Exploring DNA Strand-Displacement Computational Elements

Luca Cardelli[1], *Andrew Phillips*[1], *Simon Youssef*[2]

[1]Microsoft Research Cambridge
[2]Ludwig Maximilians Universität Munich

Abstract

Using the Visual DSD language and tool for describing DNA computational structures, we investigate a gate buffering technique that helps maintaining desired reaction kinetics for unbounded time. We test it by simulation on an oscillator and on an ultrasensitive system.

1 Introduction

Visual DSD is an implementation of the programming language for composable DNA circuits described in [1]. The language includes basic elements of sequence domains, toeholds and branch migration, and assumes that strands do not possess any secondary structure. The Visual DSD tool compiles a collection of DNA molecules into a set of chemical reactions. It also includes a stochastic simulator which computes a possible trajectory of the system and graphs the populations of species over time.

In this paper we use Visual DSD to investigate implementations of some basic computational elements ('*gates*') that are sufficient to represent interesting classes of dynamical systems; namely chemical reaction networks [3], stochastic Petri nets, and interacting automata [4]. In particular, we provide *buffered* implementations of those gates that support long-running and even unbounded computations at fixed rates.

The principle of operation of our gates is entropy-driven reactions [11] that proceed autonomously without external intervention (in contrast, for example, to thermal cycling or some form of refueling). These systems must necessarily stop when the entropy is eventually maximized, and the reactions slow down while approaching that limit, implying that the desired kinetic regime is achieved only in the initial stages of system evolution. In turn, this requires using large initial populations so that the desired kinetic regime lasts sufficiently long for the intended purposes. Some of these systems are based on *fuel strands* (components that are turned to waste and provide energy for other reactions), and it is possible to refuel the system periodically by replenishing the fuel strands. However, refueling will normally change the rate of the reactions that use the fuel, resulting in uneven kinetics.

A buffering technique is proposed in [4] to the effect that refueling has no kinetic effect on base reactions and only affects the rate at which gates are replenished. The rate of gate replenishment can be made arbitrarily fast by keep-

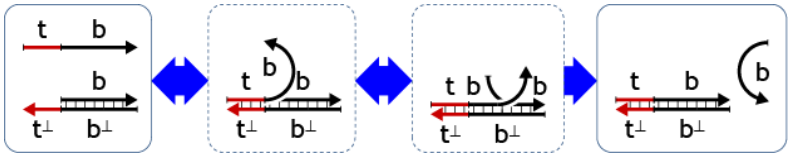

Fig. 1: Strand Displacement

ing the buffers sufficiently high, to compensate for the uneven consumption of gates by the base reactions. This way, one can envision replenishing the buffers periodically and indefinitely without significantly affecting the kinetics of the desired system, and without needing to initialize the system with larger and larger populations to make it run longer.

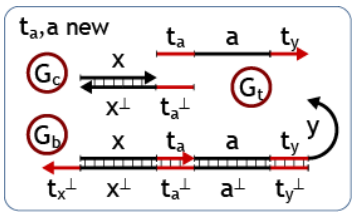

Fig. 2: Transducer

The paper is organized as follows. In Section 2 we give an introduction to Visual DSD and basic gate architecture. In Section 3.1 we describe the buffered gates and in Sections 3.2 and 3.3 we exercise them on two examples: an oscillator and an ultrasensitive system. The former requires a long running computation, and the latter involves an exponential growth, both stressing the supply of buffered gates.

2 DNA Strand Displacement Language

In [1] we introduced a programming language able to describe a class of DNA strand-displacement systems. The systems that can be expressed include general chemical kinetics [3] and automata-like computation [4]. All the systems that can be expressed in this language are composed of a finite number of species and reactions (more generally, DNA strands can combine in an unbounded number of configurations). The class of expressible systems is still a rich one, equivalent to Finite Stochastic Reaction Networks and Petri Nets, and the restriction to a finite set of species means that we can automatically generate all the species that can arise during the evolution of a system. Even for small source programs, the set of species and reactions can easily grow to a large size, and hence it is useful to have a tool to analyze them. The result of this analysis can be depicted in various ways, and can be used as input for stochastic and deterministic simulation.

