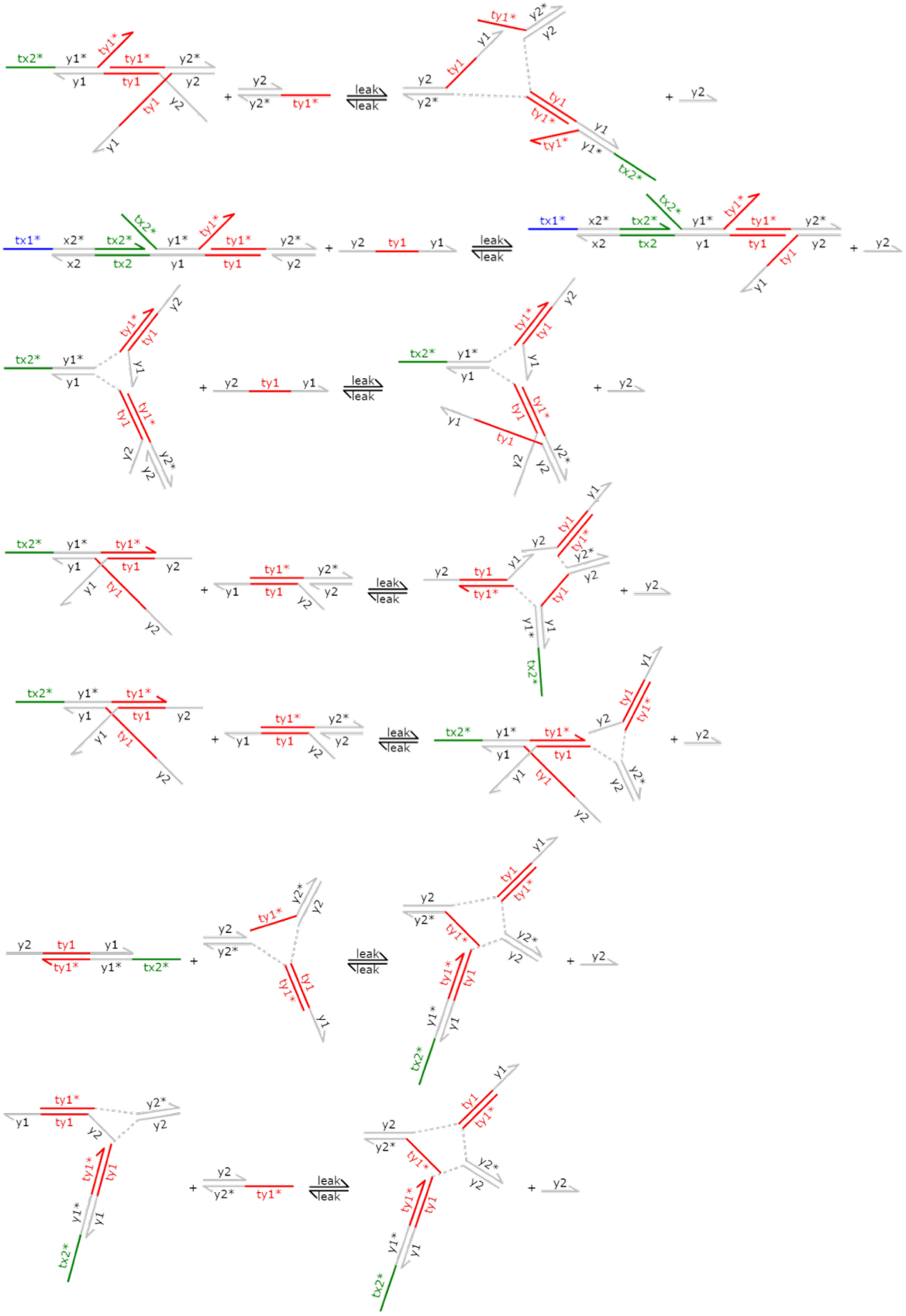
(a) Leak reactions of the SLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode (continued on next page).



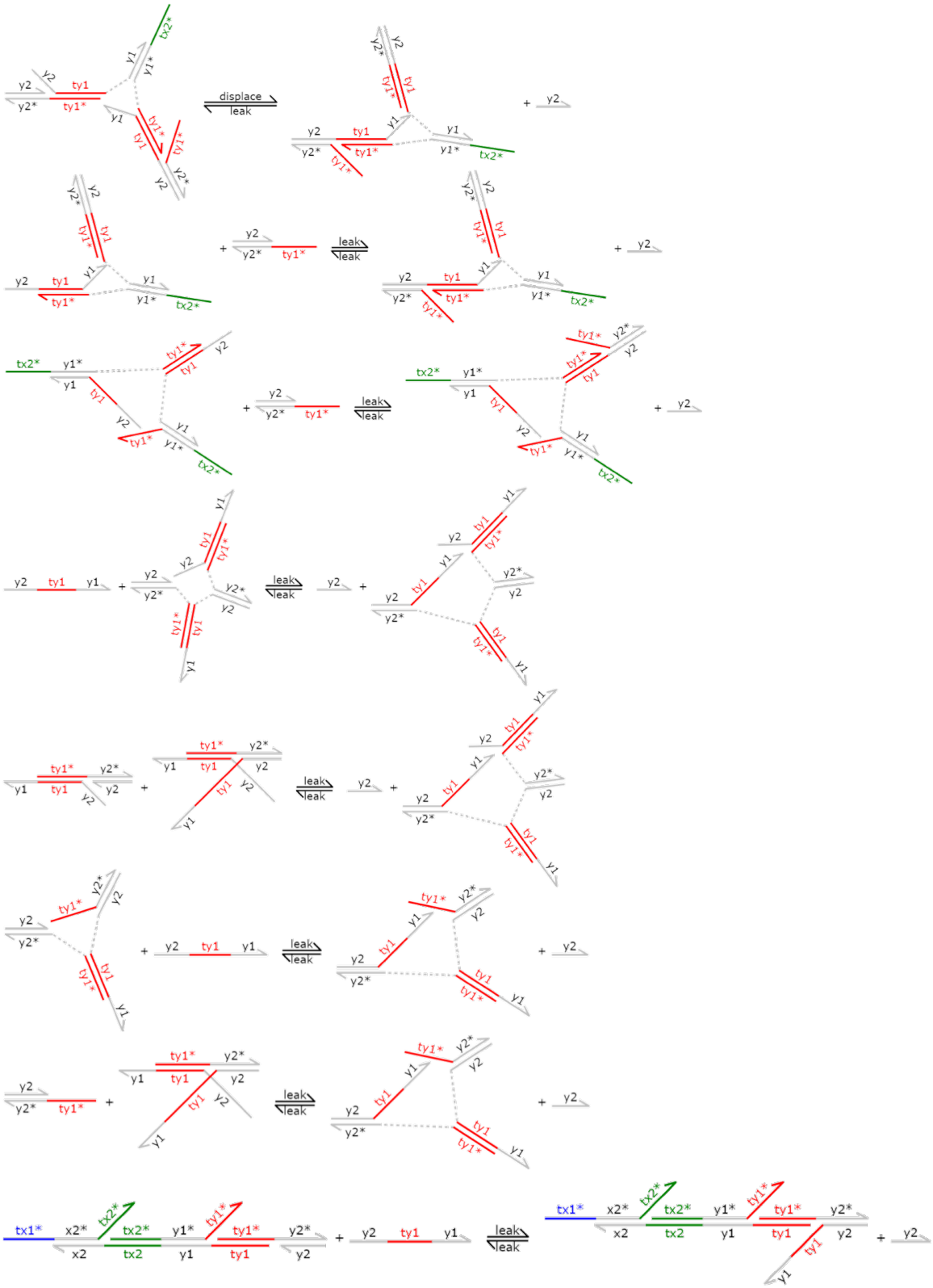
(b) Leak reactions of the SLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode (continued on next page)



(c) Leak reactions of the SLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode (continued on next page)

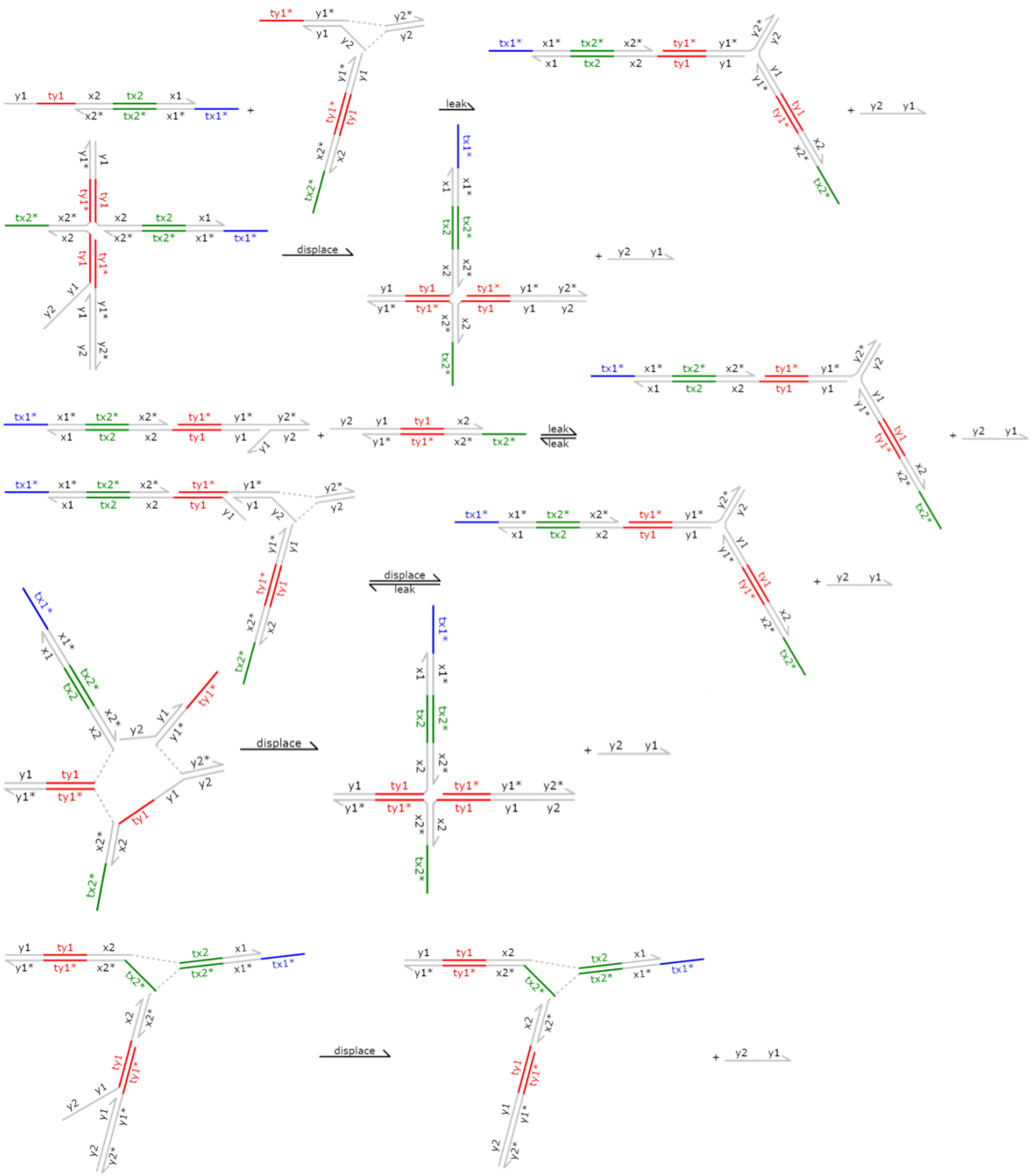


(d) Leak reactions of the SLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode (continued on next page)

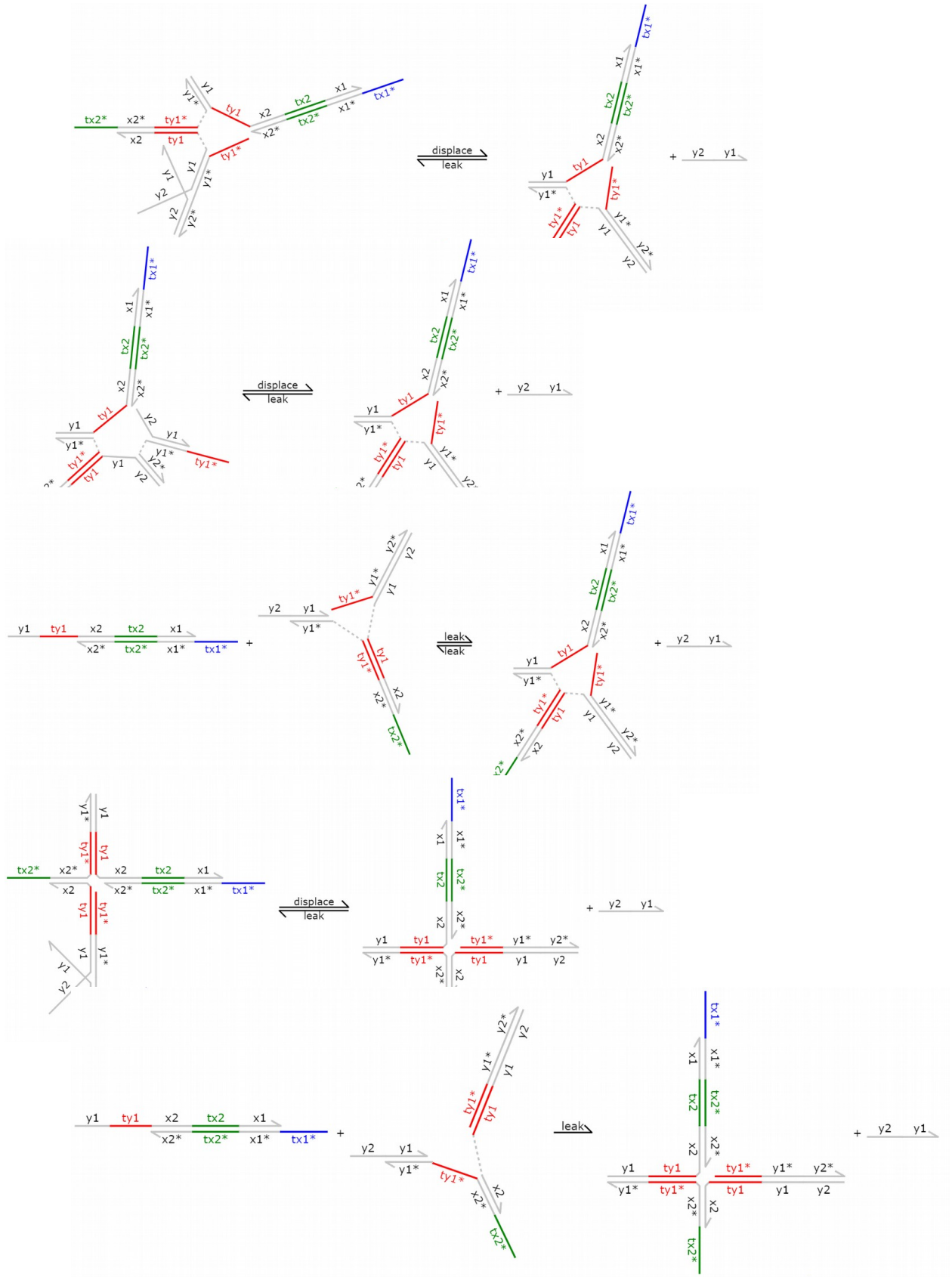


(e)