The design of strand-displacement systems presents a number of logical pitfalls, even beyond all the care that needs to be taken while designing non-interfering DNA codings [7]. These logical pitfalls include unwanted interference between elements of a single construction, unwanted interference between separately designed constructions, and unimagined interactions due to the sheer combinatorial complexity of the systems, sometimes affecting performance rather than functionality. We believe that all these issues will eventually require formal verification, but here we carry out a more empirical analysis of the state space to verify whether certain systems function as expected, and to gain experience in their logical and functional design. In particular, we investigate a number of relatively simple constructions that have been previously proposed [4].

The particular DNA architecture we exploit here is based on two kinds of DNA strand domains, *short* and *long*, each encoded as a sequence of nucleotides. For a fixed set of physical parameters, the short domains are such that they hybridize reversibly to their Watson-Crick complements, while the long domains are such that they hybridize irreversibly. The short domains are used as *toeholds* to initi-

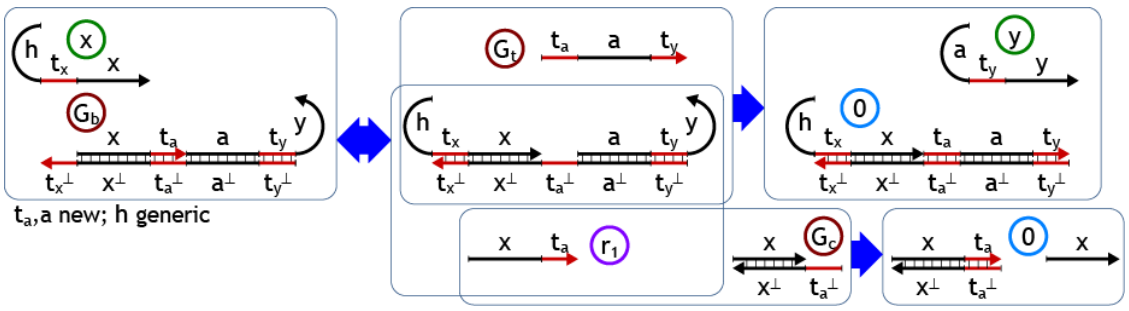

Fig. 3: Transducer Reactions

ate *branch migration* on the long domains, which then leads to *strand displacement*. This process is illustrated in Fig. 1 with the initial reversible binding of a top strand to a double strand by a free toehold. Then, branch migration consists of a branching point between two equal top segments performing a random walk by reversible single-nucleotide swaps. When the random walk reaches the right end, the original top strand is irreversibly displaced. If the two top strands do not match, then the random walk cannot reach the right end and will eventually go back to the left end, where the toehold can detach as if 'nothing' had happened.

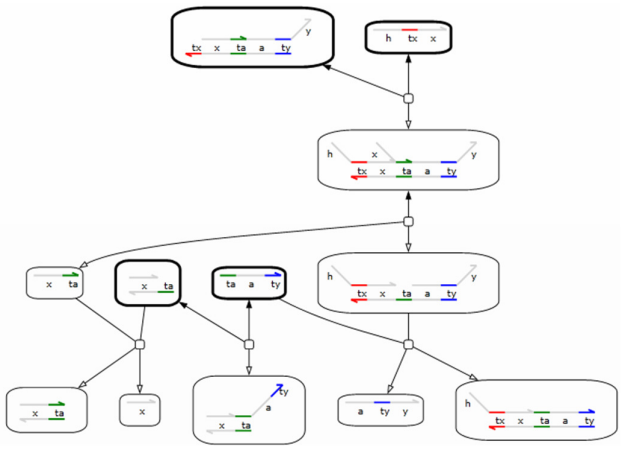

Fig. 4: Transducer Reactions

This basic mechanism of *toehold mediated strand displacement* can be used to design computational elements that process *signals*. An abstract signal 'x' is encoded as a single-stranded DNA sequence consisting of three consecutive domains (see top left of Fig. 3). The first (long) domain is a *history* domain that is produced by past interactions and that is not significant for future interactions; it is indicated as 'h_x' or 'h'. All computational elements are designed so that they are oblivious to the history segments of their inputs, but may produce outputs with specific history segments due to their own mechanics. The second (short) domain is a *toehold* that initiates the interaction between signals and gates, and that starts branch migration on the third domain; the toehold for signal x is denoted by 't_x'. Toeholds are, again, short enough that they bind reversibly: this assumption limits the number of toeholds one can encode, but for simplicity we assume that we have an arbitrary number of distinct ones (it is in fact possible to identify toeholds, but care must then be taken to avoid unwanted interferences). The third (long) domain is the proper *recognition* domain: for a signal x we also denote this domain as 'x', although strictly it is only a part of the signal strand.

Computational elements (*gates*) transform signals into other signals, and are made from double-stranded DNA structures with a free toehold-complement that can bind to an input. The Watson-Crick complement of a segment x is indicated by $x^{\perp}$, with $x^{\perp\perp} = x$. As noted above, if the toehold of a signal binds to a gate, but the signal recognition region does not match, then the branch migration of the recognition segment cannot complete, and the signal will eventually unbind from the gate. These *unproductive interactions* however should be properly modeled as chemical reactions, because they can have a kinetic effect by temporarily sequestering signals and occluding gate toeholds.

Let us start with perhaps the simplest example of a computational element: a transducer 'x.y' that can transform an arbitrary symbolic signal 'x' into another arbitrary symbolic signal 'y'. The transducer is composed of three structures: G_b (gate backbone), G_t (gate trigger), and G_c (gate collector) (Fig. 2). Its intended kinetics is described schematically in Fig. 3, in presence of an input signal x. The annotation 'h generic' means that the transducer works for any history domain of the input signal. The annotations 't_a new' and 'a new' mean that those segment are distinct from the segments of any other signal or gate in the system, to prevent unwanted interferences. The first reaction between the input x and the gate backbone G_b is reversible, but the gate trigger G_t may combine with the result of the x+G_b reaction to make the whole transduction irreversible, leaving an inert residual, '0' (that is, one that does not contain any exposed toeholds), as well as the output y. The ac-

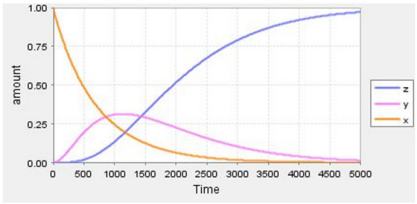

Fig. 5: Transducer Composition

tive residual r_1 is removed by the G_c structure, again leaving inert residuals.

The DNA Strand Displacement Language (DSD [1]) allows us to express the structures of Fig. 2 in a machine readable format, namely as follows:

G_b = tx^:[x ta^]:[a ty^]<y>
G_t = <ta^ a ty^>
G_c = [x]:ta^

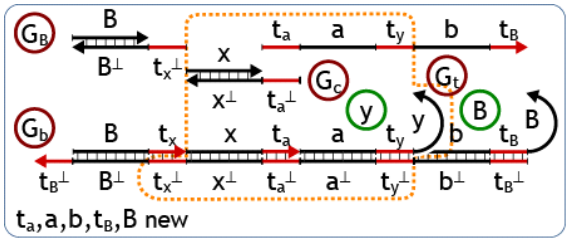

Fig. 6: Buffered Transducer

A suffix '^' is required for a toehold; binding and unbinding rates are separately associated to toeholds (the branch migration rates are instead assumed instantaneous). A *strand* S is a concatenation of abstract domains like x or t^. The syntax <S> represents an upper-strand, and S a bottom strand. The syntax [S⊥] represents instead a double-strand with lower strand S⊥ and upper strand S. The syntax $<S_1>[S_2^\perp]<S_3>$ represents the bottom strand $S_2^\perp$ hybridized to the top strand $S_1S_2S_3$, with the S_1 and S_3 parts 'hanging out' if non-empty. A ':' indicates the concatenation by the bottom strand; for example, $[S_1^\perp]:[S_2^\perp]$ has a contiguous bottom strand $S_1^\perp S_2^\perp$, but an interruption in the top strand between S_1 and S_2. Any expressible structure has a single contiguous bottom strand, but may have interrupts, gaps, or overhangs on the top strand. Finally, we assume that all the hybridization reactions are pre-built into the gate structures, and no hybridization is modeled during execution: only toehold-mediated branch migrations and strand displacements are modeled.