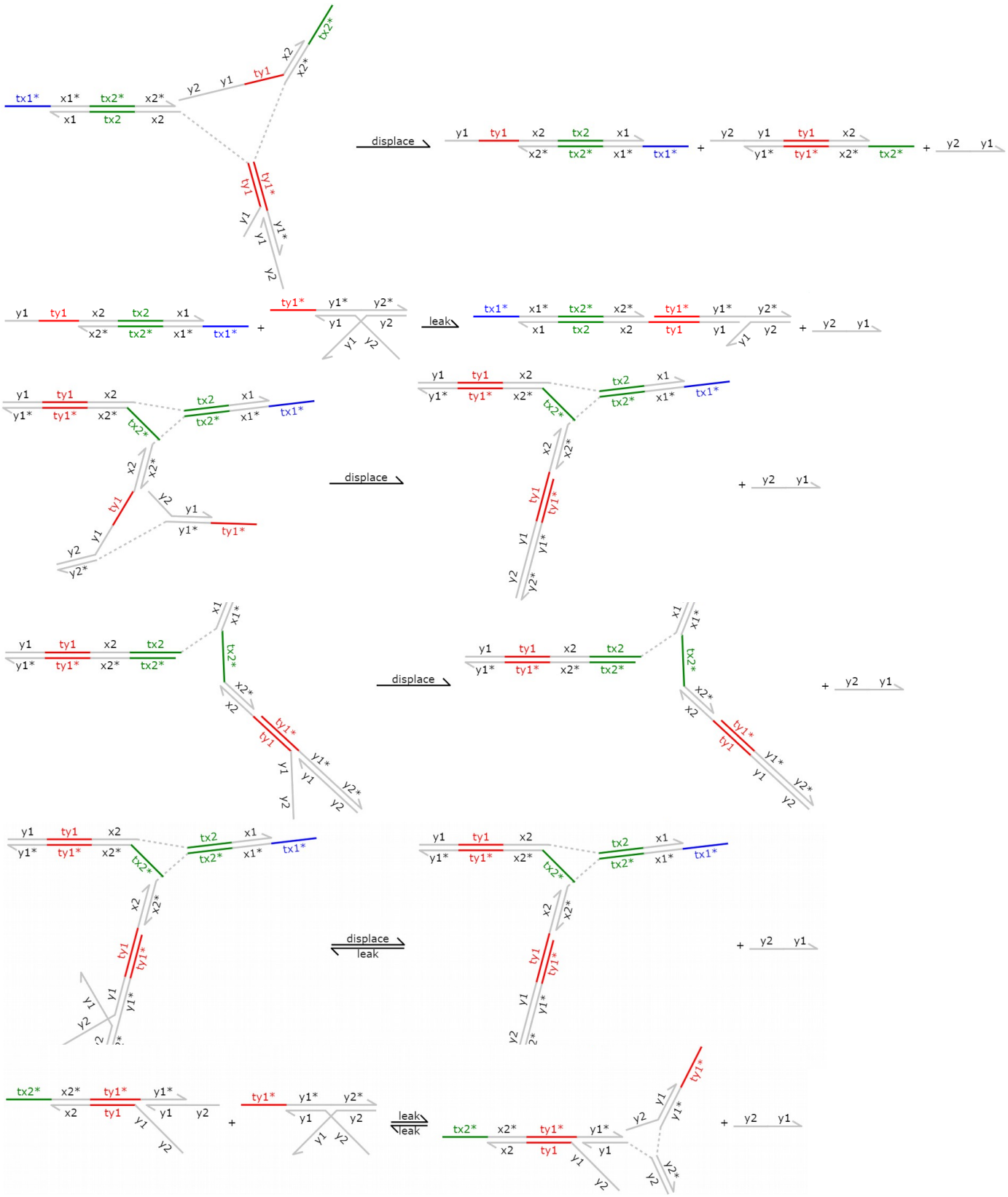
Figure S18: Leak reactions of the SLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode.



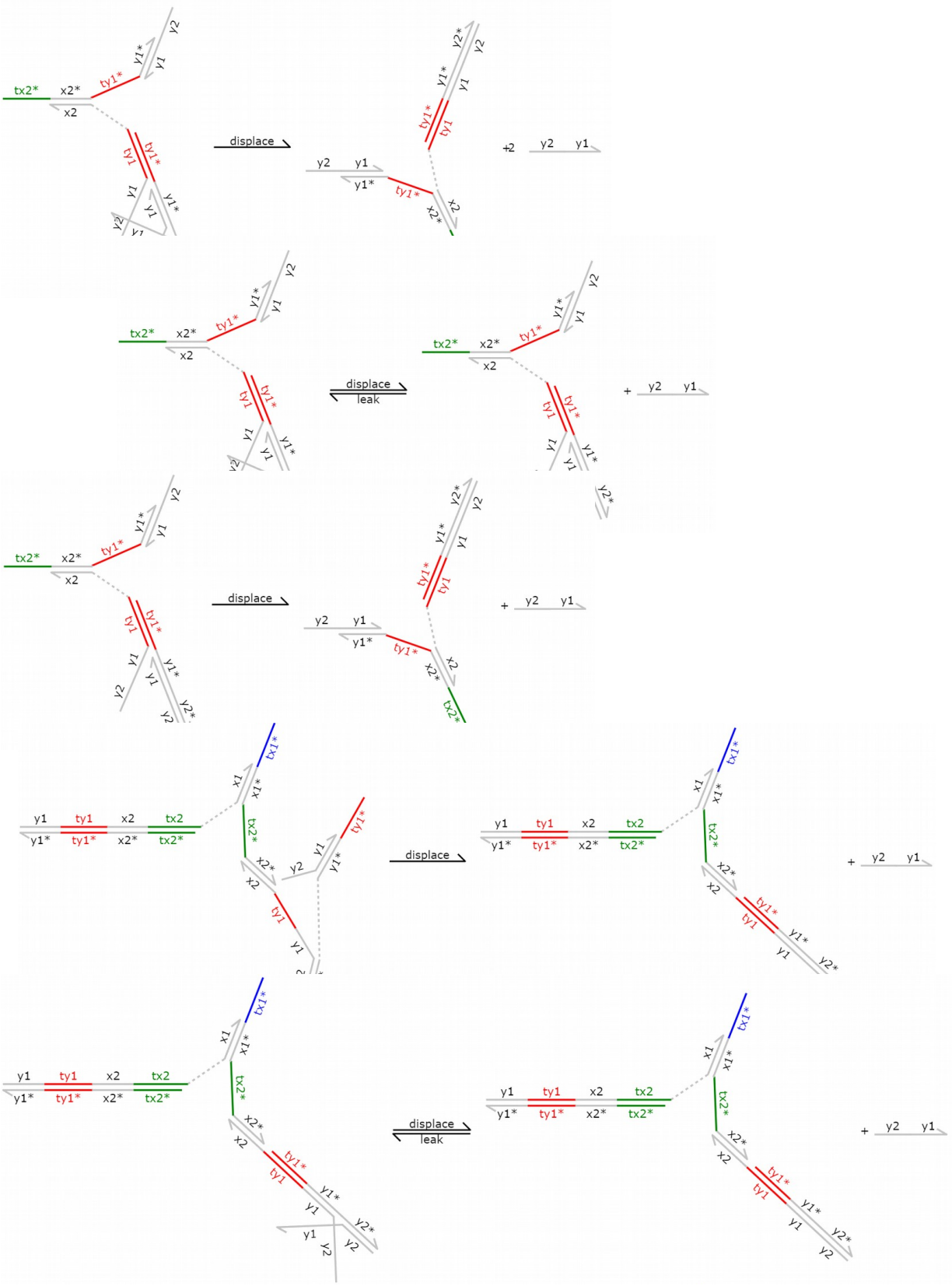
(a) Leak reactions of the DLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode (continued on next page).



(b) Leak reactions of the DLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode (continued on next page)



(c) Leak reactions of the DLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode (continued on next page)



(d)

Figure S19: Leak reactions of the DLD circuit producing *Reporter.top* detected by the leak semantics in the detailed mode.

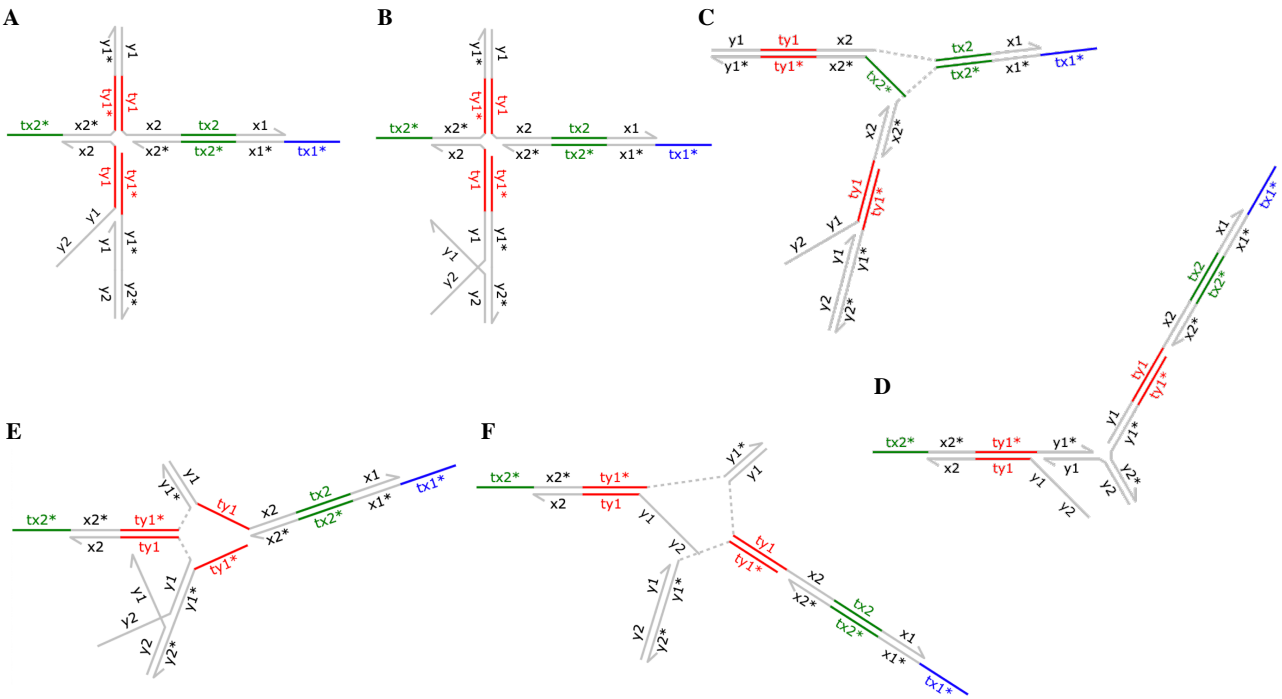


Figure S20: (**A-E**) Different $Y^*_{complex}$ conformations of the DLD circuit generated by the semantics.

S5 Leak reduction of control circuits

S5.1 Elementary circuits

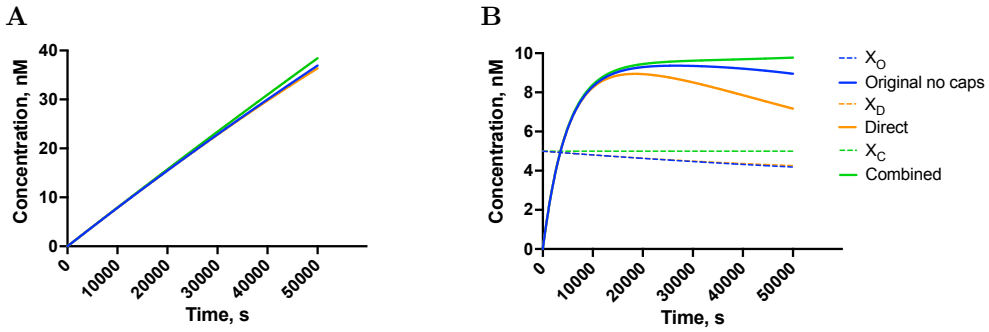


Figure S21: **Comparison of performance of different new architectures for 2-domain catalysis (A) and 2-domain gain circuit (B)**, i.e., original no caps (with and without protectors), direct (with and without protectors) and combined (with protectors, without protectors, and with protectors and caps), simulated without leaks.

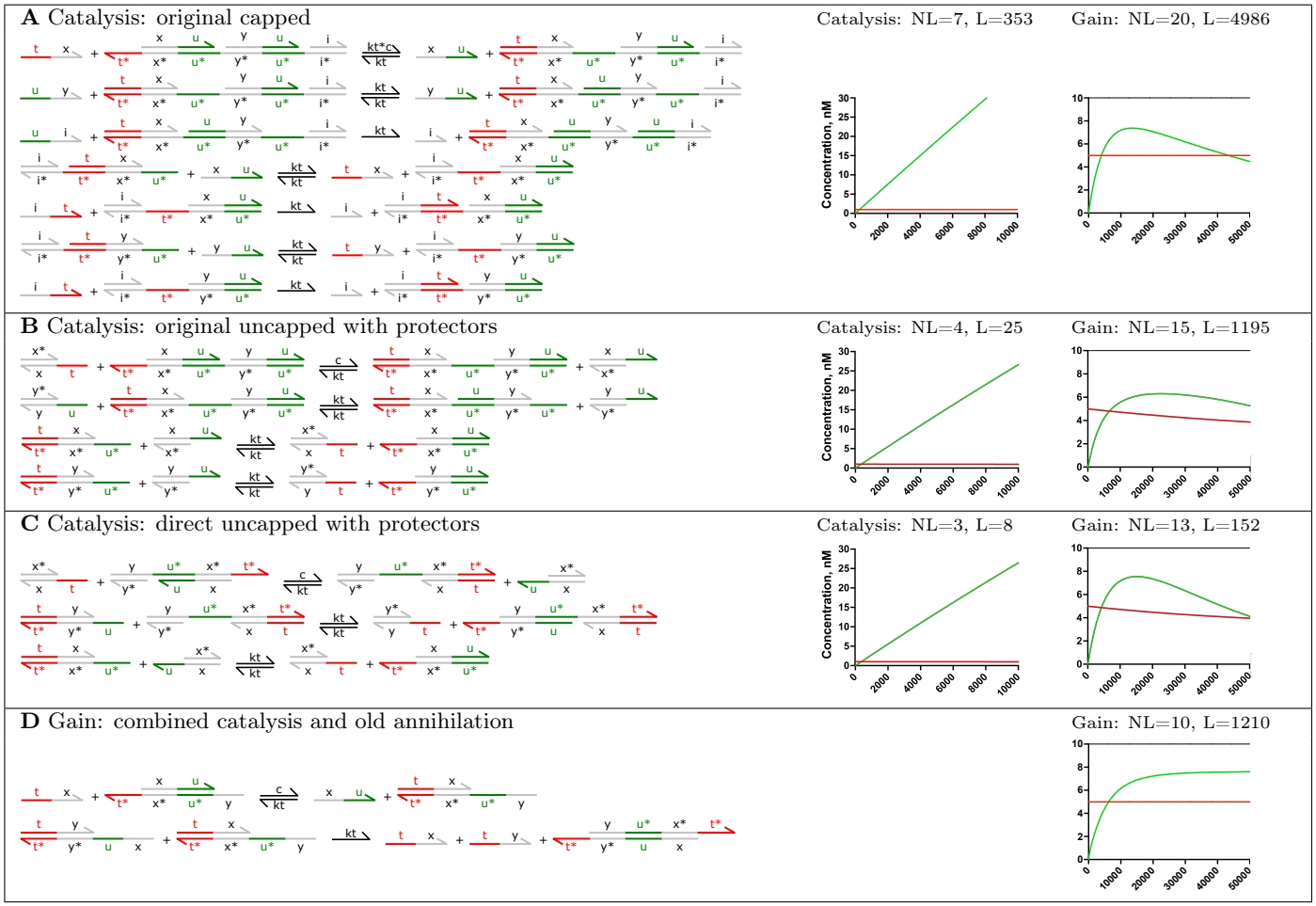


Figure S22: **Application of the automated leak analysis methodology to reduce leakage of the 2-domain catalysis circuit - additional circuit designs.** NL and L denote the number of reactions in the absence and presence of leaks, respectively. Plots illustrate full leak simulations. In the plots, red lines are input X, green lines are output Y, black lines are ideal output of the gain circuit.(**A**) Original catalysis circuit.[4] (**B**) Original catalysis circuit without caps and with protector strands. (**C**) direct displacement of signal and catalyst with protector strands. (**D**) Using combined catalysis with the previous annihilation design.[4]