It is convenient to be able to use multiple instances of the transducer structure of Fig. 2 in the same system, but for different inputs and outputs. To this end, the DSD language allows parameterization, and we can thus abstract a reusable *transducer gate* Tr:

```
def Tr(N, tx,x, ty,y) =
  new a
  new ta@bind,unbind
  ( N * tx^:[x ta^]:[a ty^]<y>
  | N * <ta^ a ty^>
  | N * [x]:ta^)
```

Here the line 'def Tr(N, tx,x, ty,y) =' indicates that we are defining a gate named Tr, parameterized by N,tx,x,ty,y, where N is the number of copies of the gate that we wish to produce, x is the input signal with toehold tx, and y is the output signal with toehold ty (more precisely, x and y are the third segments of the input and the output signals). The line 'new a' indicates that a is taken to be distinct from all other segments; in particular, if we instantiate Tr twice we need to generate two distinct DNA sequences for the distinct a's. For a toehold, 'new ta@bind,unbind' includes the binding and unbinding rates. The body of the definition contains the three structures G_b, G_t, G_c defined as above, each multiplied by the number of copies N, and separated by the symbol '|' that generally indicates parallel composition of structures, akin to chemical '+'.

Given the above definition of Tr, and a suitable integer value for a scaling constant S, the following script produces a full executable system consisting of 10*S copies of the transducer gate, and S copies of the x input. We specify a history 'h' for the input, but the system would behave similarly for any other history, including an empty

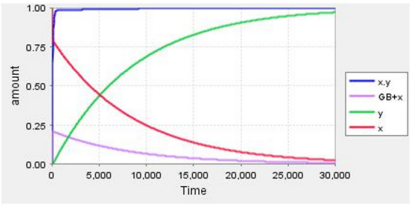

Fig. 7: Buffered Transducer

one.

```
( Tr(10*S, tx,x, ty,y)
| S * <h tx^ x> )
```

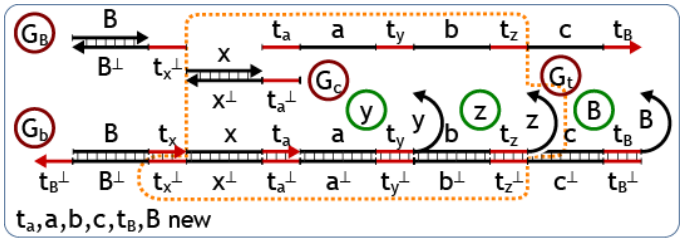

Fig. 8: Buffered Fork

Our Visual DSD tool can analyze such a system and generate the set of all possible species and reactions, resulting in the diagram in Fig. 4, which closely matches Fig. 3: there is a node for each species, with starting species in bold, and arcs for the reactions. However, the analysis reveals a 'forgotten' reversible reaction between G_t and G_c, resulting in an extra species [x]:[ta^]<a ty^>. The tool can be set to produce or ignore these *unproductive* reactions that do not lead to successful strand displacements and can be very numerous. In an actual chemical system, all these would of course occur and have some kinetic effect.

As an example of use of a parametric definition, consider the composition of two transducers: Tr(N, tx,x, ty,y) | Tr(N, ty,y, tz,z), resulting in a transducer from x to z via y:

```
( Tr(10*S, tx,x, ty,y)
| Tr(10*S, ty,y, tz,z)
| S * <h tx^ x> )
```

If we were to expand the definition of Tr twice, the structures generated would be equivalent to the following (for any new ta_1,a_1,ta_2,a_2: the tool automatically generates separate strands like a.1 and a.2):

```
( 10*S * tx^:[x ta1^]:[a1 ty^]<y>
| 10*S * <ta1^ a1 ty^>
| 10*S * [x]:ta1^
| 10*S * ty^:[y ta2^]:[a2 tz^]<z>
| 10*S * <ta2^ a2 tz^>
| 10*S * [y]:ta2^
| S * <h tx^ x> )
```