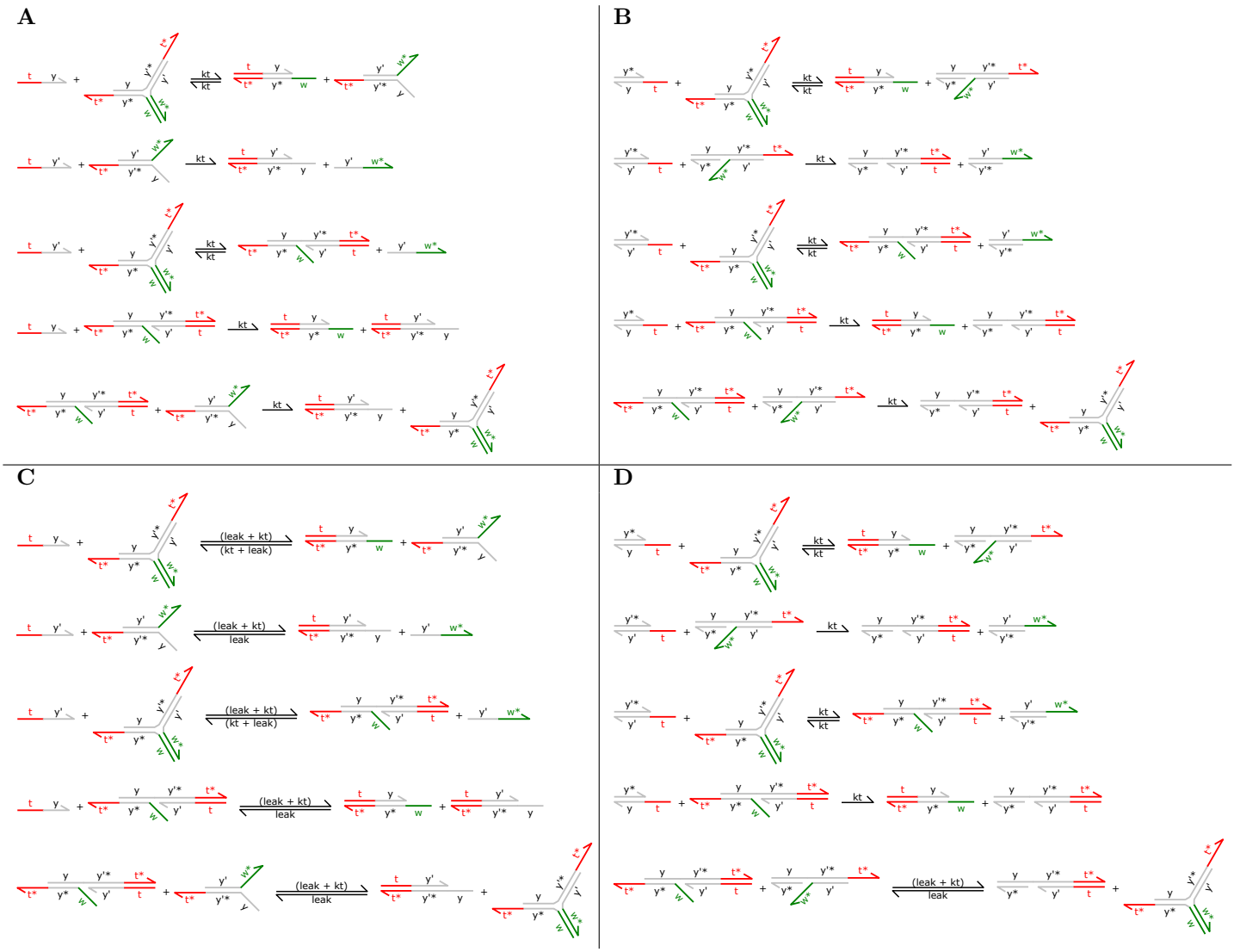


Figure S23: **Application of automated leak analysis methodology to reduce leakage of the 2-domain annihilation circuit:** (**A**) for single-stranded input and (**B**) for input with protectors. (**C** and **D**) illustrate the reactions in the presence of leaks for unprotected and protected signals, respectively.

S5.2 Leak reduction: PI controller

We investigated how the maximum complex size affects the PI circuit performance. The PI controller was first simulated with the maximum complex size set to N = 8, resulting in 1187 reactions for *combined uncapped* design and 1730 reactions for *combined capped* design. When the size limit was removed, the *combined uncapped* circuit generated 5633 reactions which did not affect its performance, as seen in Figure S24 but increased the simulation time by more than six times. The *combined capped* circuit without the size limit could not be simulated due to a large number of reactions. Therefore, setting an appropriate size limit is important in order to simplify the computation by cutting off the low probability reactions that do not affect the overall circuit performance, and in some cases is crucial for enabling the simulation altogether.

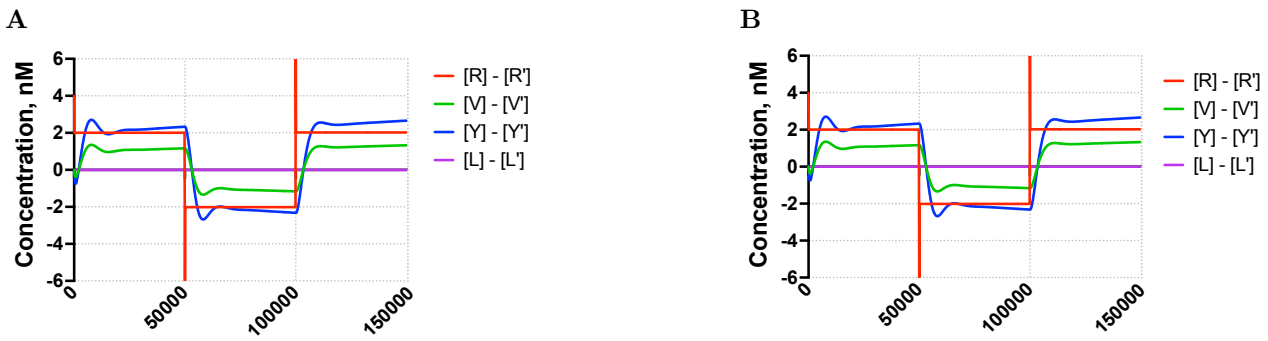


Figure S24: **Leaky simulation of the PI controller using optimised 2-domain combined uncapped catalysis with protectors and branched annihilation with protectors** (**A**) with the maximum complex size N = 8. (**B**) with no limit on maximum complex size.

```
directive rules {
rateDef(D,Type,Rate) :- rate(D,Type,Rate).
rateDef(D,Type,Rate) :- not rate(D,Type,_),
  compl(D, D'), rate(D',Type,Rate).

find(D,Type,Rate):- rateDef(D,Type,Rate).
find(D,Type,Rate):- default(D,Type,Rate),
  not rateDef(D,Type,_).

rate(_{complementarity(X)}, "bind", X).
rate(t^, "bind", "kt").
rate(u^, "bind", "kt").
rate(s^, "bind", "ks").

default([_],"unbind","unbind").
default(_,"bind","kt").
default(_,"displace","displace").
default(_,"cover","cover").
default(_,"migrate","migrate").
default(_,"leak","leak").

mismatched(D) :- D = _{complementarity(_)}.

bindDomain(D,D',D):- mismatched(D).
bindDomain(D,D',D'):- mismatched(D').
bindDomain(D,D',D):- not mismatched(D').

// Interactions on a single domain
bind(P1,P2,Q,E!i) :-
  P1 = C1 [D], P2 = C2 [D'], compl(D, D'),
  bindDomain(D,D',E),
  Q = C1 [D!i] | C2 [D'!i], freshBond(D!i, P1|P2),
  productive(Q, P1|P2, D!i).
productive(Q, P, D!i):- not toehold(D).
productive(Q, P, D!i):- fast(Q,Rate,R), not (P = R).
productive(Q, P, D!i):- slow(Q,Rate,R), not (P = R).

//Junctions
unbind(P,Q,D!i) :-
  P = C [D!i] [D'!i], toehold(D),
  Q = C [D] [D'], not adjacent(D!i,_,P).
adjacent(D!i,E!j,P) :- P = C [D!i E!j] [E'!k D'!i],
  junctionL(E!j, E'!k, P).
adjacent(D!i,E!j,P) :- P = C [E!j D!i] [D'!i E'!k],
  junction(E!j, E'!k, P).
```

```
(*// Used for 4D circuits, for efficiency reasons
unbind(P,Q,D!i) :-
  P = C0 [D1 D!i D2], unbound(D1), unbound(D2),
  toehold(D),
  P = C [D!i] [D'!i], Q = C [D] [D'].
unbind(P,Q,D!i) :-
  P = C0 [<D!i D2], unbound(D2), toehold(D),
  P = C [D!i] [D'!i], Q = C [D] [D'].
unbind(P,Q,D!i) :-
  P = C0 [D1 D!i>], unbound(D1), toehold(D),
  P = C [D!i] [D'!i], Q = C [D] [D'].
unbind(P,Q,D!i) :-
  P = C0 [E!j D!i>] [<E'!j], toehold(D),
  P = C [D!i] [D'!i], Q = C [D] [D'].
unbind(P,Q,D!i) :-
  P = C0 [<D!i E!j][E'!j>], toehold(D),
  P = C [D!i] [D'!i], Q = C [D] [D'].
*)
cover(P,Q,E!j,D!i) :-
  P = C [E!j D] [D' E'!k], compl(D, D'),
  junction(E!j, E'!k, P),
  Q = C [E!j D!i] [D'!i E'!k], freshBond(D!i,P).
coverL(P,Q,E!j,D!i) :-
  P = C [D E!j] [E'!k D'], compl(D, D'),
  junctionL(E!j, E'!k, P),
  Q = C [D!i E!j] [E'!k D'!i], freshBond(D!i,P).

displace(P,Q,E!j,D!i) :-
  P = C [E!j D] [D!i] [D'!i E'!k],
  junction(E!j, E'!k, P),
  Q = C [E!j D!i] [D] [D'!i E'!k].

displaceL(P,Q,E!j,D!i) :-
  P = C [D!i] [D E!j] [E'!k D'!i],
  junctionL(E!j, E'!k, P),
  Q = C [D] [D!i E!j] [E'!k D'!i].

migrate(P, Q, E!j, D!i1) :-
  P = C [E!j D!i1] [D'!i2 E'!k] [D'!i1] [D!i2],
  junctionM(E!j, E'!k, P, [i1;i2]),
  Q = C [E!j D!i1] [D'!i1 E'!k] [D'!i2] [D!i2].

junction(D!i, D'!i, P).
junction(D!i, E'!k, P) :-
  P = C [E!j D'!i] [D!i], junction(E!j, E'!k, P).
```

(a) Visual DSD program code for the PI controller circuit implemented with combined protected catalysis and junction annihilation (continued on next page)

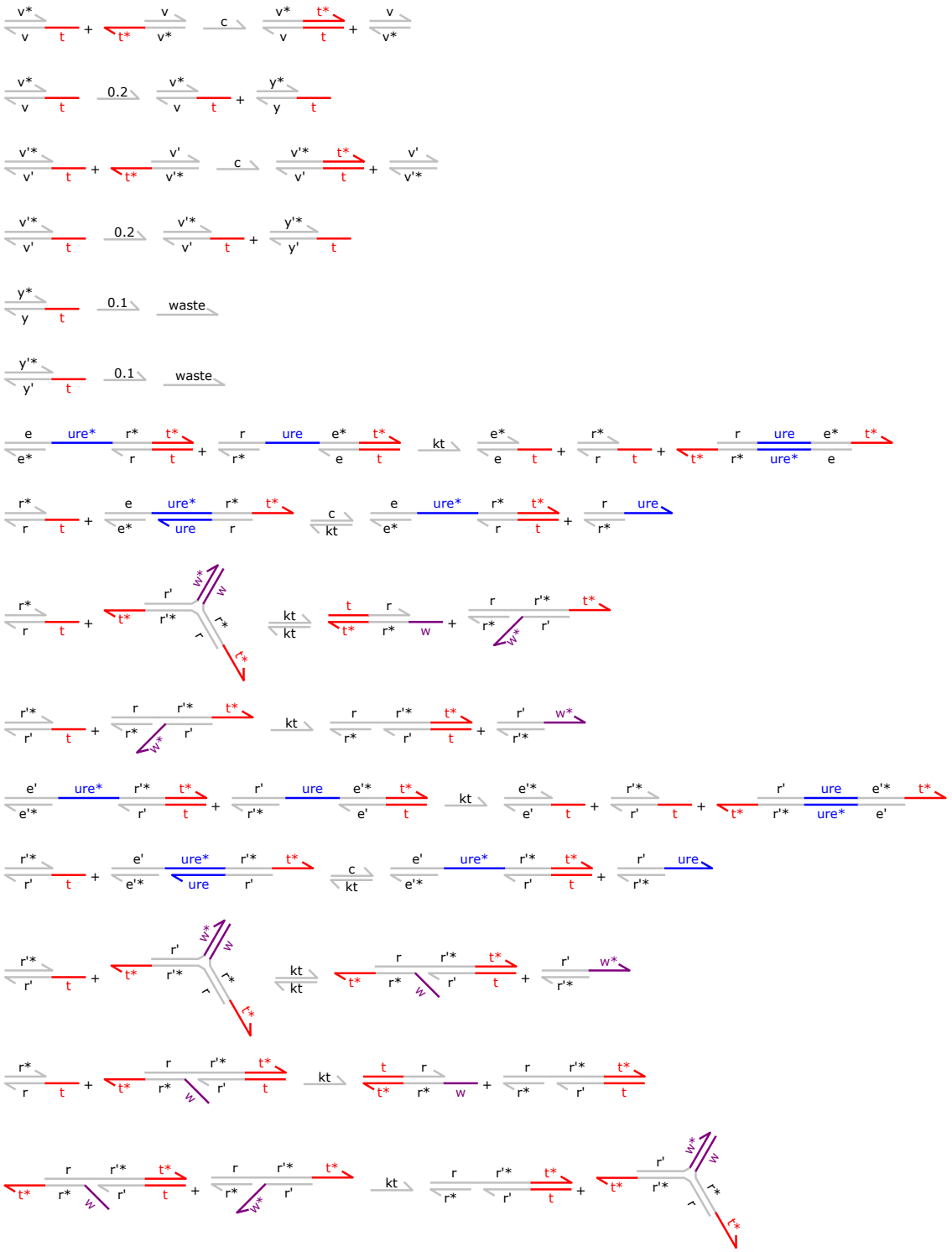
```
junctionL(D!i, D'!i, P).
junctionL(D!i, E'!k, P) :-
  P = C [D'!i E!j ] [D!i], junctionL(E!j, E'!k, P).

junctionM(D!i, D'!i, P, L).
junctionM(D!i, E'!k, P, L) :-
  not (i=k), P = C [E!j D'!i] [D!i], not member(j,L),
  junctionM(E!j, E'!k, P, [j#l]).

// Interactions on consecutive domains
binds(P1,P2,R,D!i,[D#L]) :- bind(P1,P2,Q,D!i),
  not coverL(Q,_,D!i,_), covers(Q,R,D!i,L).
binds(P1,P2,Q,D!i,[D]) :- bind(P1,P2,Q,D!i),
  not coverL(Q,_,D!i,_), not cover(Q,_,D!i,_).

unbinds(P,R,D!i,[D#L]) :-
  P = C [D!i E!j] [E'!j D'!i], toehold(D),
  not boundL(D!i,_,P),
  Q = C [D E!j] [E'!j D'], unbinds(Q,R,E!j,L).
boundL(D!i,E!j,P) :- P = C [E!j D!i] [D'!i E'!j].
unbinds(P,Q,D!i,[D]) :- unbind(P,Q,D!i).

displaces(P,R,E!j,[D#L]) :- displace(P,Q,E!j,D!i),
  displaces(Q,R,D!i,L).
displaces(P,Q,E!j,[D]) :- displace(P,Q,E!j,D!i),
  not displace(Q,_,D!i,_).
displacesL(P,R,E!j,[D#L]) :- displaceL(P,Q,E!j,D!i),
  displacesL(Q,R,D!i,L).
displacesL(P,Q,E!j,[D]) :- displaceL(P,Q,E!j,D!i),
  not displaceL(Q,_,D!i,_).

covers(P,R,E!j,[D#L]) :- cover(P,Q,E!j,D!i),
  covers(Q,R,D!i,L).
covers(P,Q,E!j,[D]) :- cover(P,Q,E!j,D!i),
  not cover(Q,_,D!i,_).
coversL(P,R,E!j,[D#L]) :- coverL(P,Q,E!j,D!i),
  coversL(Q,R,D!i,L).
coversL(P,Q,E!j,[D]) :- coverL(P,Q,E!j,D!i),
  not coverL(Q,_,D!i,_).

migrates(P,R,E!j,[D#L]) :- migrate(P,Q,E!j,D!i),
  migrates(Q,R,D!i,L).
migrates(P,Q,E!j,[D]) :- migrate(P,Q,E!j,D!i),
  not migrate(Q,_,D!i,_).

leak(P1,P2,Q,D!i) :- doLeak(P1, P2, Q, D!i).
leak(P1,P2,Q,D!i) :- doLeak(P2, P1, Q, D!i).
clamped(D!i,P):- P = C [D1!i1 D!i D2!i2]
   [D2'!i2 D'!i D1'!i1].
doLeak(P1,P2,Q,D!i) :-
  P1 = C1 [D!i] [D'!i], P2 = C2 [D],
  not clamped(D!i,P1),
  Q  = C1 [D] [D'!i] | C2 [D!i].

// Label reactions as either slow or fast
slow(P1, P2, Rate, Q) :- binds(P1,P2,Q,D!i,_),
  find(D, "bind", Rate).
//slow(P1, P2, Rate, Q) :- leak(P1,P2,Q,D!i),
//  find(D, "leak", Rate). // Full leaks

fast(P, Rate, Q) :- covers(P,Q,D!i,L),
  find(L, "cover", Rate).
fast(P, Rate, Q) :- coversL(P,Q,D!i,L),
  find(L, "cover", Rate).
fast(P, Rate, Q) :- displaces(P,Q,D!i,L),
  find(L, "displace", Rate).
fast(P, Rate, Q) :- displacesL(P,Q,D!i,L),
  find(L, "displace", Rate).
fast(P, Rate, Q) :- unbinds(P,Q,D!i,L),
  find(L, "unbind", Rate).
fast(P, Rate, Q) :- migrates(P,Q,D!i,L),
  find(L, "migrate", Rate).

// Merge fast reations
allFastReactionsFromInner([], _, []).
allFastReactionsFromInner([Q # ToVisit], V, [[Q; Rs] # G']) :-
  findAll(R, fast(Q, _, R), Rs),
  difference(Rs, [Q # V], Rs'),
  difference(Rs', ToVisit, Rs''),
  append(ToVisit, Rs'', ToVisit'),
  allFastReactionsFromInner(ToVisit', [Q # V], G').
allFastReactionsFrom(Q, G) :-
  allFastReactionsFromInner([Q], [], G).

merge(Q, R) :-
  allFastReactionsFrom(Q, G),
  tscc(G, TS),
  member([R # _], TS).
merge(Q, Q) :- not fast(Q, _,_).

// Normalize species
dL(P,Q) :- displaceL(P, Q, D!i, E!j).
normalize(R, R) :- not dL(R, _).
normalize(P, R) :- displaceL(P, Q, D!i, E!j), normalize(Q, R).

// Reject complexes P1 and P2 with total number of strands
// greater than some fixed integer
sizeLimit(P1,P2) :-
  toList(P1, L1), length(L1, N1), toList(P2, L2), length(L2, N2),
  N is N1 + N2, N <= 8.

// Generate reactions
reaction([P1; P2], Rate, Rn) :-
  sizeLimit(P1,P2),
  slow(P1, P2, Rate, Q), merge(Q, R), normalize(R, Rn).
reaction([P], Rate, Rn) :- slow(P, Rate, Q), merge(Q, R),
  normalize(R, Rn).

// Plant reactions (protected)
reaction([<t^ v!0> | <v*!0>], "0.2", (<t^ v!0>
  |<v*!0> | <t^ y!1> |<y*!1>) ).
reaction([<t^ v'!0> | <v'*!0>], "0.2", (<t^ v'!0>
  |<v'*!0> | <t^ y'!1> |<y'*!1>) ).
reaction([<t^ y!0> | <y*!0>], "0.1", <waste> ).
reaction([<t^ y'!0> | <y'*!0>], "0.1", <waste> ).
reaction([(<t^ y!0> | <y*!0>); (<t^ y'!0> | <y'*!0>)],
  "0.1", <waste> ).
reaction([<load>; (<t^ y!0> | <y*!0>)], "0.01", <load> ).
reaction([<load'>; (<t^ y'!0> | <y'*!0>)], "0.01", <load'> ).
reaction([<load>; <load'>], "0.1", <waste> ).
}
```

(b) Visual DSD program code for the PI controller circuit implemented with combined protected catalysis and junction annihilation (continued on next page)

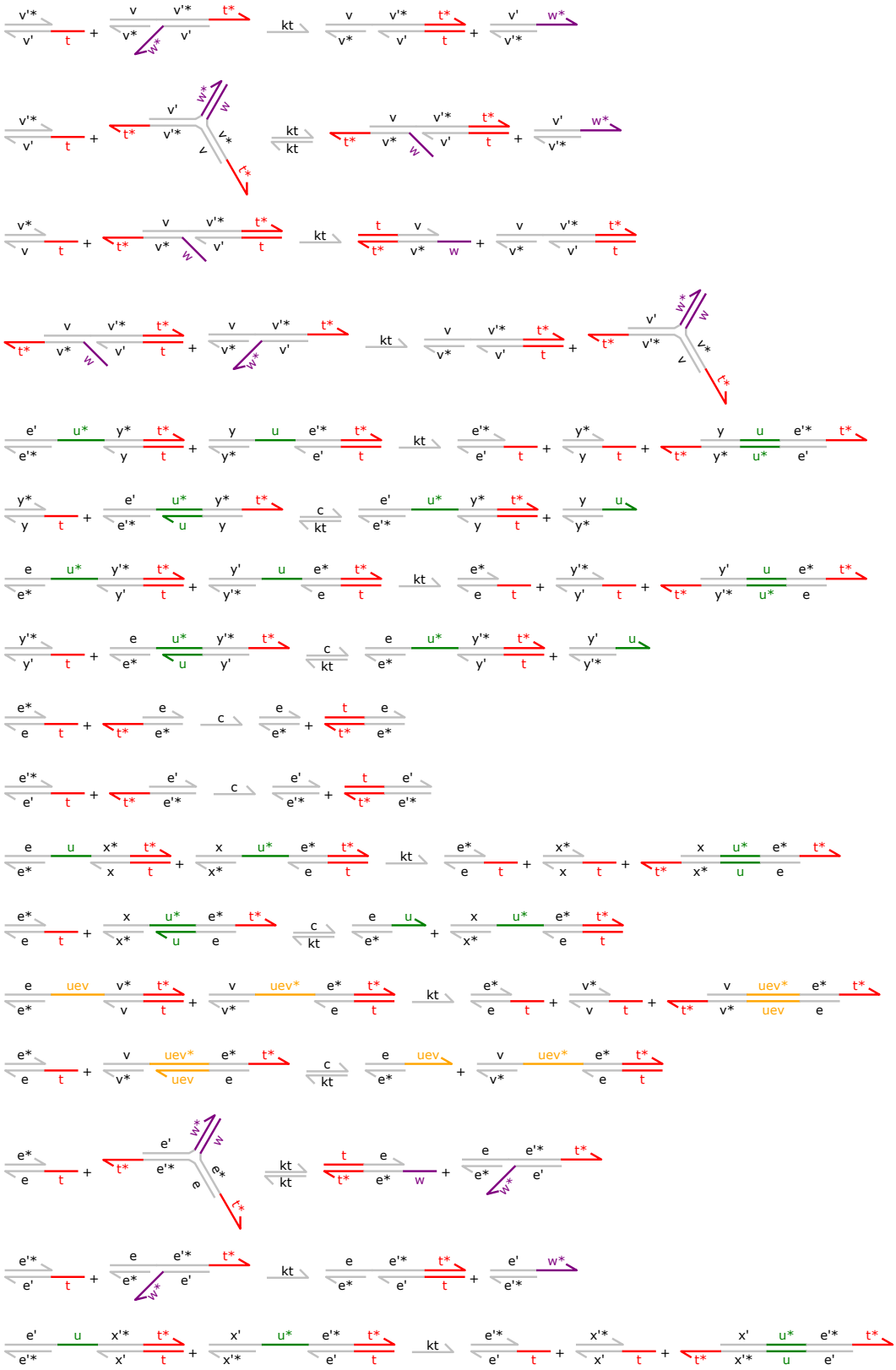
```
directive deterministic { stiff=true }
directive simulator sundials
directive parameters [
  Cmax = 1000; c = 0.0000008; kt = 0.001;
  unbind = 0.1; displace = 1; cover = 1;
  migrate = 1; leak = 1E-09; x0 = 4; ]
directive declare
directive simulation { final=150000; plots=[ [R()]-[R'()];
[V()]-[V'()]; [Y()]-[Y'()]; [L()]-[L'()] ] }

dom t = {colour = "red"; bind = kt}
dom u = {colour = "green"; bind = kt}
dom w = {colour = "blue"; bind = kt}
dom ure = {colour = "green"; bind = kt}
dom uev = {colour = "green"; bind = kt}

new x new x' new v new v' new y new y' new r new r'
new e new e' new load new load' new i

//Signal
def Signal((x,x')) = <t^>[x] // protected

//Degradation
def Degradation(k,(x,x')) = Cmax/k * {t^(c)*}[x]

//Annihilation junction protected
def Annihilation((y,y')) =
( Cmax * [<w^!0 y*!1 t^*> | <y!1 y'*!2 t^*> | <y'!2 w^*!0>]
| Cmax * [t^ y]{w^} | Cmax * {w^*}[y'*])
def AnnihilationR((r,r')) =
( Cmax * [<w^!0 r*!1 t^*> | <r!1 r'*!2 t^*> | <r'!2 w^*!0>]
| Cmax * [t^ r]{w^} | Cmax * {w^*}[r'*])
def AnnihilationX((x,x')) =
( Cmax * [<w^!0 x*!1 t^*> | <x!1 x'*!2 t^*> | <x'!2 w^*!0>]
| Cmax * [t^ x]{w^} | Cmax * {w^*}[x'*])
def AnnihilationV((v,v')) =
( Cmax * [<w^!0 v*!1 t^*> | <v!1 v'*!2 t^*> | <v'!2 w^*!0>]
| Cmax * [t^ v]{w^} | Cmax * {w^*}[v'*])
def AnnihilationE((e,e')) =
( Cmax * [<w^!0 e*!1 t^*> | <e!1 e'*!2 t^*> | <e'!2 w^*!0>]
| Cmax * [t^ e]{w^} | Cmax * {w^*}[e'*])

//Catalysis combined protected
def Catalysis(k,(x,x'),(y,y'),u) =
( Cmax*k * {t^(c)*}[x u^]:[y*]
| Cmax*k * [t^ y]{u^}:[x*] )

def Signal'((x,x')) = Signal((x',x))
def Degradation'(k,(x,x')) = Degradation(k,(x',x))
def Catalysis'(k,(x,x'),(y,y'),u) =
  Catalysis(k,(x',x),(y',y),u)
def CatalysisInv(k,x,(y,y'),u) = Catalysis(k,x,(y',y),u)
def CatalysisInv'(k,(x,x'),y,u) = Catalysis(k,(x',x),y,u)

def x = (x,x')
def y = (y,y')
def v = (v,v')
def e = (e,e')
def r = (r,r')

//---------------- Blocks ----------------
def Catalysis2(k,x,y,u) =
( Catalysis(k,x,y,u) | Catalysis'(k,x,y,u) )
def Catalysis2Inv(k,x,y,u) =
( CatalysisInv(k,x,y,u) | CatalysisInv'(k,x,y,u) )
def Degradation2(k,x) =
( Degradation(k,x) | Degradation'(k,x) )
def Integration(kI,x,y,u) = Catalysis2(kI,x,y,u)
def Gain(k,kD,x,y,u) =
( Catalysis2(k,x,y,u) | Degradation2(kD,y) )
def Summation(k1,k2,kD,x1,x2,y,u1,u2) =
( Catalysis2(k1,x1,y,u1)
| Catalysis2(k2,x2,y,u2)
| Degradation2(kD,y)
)
def SummationInv(k1,k2,kD,x1,x2,y,u1,u2) =
( Catalysis2(k1,x1,y,u1)
| Catalysis2Inv(k2,x2,y,u2)
| Degradation2(kD,y)
)
def PI(kP,kI, r, y, v,e) =
( SummationInv(1,1,1,r,y,e,ure,u)
| Integration(kI,e,x,u)
| Summation(kP,1,1,e,x,v,uev,u)
| AnnihilationX(x)
)

def R() = Signal(r)
def R'() = Signal'(r)
def E() = Signal(e)
def E'() = Signal'(e)
def V() = Signal(v)
def V'() = Signal'(v)
def Y() = Signal(y)
def Y'() = Signal'(y)
def X() = Signal(x)
def X'() = Signal'(x)
def L() = <load>
def L'() = <load'>
def Plant(v, y) =
( rxn Signal(v) ->{0.2} Signal(v) + Signal(y)
| rxn Signal'(v) ->{0.2} Signal'(v) + Signal'(y)
| rxn Signal(y) ->{0.1}
| rxn Signal'(y) ->{0.1}
| rxn Signal'(y) + Signal(y) ->{0.1}
| rxn L() + Signal(y) ->{0.01} L()
| rxn L'() + Signal'(y) ->{0.01} L'()
| rxn L() + L'() -> {1.0}
)

//---------------- Full system ----------------
def System() =
( 0 * V() | 0 * V'()
| 0 * Y() | 0 * Y'()
| 0 * L() | 0 * L'()
| x0* R() | 0 * R'()
| PI(1,1,r,y,v,e)
| Plant(v,y)
| AnnihilationR(r)
| AnnihilationV(v)
| AnnihilationE(e)
)
def VarySignal() =
  ( 2*x0 * R'() @ 50000 | 2*x0 * R() @ 100000 )

//---------------- Run ----------------
( System() | VarySignal() )
```

(c)

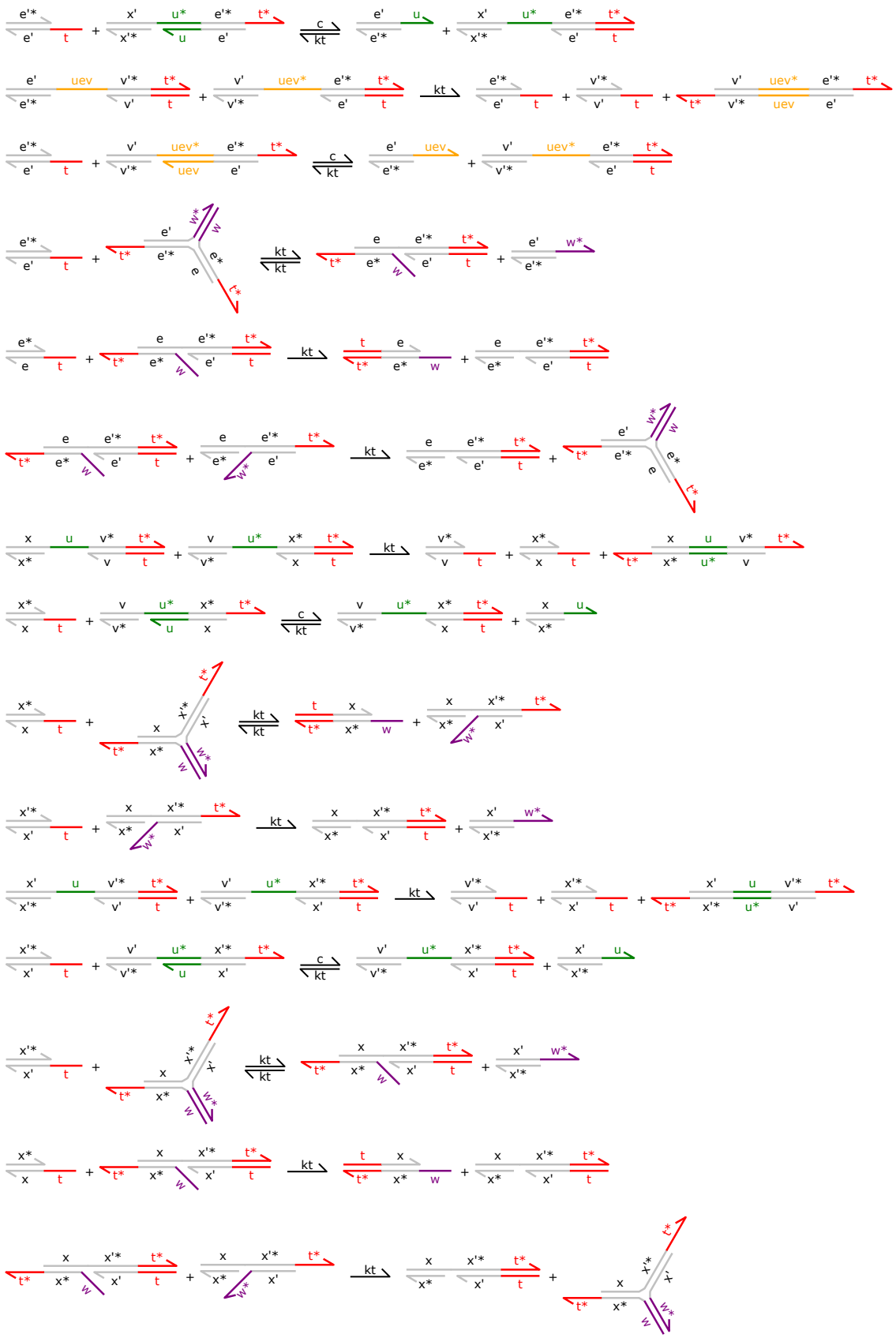
Figure S25: Visual DSD program code for the PI controller circuit implemented with combined protected catalysis and junction annihilation.