This system results in 24 species and 11 reactions. We extract the system of chemical reactions as an SBML file and simulate it in CellDesigner [6] using S = 1, resulting in the plot of Fig. 5. The reaction rates used through the paper fall conservatively within experimental range [7]: toehold binding at 3×10^5/M/s and toehold unbinding at 0.1126/s. The vertical axis is in nM (with 10nM, not shown, of gate species) and the horizontal axis is in seconds.

3 Gate Architecture

In this section we investigate a gate architecture that supports long-term system evolution by replenishing gates from

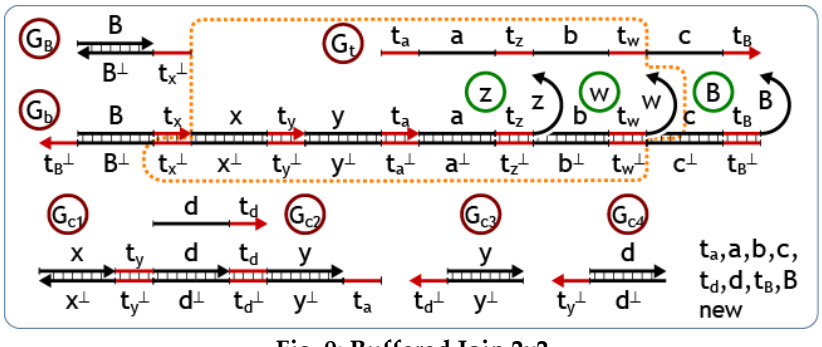

Fig. 9: Buffered Join 2x2

buffer pools. We then show two examples of systems built out of such gates: an oscillator and an ultrasensitive switch.

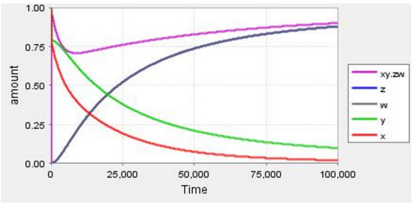

Fig. 10: Buffered Join 2x2

3.1 Buffered Gates

Gate structures are consumed during signal processing: the energy driving the reactions is in fact provided by the gate structures being turned into waste structures. Hence the gate population is not fixed, and the kinetics of signal processing changes over time. One could use a very large and hence almost-constant concentration of gates with respect to the concentration of signals, but this puts limits on the concentration of signals. Moreover, the instantaneous concentration of signals is dependent on the dynamics of the reactions, and it may be difficult in general to keep it at a relatively low level. Finally, if there is a very large amount of free toeholds (given by a very large populations of active gates) then the signals may too often bind reversibly to the 'wrong' gates, impeding progress. The optimal situation would be to keep the concentration of active gates at a constant level that is close to the concentration of the signals.

To alleviate these problems, an automatic buffering technique is proposed abstractly in [4]: here we flesh it out with concrete strand structures. The idea is to keep a quasi-constant but relatively low concentration of gate structures by means of a higher concentration of buffer structures that are turned into gates on demand. The buffer levels do not significantly affect the kinetics of the reactions (at least until they run out), and they could be replenished periodically without significantly affecting the ongoing kinetics of the gates. The effective rates of the signal processing reactions remain then almost constant, provided the gates are replenished fast enough from the buffers.

A buffered gate is a *curried gate*; that is, a gate that produces another gate after an input. In Fig. 6, an input signal <_ t_B B> (where _ represents any segment) binding to the G_b structure ejects a <B t_x^> segment that is neutralized by the G_B structure. The residual G_b structure, in dotted outline, then works like a normal transducer from x to y together with G_t and G_c. But in addition to the y output, this gate also outputs another <_ t_B B> signal. Hence, every time one transducer gate is consumed, it triggers the release of one similar gate from a pool of G_b structures. (We are assuming that B is a signal used only for buffering purposes; otherwise the G_B structure might interfere with other gates, and in particular with the Join gate shown later. Here B is private to the buffer implementation, and the 'new B' declaration is sufficient to prevent any interference: a slightly different solution is available for a general curried gate where B is an arbitrary public signal [4].)