(a) Reactions of the PI controller implemented with combined protected catalysis and junction annihilation (continued on next page).



(b) Reactions of the PI controller implemented with combined protected catalysis and junction annihilation (continued on next page)



(c)

Figure S26: Reactions of the PI controller implemented with combined protected catalysis and junction annihilation.

S5.3 Additional Information

All simulations were performed on PC with the following characteristics:

Processor Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz 2.30 GHz

Installed RAM 16.0 GB (15.9 GB usable)

System type 64-bit operating system, x64-based processor

Edition Windows 10 Education Version 20H2

Table S4 illustrates the times required for completing each simulation. All indicated simulations for 4-domain and 2-domain control circuits were performed in infinite mode with the maximum complex size N set to 8; for SLD and DLD circuits of Wang et al.[7] – in detailed mode with N set to 6.

Scheme	Circuit	Duration	Improvement
4-domain Full leaks	Catalysis	00:00:14	
	Degradation	00:00:01	
	Annihilation	00:00:53	
	Gain	00:33:26	
4-domain First-order leaks	Catalysis	00:00:03	71%
	Degradation	00:00:01	0%
	Annihilation	00:00:17	68%
	Gain	00:02:20	93%

(a) 4-domain scheme

Scheme	Circuit	Duration	Improvement
2-domain Full leaks	Catalysis	00:10:00	
	Degradation	00:00:01	
	Annihilation	00:00:41	
	Gain	00:50:53	
2-domain First-order leaks	Catalysis	00:02:02	80%
	Degradation	00:00:01	0%
	Annihilation	00:00:11	73%
	Gain	00:08:09	84%

(b) 2-domain scheme

Scheme	Circuit	Duration	Number of reactions
Wang et al.[7]	SLD leak	00:00:15	532
	DLD leak	00:00:04	183

(c) Molecular cascades

Circuit	Duration
Gain: original uncapped	00:11:44
Gain: original uncapped with protectors	00:10:20
Gain: direct	00:03:06
Gain: direct with protectors	00:02:24
Gain: combined	00:01:34
Gain: combined with protectors	00:00:41
Gain: combined capped with protectors	00:01:53
Gain: combined catalysis and original annihilation	00:15:00
PI controller: combined with protectors	00:19:31
PI controller: combined capped with protectors	00:46:05

(d) 2-domain circuits with reduced leakage, full leaks simulation. All circuits use branched annihilation unless stated otherwise.

Table S4: Simulation times for the tested circuits.

S6 CRNs for elementary reactions under no-leak and first order leak implementations

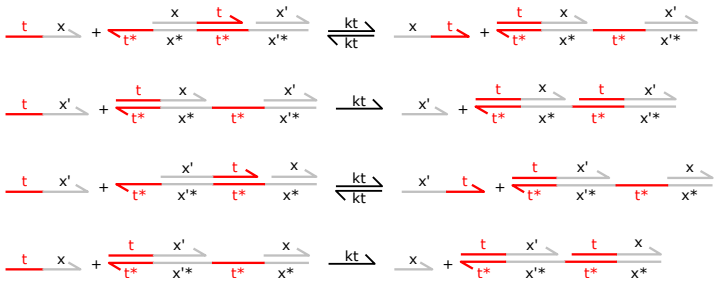
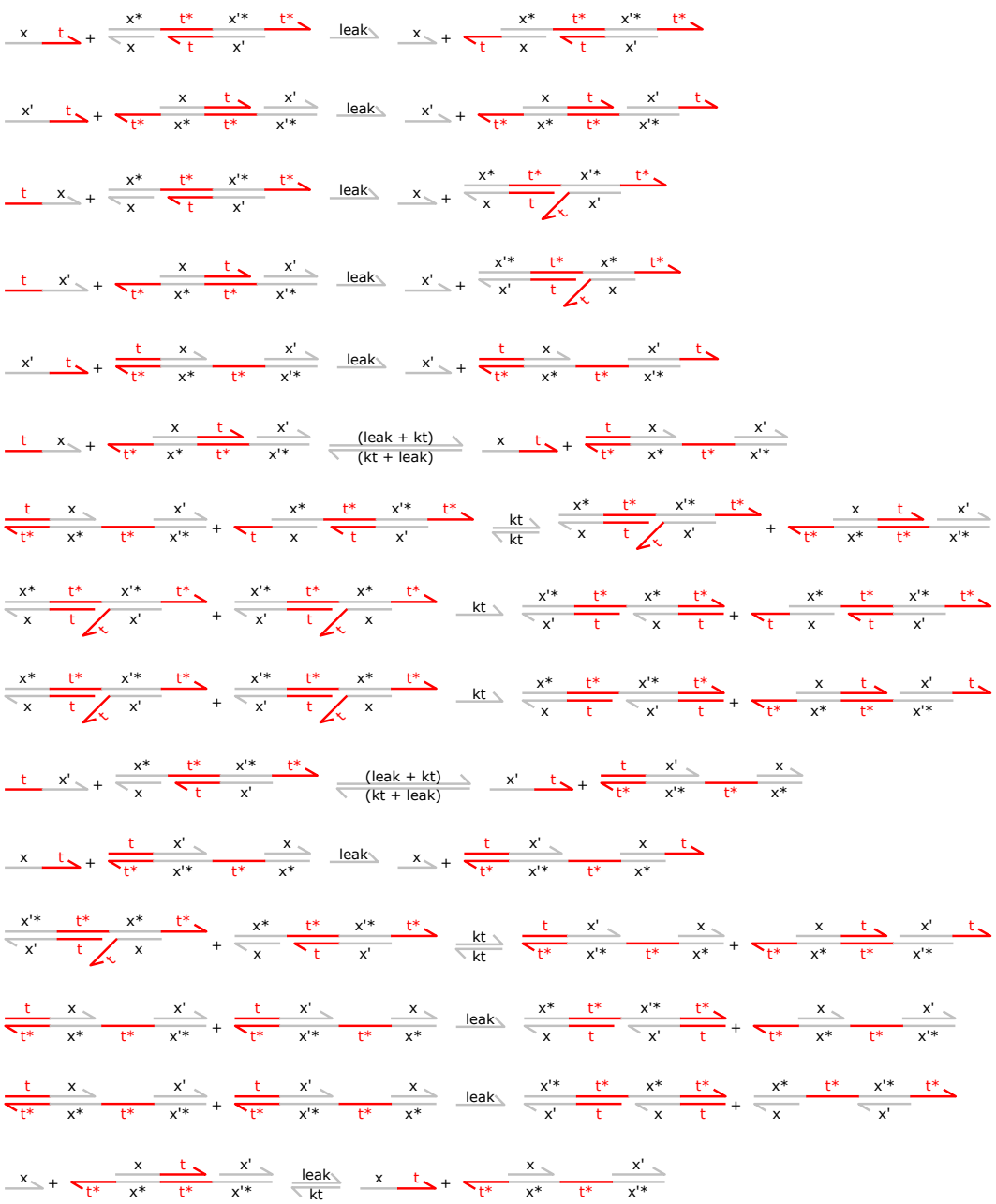
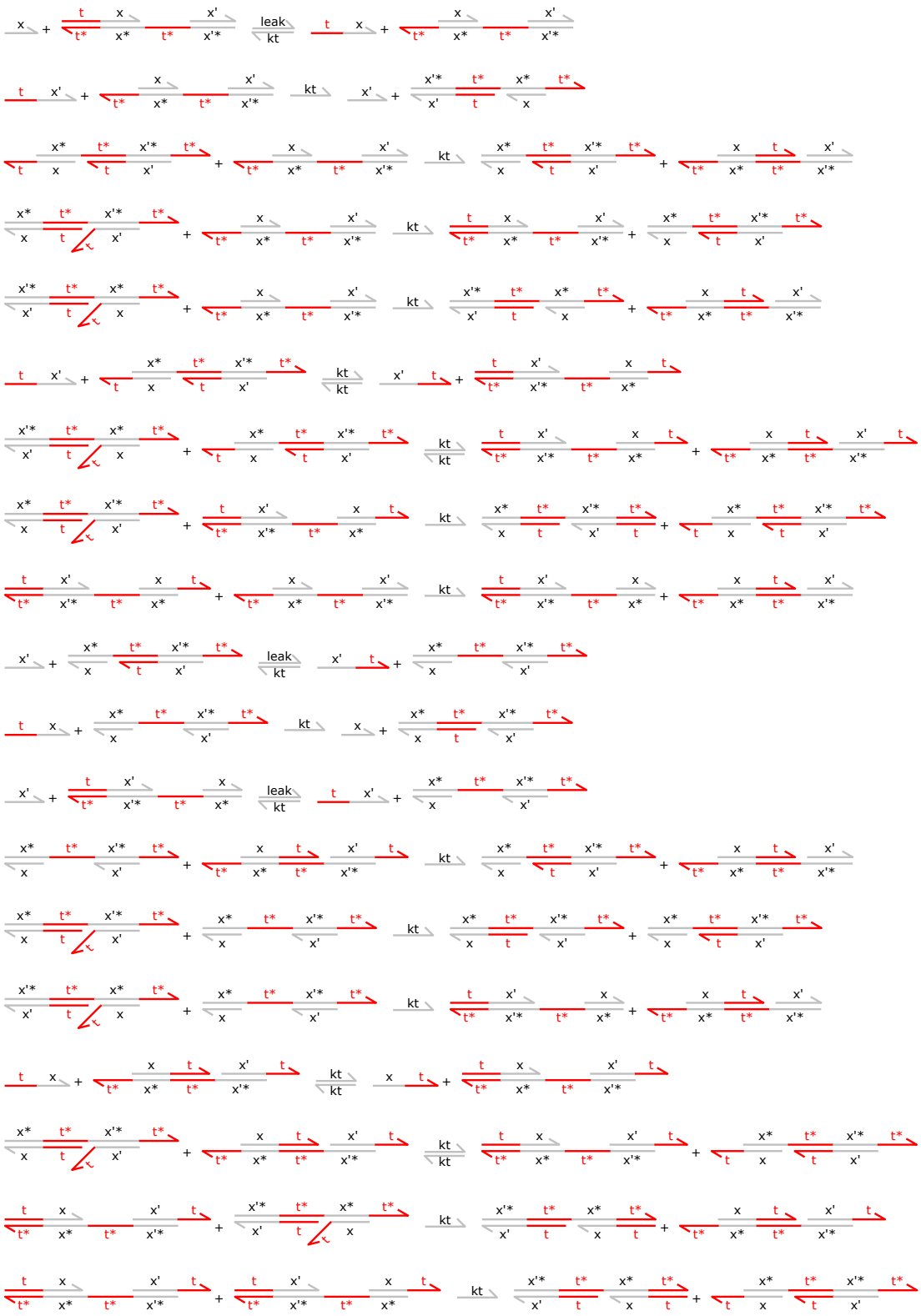


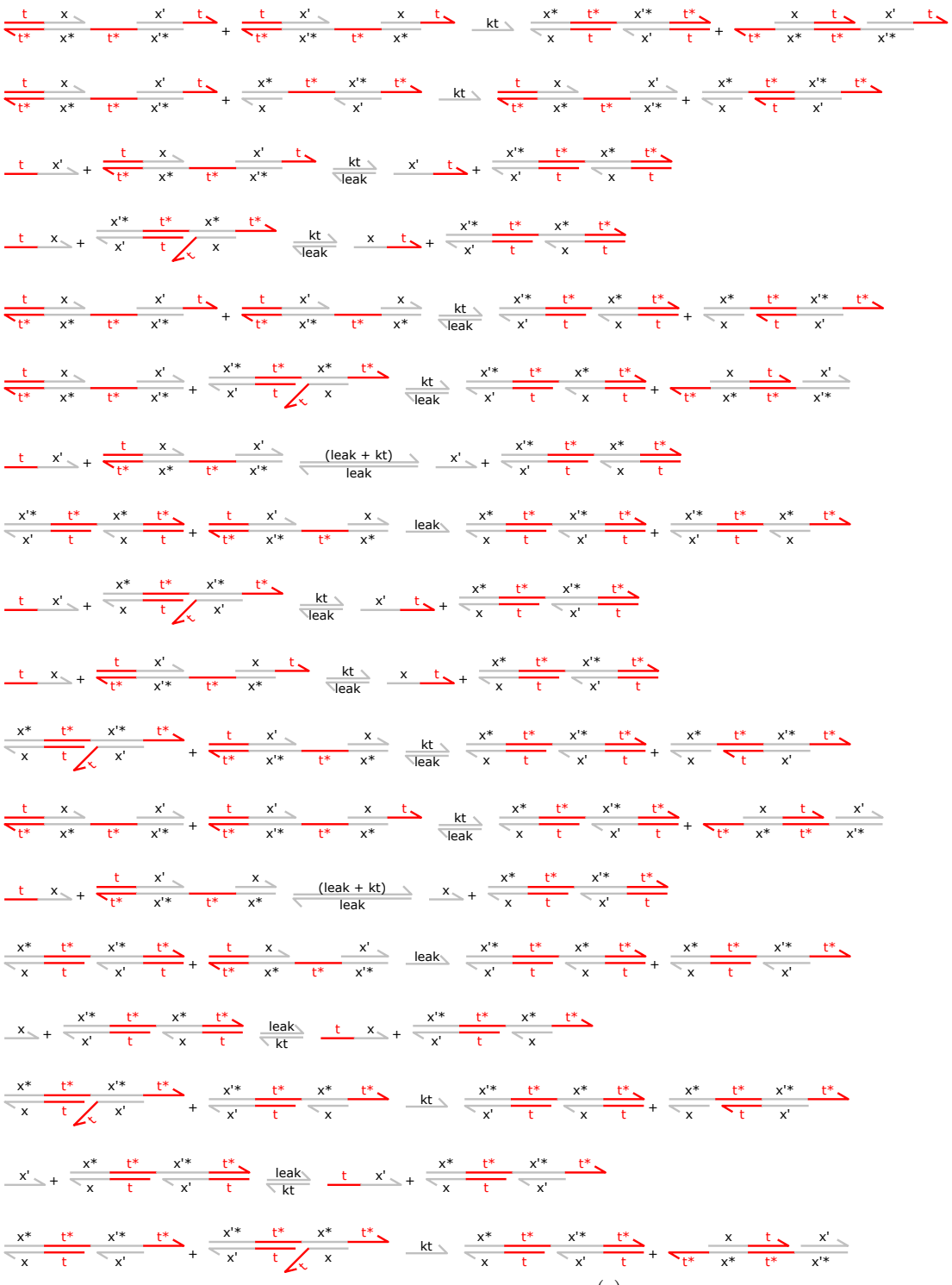
Figure S27: CRN for no-leak 2-domain annihilation circuit.



(a) CRN for 2-domain annihilation circuit with first order leaks (continued on next page).



(b) CRN for 2-domain annihilation circuit with first order leaks (continued on next page).



(c)

Figure S28: CRN for 2-domain annihilation circuit with first order leaks.

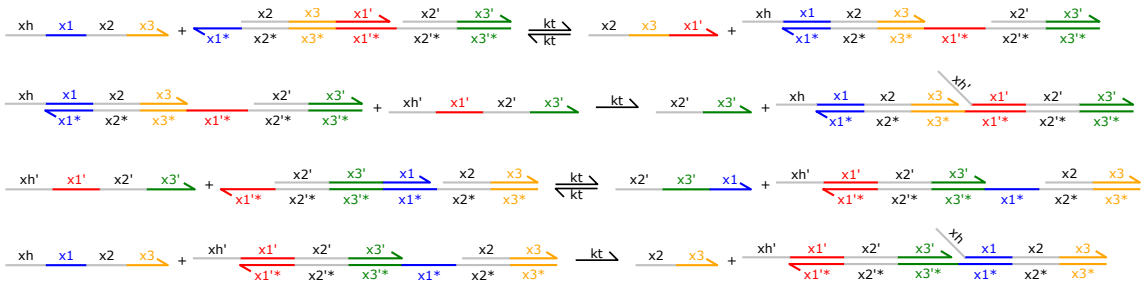
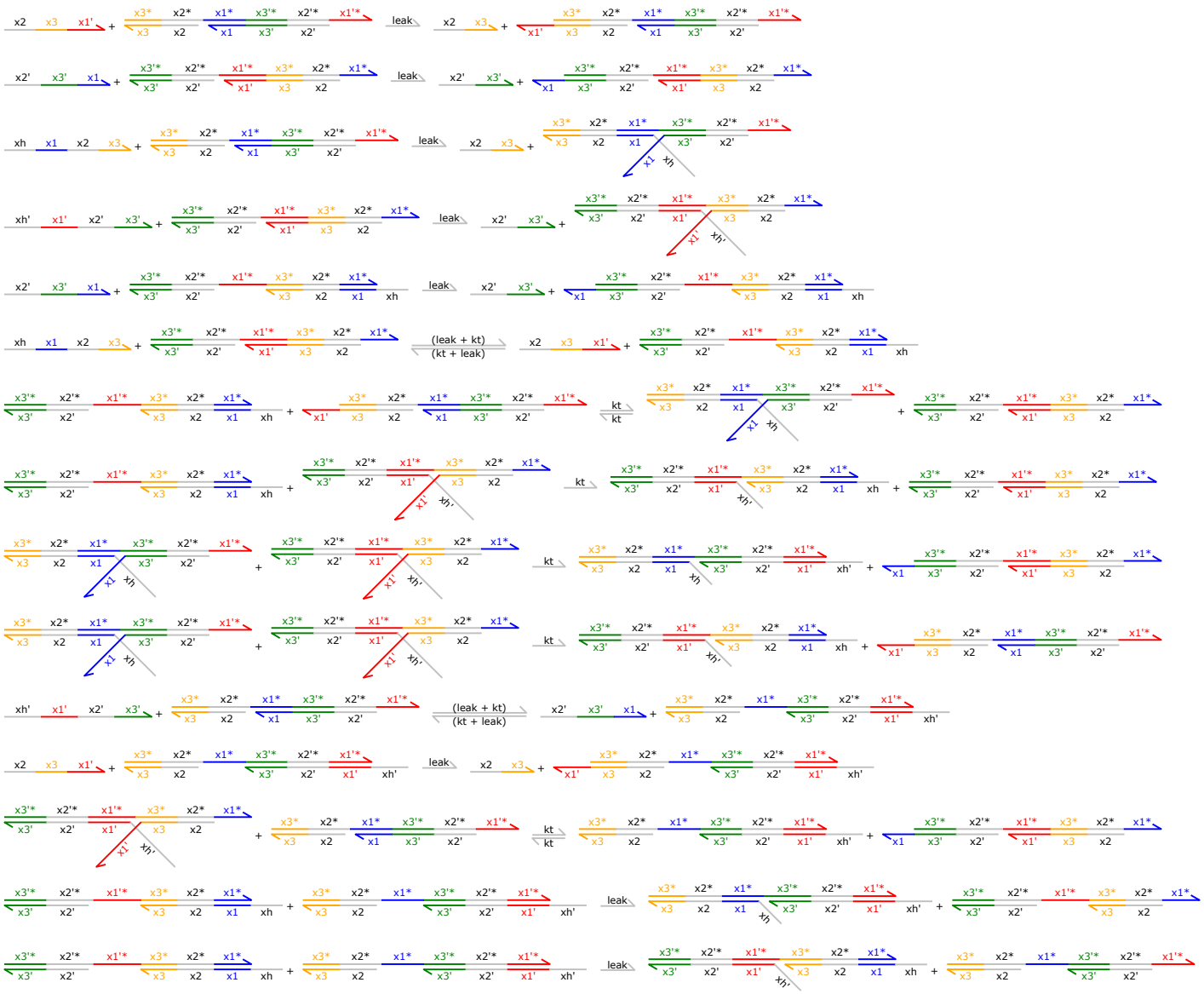
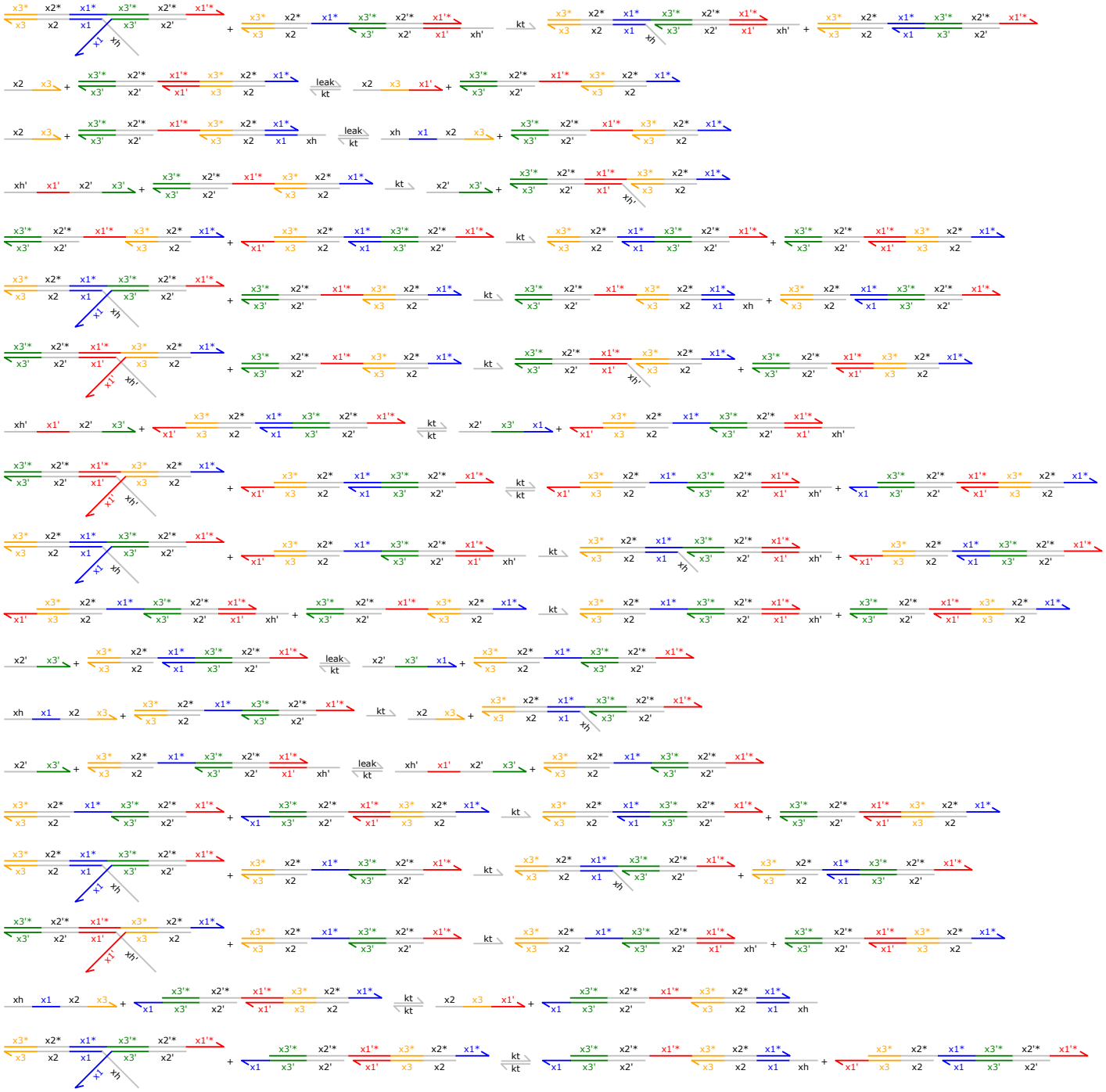


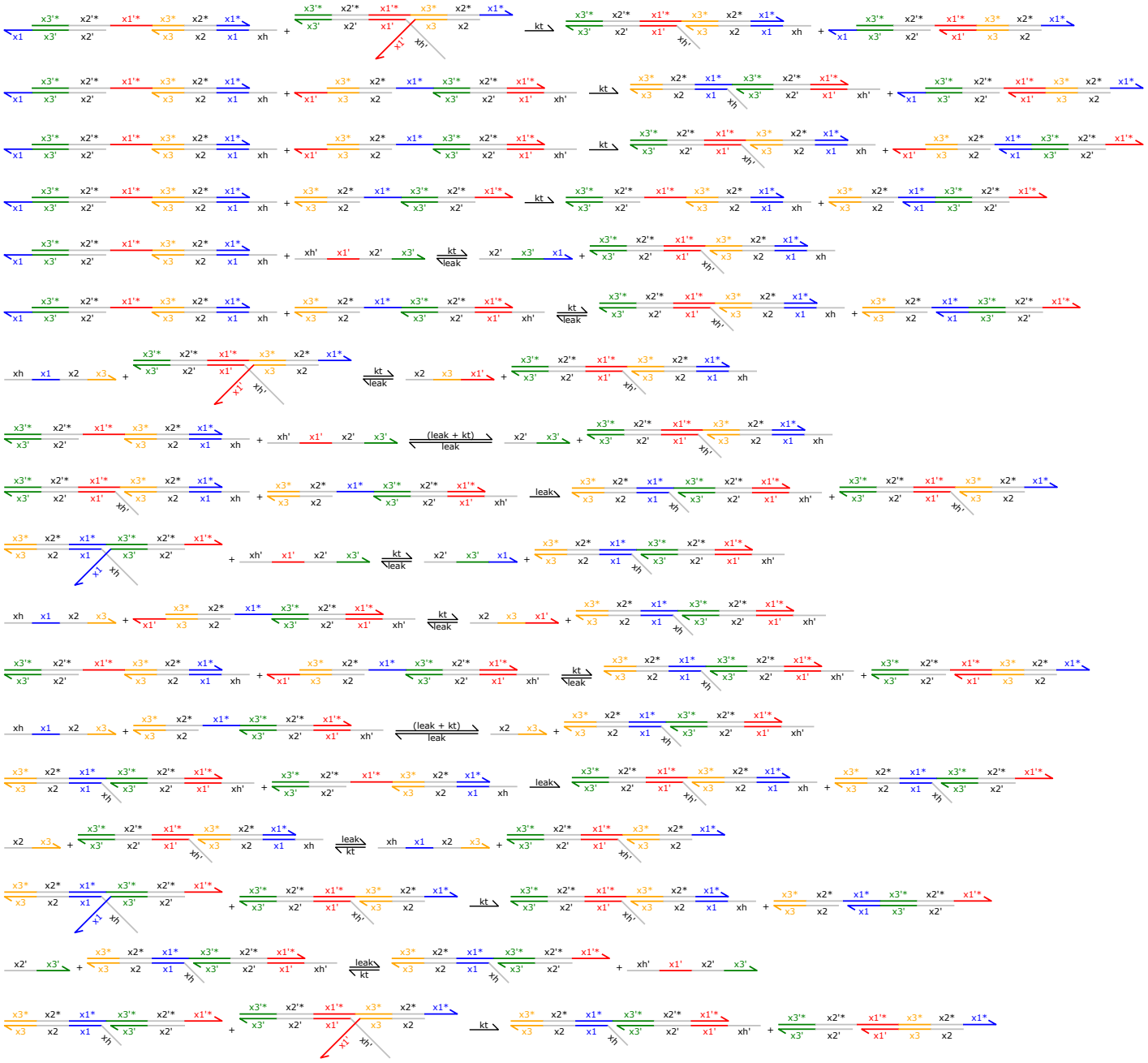
Figure S29: CRN for no-leak 4-domain annihilation circuit.



(a) CRN for 4-domain annihilation circuit with first order leaks (continued on next page).



(b) CRN for 4-domain annihilation circuit with first order leaks (continued on next page).



(c)

Figure S30: CRN for 4-domain annihilation circuit with first order leaks.

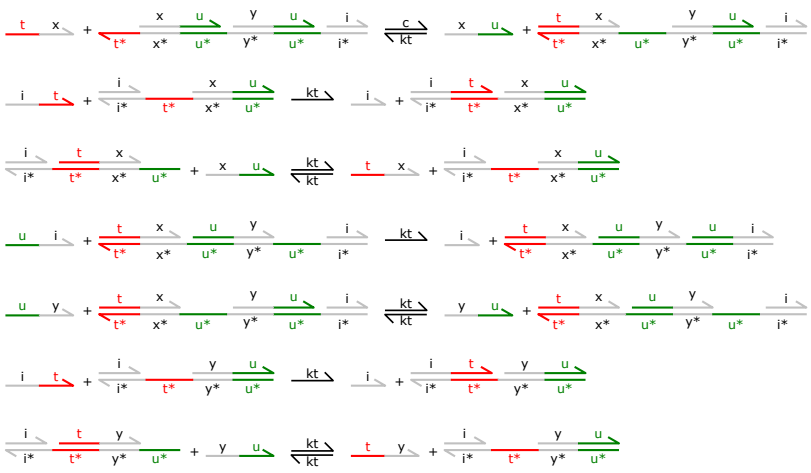
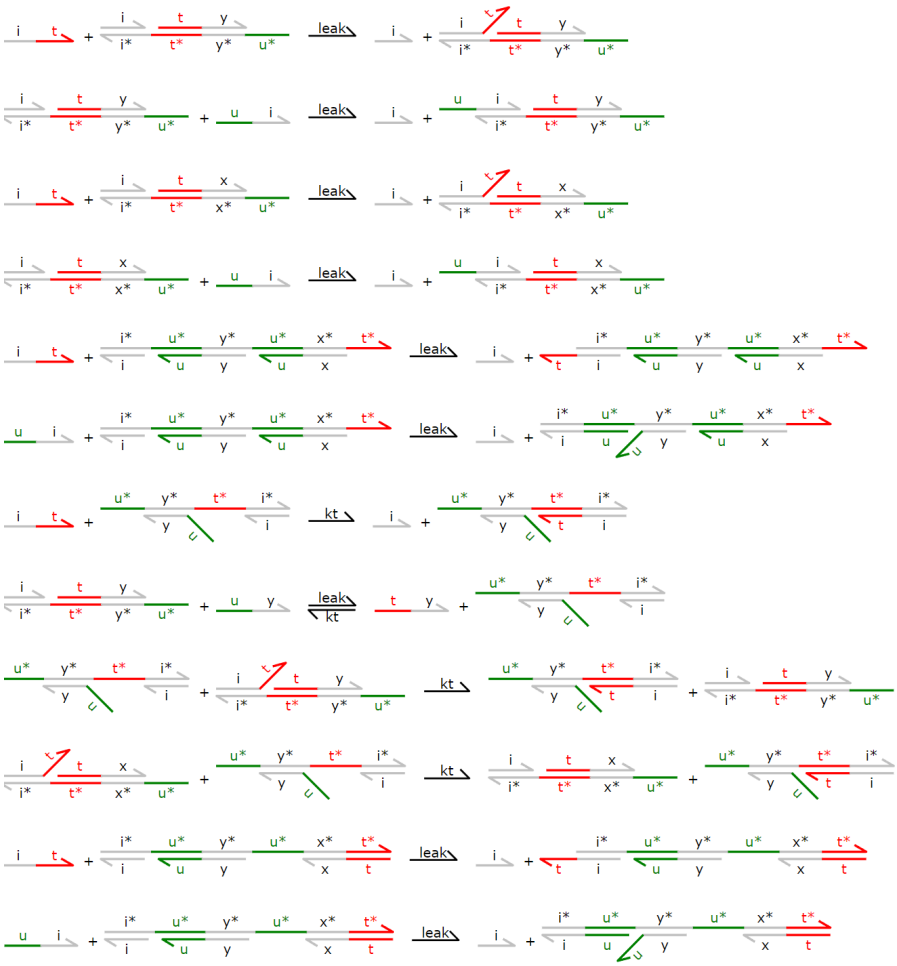
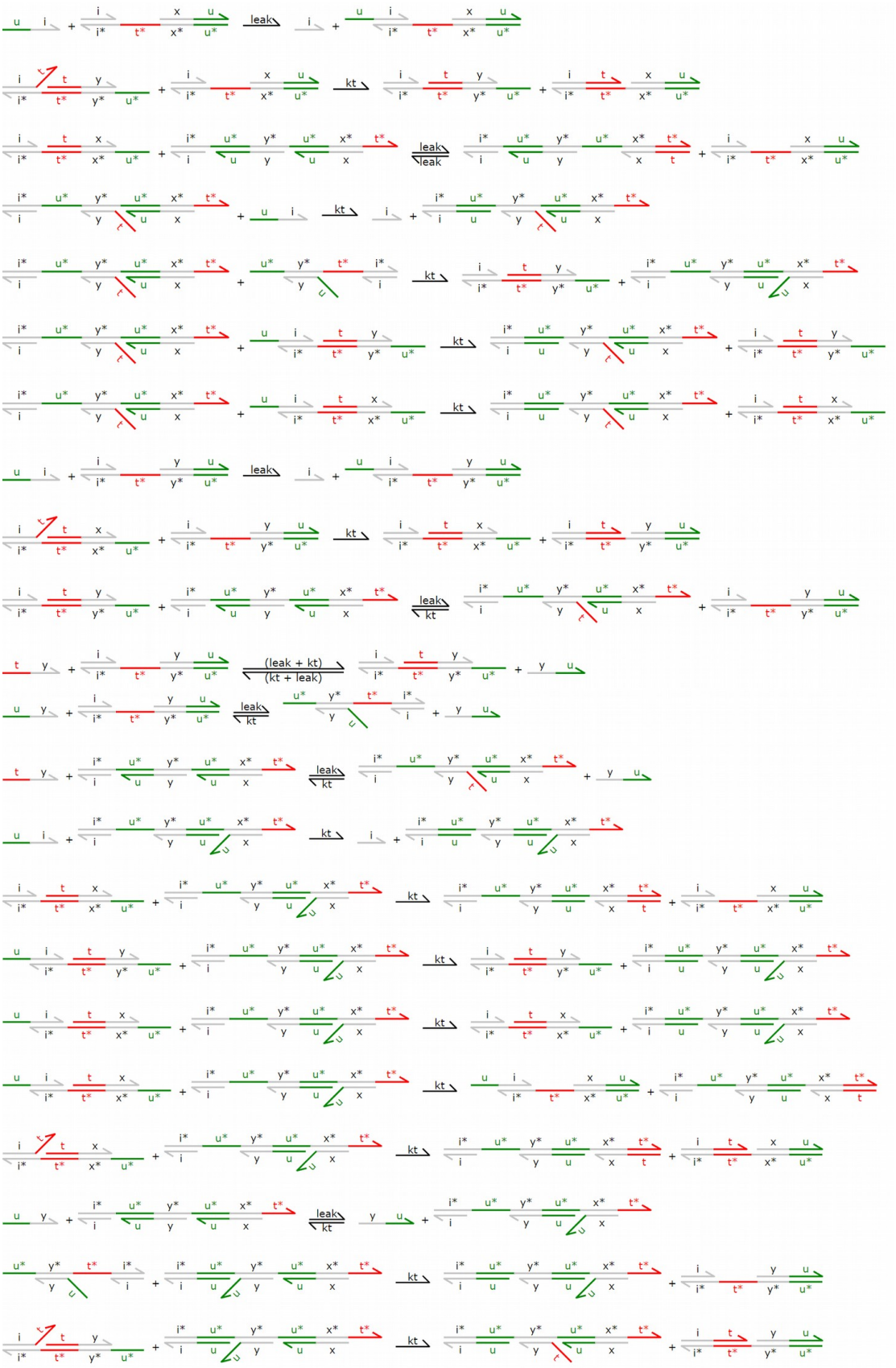


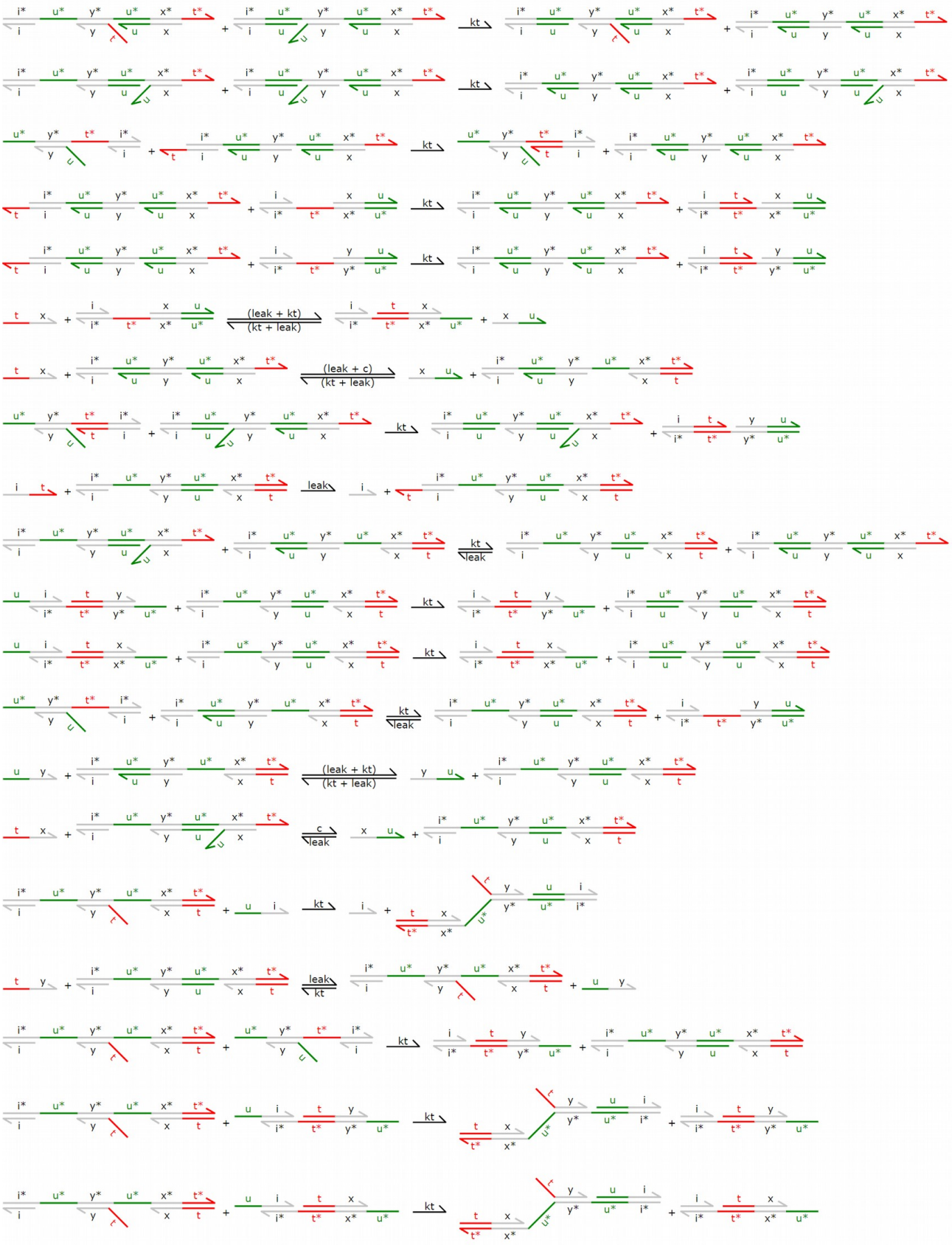
Figure S31: CRN for no-leak 2-domain catalysis circuit.



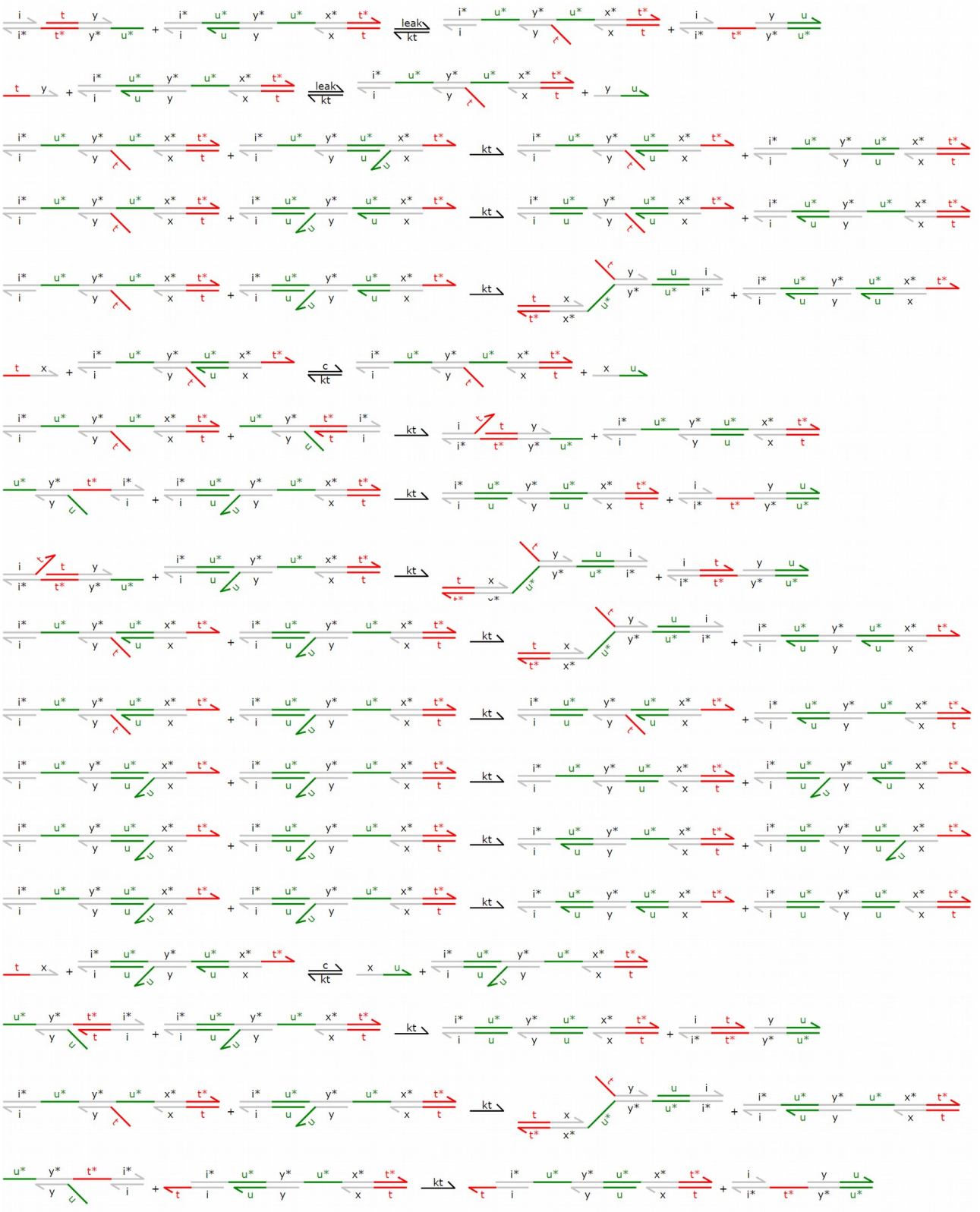
(a) CRN for 2-domain catalysis circuit with first order leaks (continued on next page).



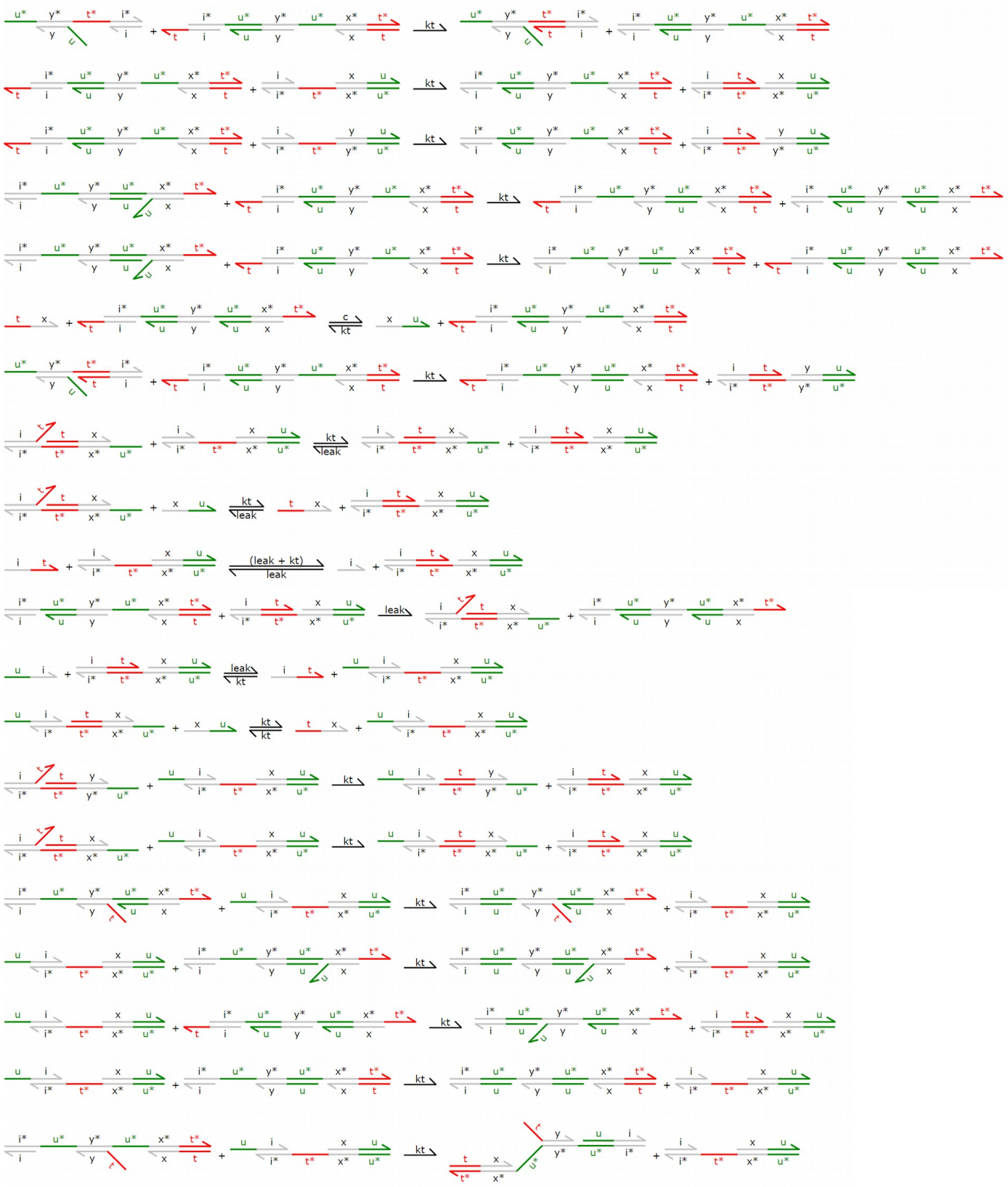
(b) CRN for 2-domain catalysis circuit with first order leaks (continued on next page).



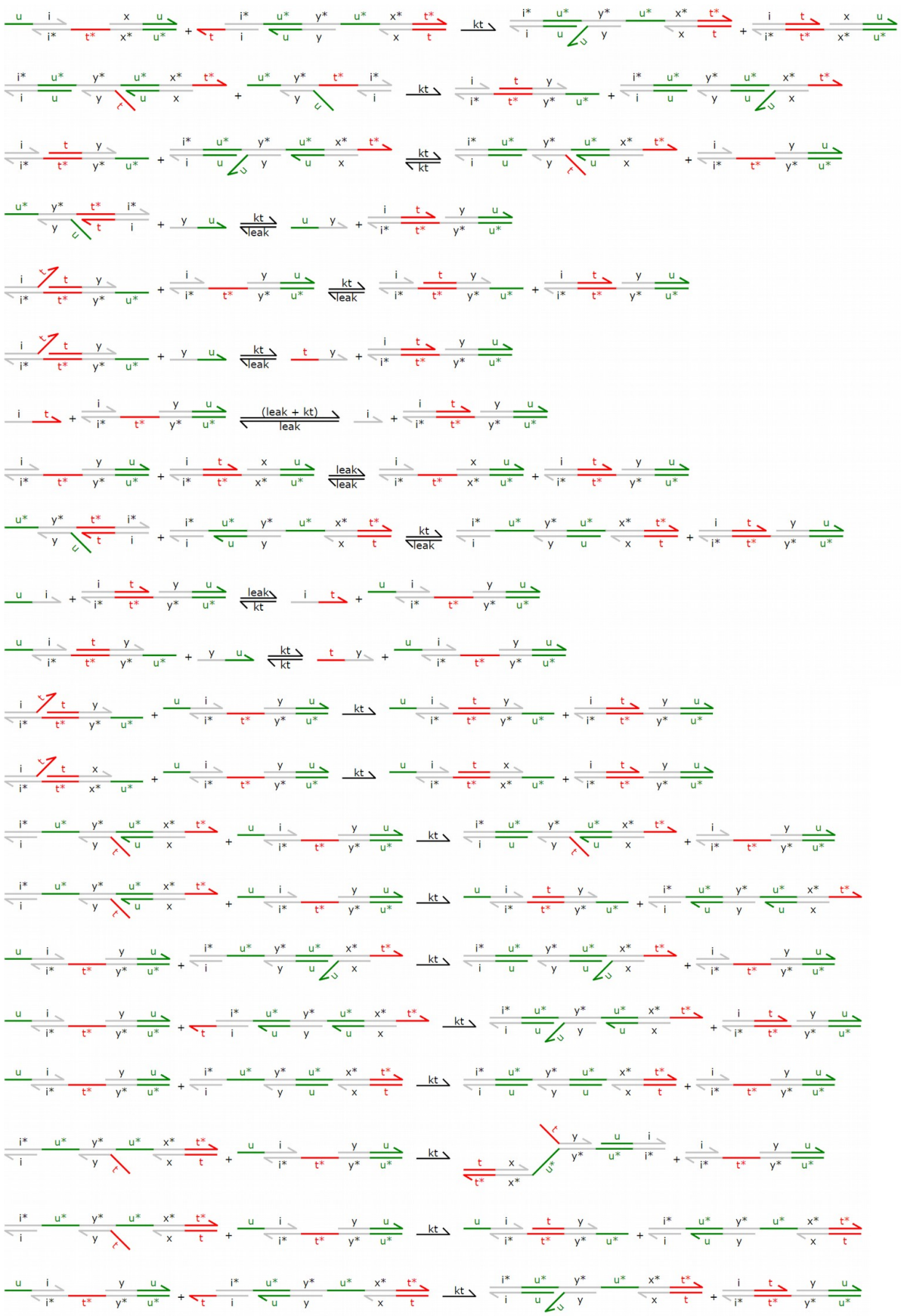
(c) CRN for 2-domain catalysis circuit with first order leaks (continued on next page).



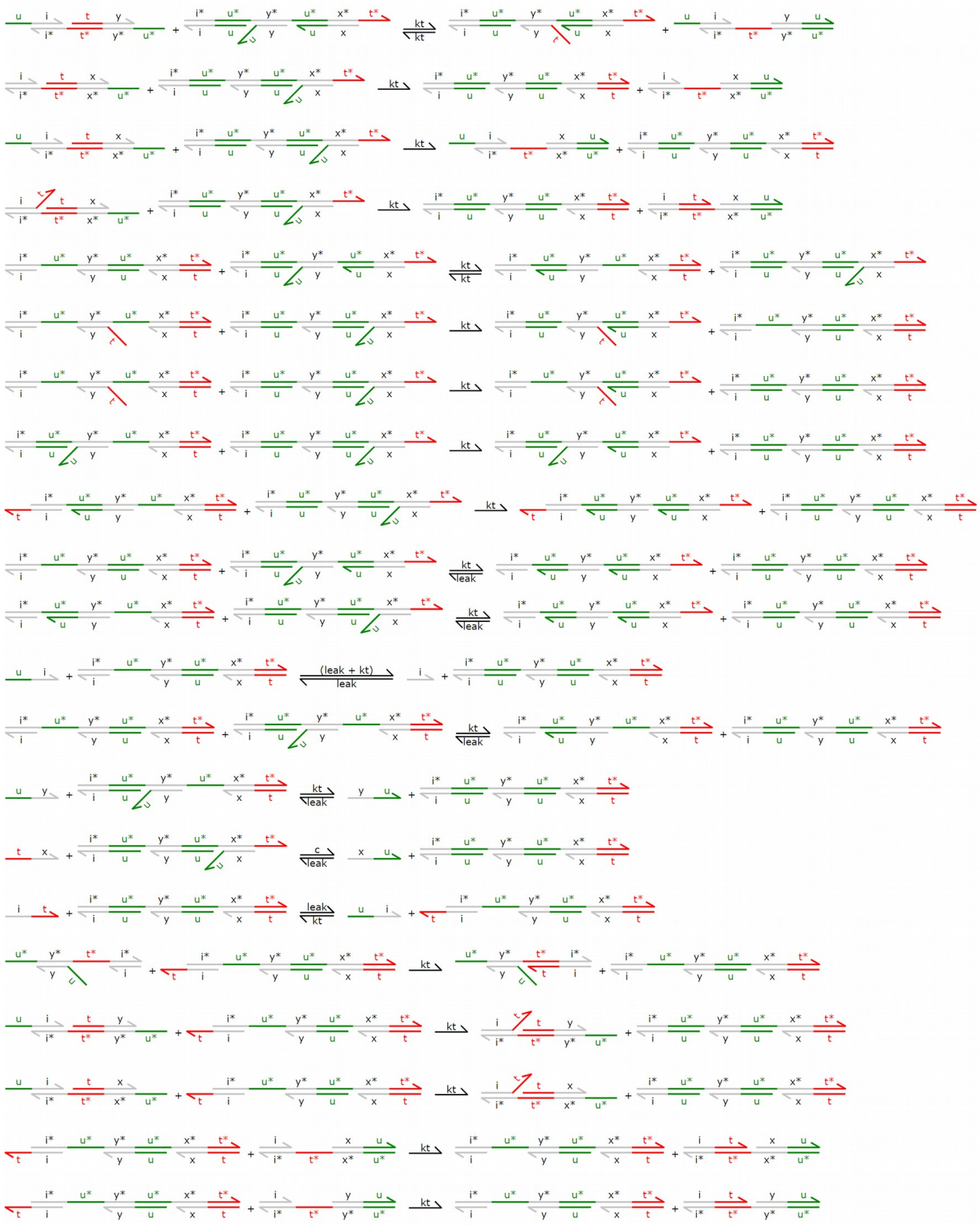
(d) CRN for 2-domain catalysis circuit with first order leaks (continued on next page).



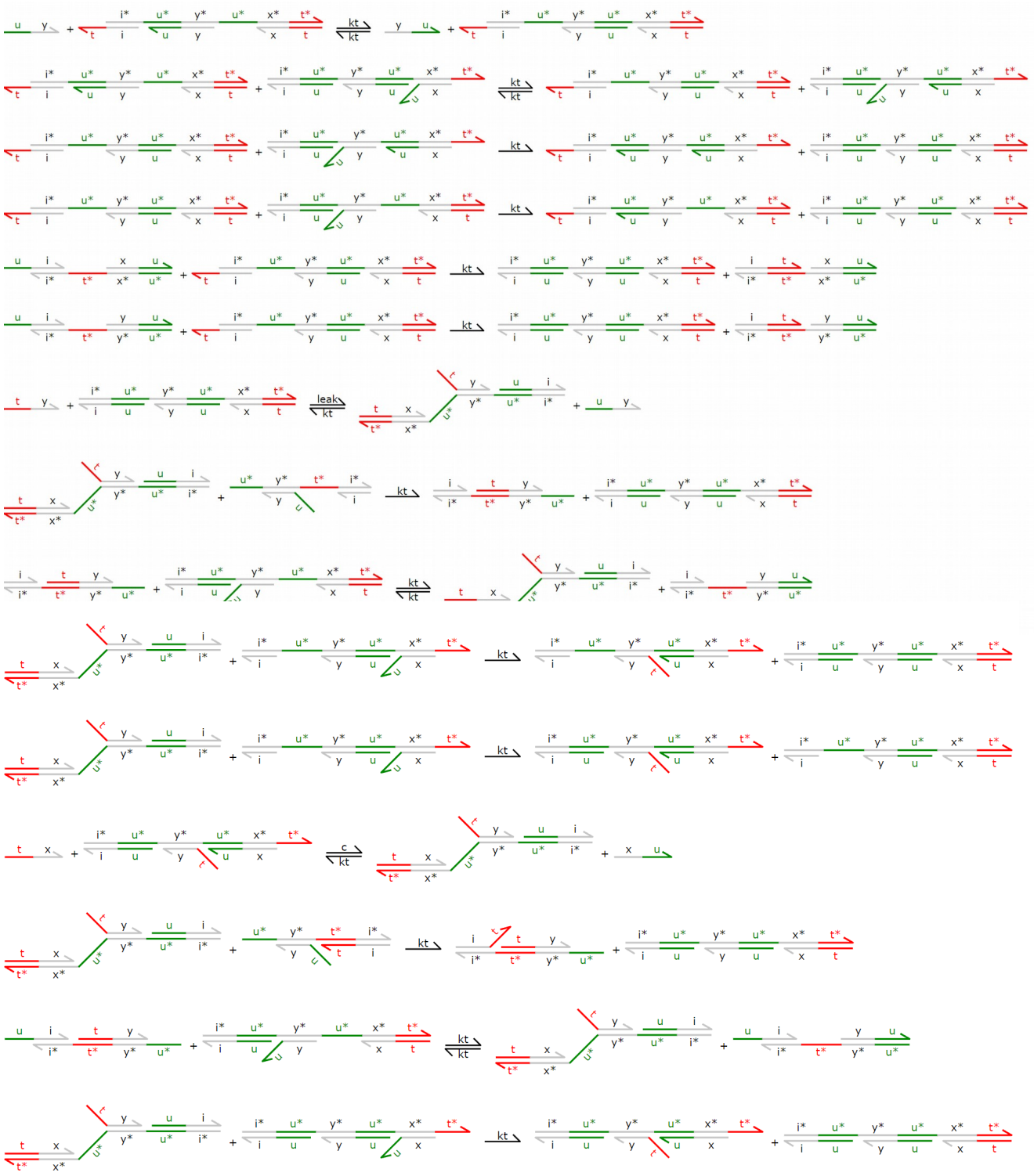
(e) CRN for 2-domain catalysis circuit with first order leaks (continued on next page).



(f) CRN for 2-domain catalysis circuit with first order leaks (continued on next page).



(g) CRN for 2-domain catalysis circuit with first order leaks (continued on next page).



(h)

Figure S32: CRN for 2-domain catalysis circuit with first order leaks.

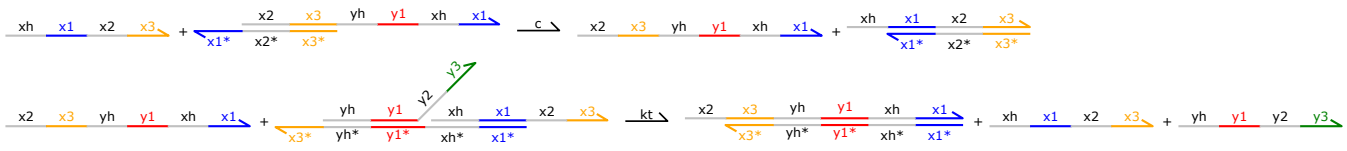


Figure S33: CRN for no-leak 4-domain catalysis circuit.

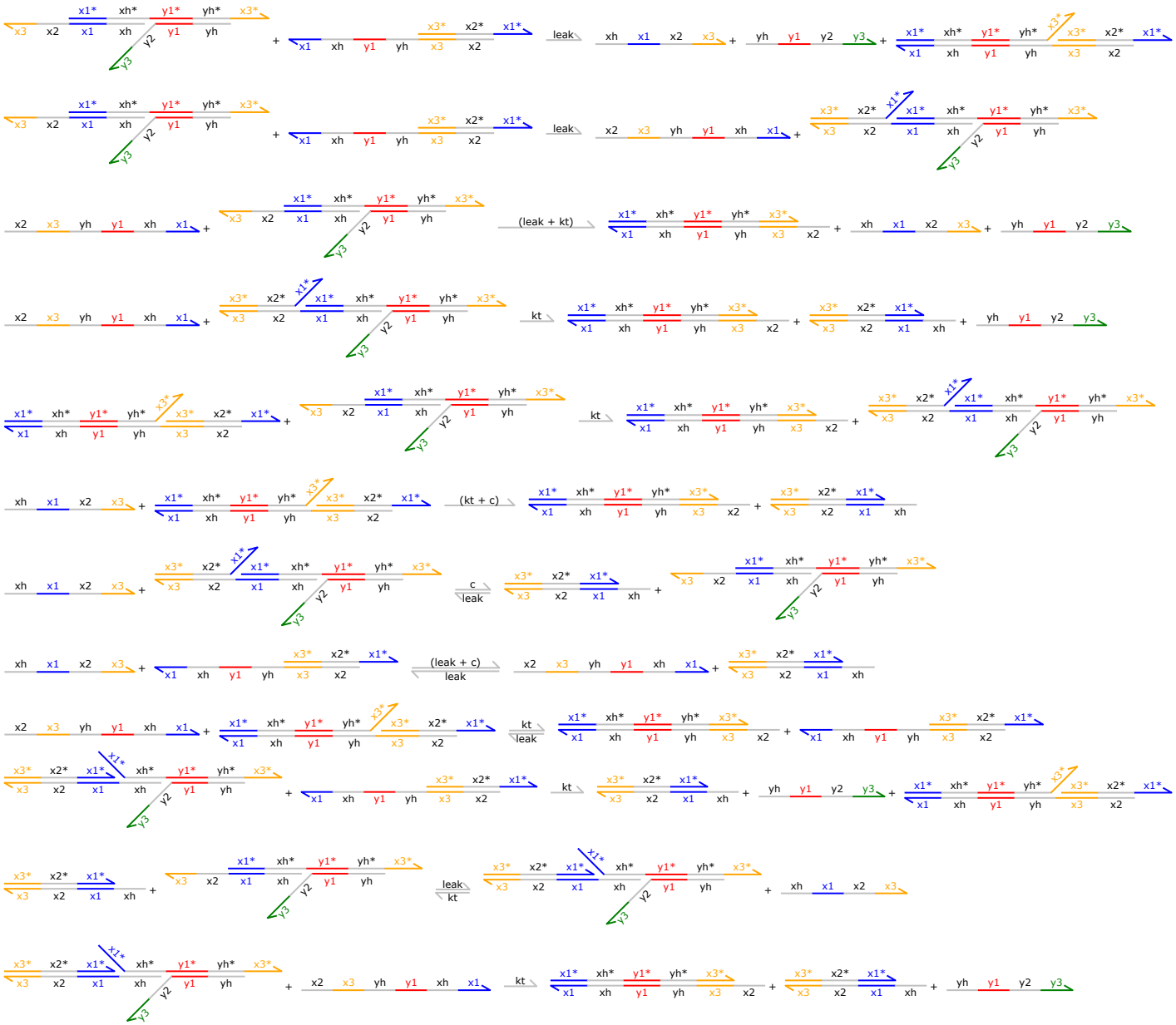


Figure S34: CRN for 4-domain catalysis circuit with first order leaks.

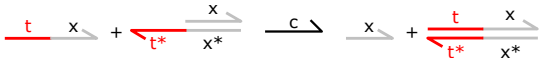


Figure S35: CRN for no-leak 2-domain degradation circuit.

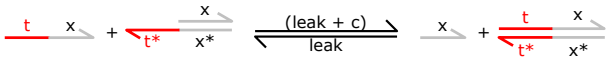


Figure S36: CRN for 2-domain degradation circuit with first order leaks.

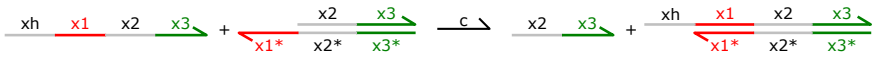


Figure S37: CRN for no-leak 4-domain degradation circuit.

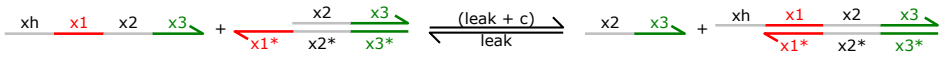


Figure S38: CRN for 4-domain degradation circuit with first order leaks.

References

[1] C. Spaccasassi, M. Lakin, and A. Phillips, "A logic programming language for computational nucleic acid devices," *ACS Synth Biol.*, vol. 8(7), pp. 1530–1547, 2019.

[2] D. Sangiorgi and D. Walker, *PI-Calculus: A Theory of Mobile Processes*. USA: Cambridge University Press, 2001.

[3] K. Oishi and E. Klavins, "Biomolecular implementation of linear I/O systems," *IET Systems Biology*, vol. 5, pp. 252–260, 2011.

[4] B. Yordanov, J. Kim, R. Petersen, A. Shudy, V. Kulkarni, and A. Philips, "Computational design of nucleic acid feedback control circuits," *ACS Synthetic Biology*, vol. 3, pp. 600–616, 2014.

[5] D. Soloveichik, G. Seelig, and E. Winfree, "DNA as a universal substrate for chemical kinetics," *PNAS*, vol. 12, pp. 5393–5398, 2010.

[6] I. Zarubiieva, J. Tseng, and V. Kulkarni, "Efficient ratio computation using abstract chemical reaction networks," in *Transactions on Engineering Technologies: World Congress on Engineering 2018* (S.-I. Ao, L. Gelman, and H.-K. Kim, eds.), ch. 21, pp. C1–C17, New York, New York: Springer Verlag, December 2019. ISBN 978-981-329-531-5.

[7] B. Wang, C. Thachuk, A. D. Ellington, E. Winfree, and D. Soloveichik, "Effective design principles for leakless strand displacement systems," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 115, no. 52, pp. E12182–E12191, 2018.