The buffered transducer can be abstracted in the following parametric definition, for arbitrary x and y signals, for a number N of gates to be kept constant until the buffer runs out, and for a buffer of initial size M. The five lines in the body of the script correspond to G_b, G_t, G_c, G_B, and to a set of N buffer signals, <b tB^ B>, which cause the release of the initial N transducers from the buffer pool.

```
def BTr(M,N, tx,x, ty,y) =
new ta@bind,unbind new a new b new
tB@bind,unbind new B
( M * tB^:[B tx^]:[x ta^]:[a ty^]<y>:[b tB^]<B>
| M * <ta^ a ty^ b tB^>
| M * [x]:ta^
| M * [B]:tx^
| N * <b tB^ B> )
```

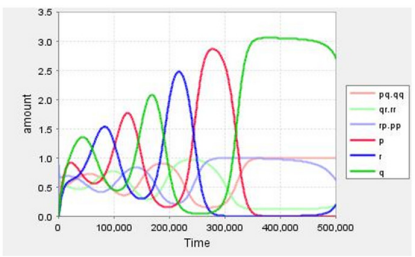

Fig. 11: Oscillator

The definition of BTr can be invoked on a set of x signals with a scaling factor S (here set to 1), and a buffering factor Buff (set to 100×S).

```
( BTr(Buff,S, tx,x, ty,y)
| S * <h tx^ x> )
```

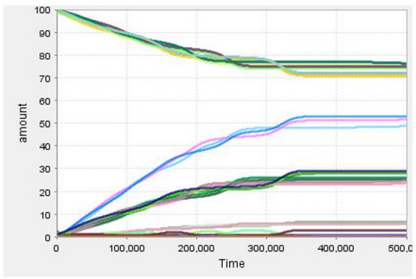

Fig. 12: Oscillator Buffers

Fig. 7 shows the time evolution (21 species and 10 reactions): the initial x signal rapidly finds a reversible equilibrium with to the more numerous G_B buffer structures, and therefore is depressed to ~0.77. The y signal is then produced over time by consuming x; the x.y curve is the freely available transducer gates being generated from the buffer.

A simple extension of the transducer design yields a buffered Fork gate with one input x and two outputs y,z (Fig. 8), which we use in a later example. It is defined as follows:

```
def BF2(M,N, tx,x, ty,y, tz,z) =
new ta@bind,unbind new a new b new c new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ta^]:[a ty^]<y>:[b tz^]<z>:[c tB^]<B>
| M * <ta^ a ty^ b tz^ c tB^>
| M * [x]:ta^
| M * [B]:tx^
| N * <c tB^ B> )
```

We now develop a more challenging gate with 2 inputs x,y and 2 outputs to z,w, in addition to a first curried input B and a third output B for its buffer. The construction in Fig. 9 is based on the Join gate design from [4], including a garbage collecting component (the G_{c1}..G_{c4} structures) that removes the active residuals of G_b; namely, <x t_y^> and <y t_a^>. In G_{c1} the [d t_d] segment has the purpose of separating [t_y] from [y]: without it, a segment <t_y y> would be released during garbage collection that would function as a y signal. Instead, a segment <t_d y> is released (and then collected by G_{c3}), and it is essential that $t_d \neq t_y$. Similarly, <t_y d> is released (and then collected by G_{c4}) and it is essential that d≠y. Hence, t_d and d are chosen 'new'. Moreover, the segment <x t_y> from G_b is absorbed by G_{c1}, and it is essential that no other free segment in the whole system has toehold t_y and history x, because it would be absorbed here. Because of this, we make sure that all gates (including BTr) always output signals with a 'new' history, which is therefore different from a signal history like x. Simpler transducer designs exist that do not have this property [4], and that therefore cause conflicts. Finally, the G_b and G_{c1} structures are chained via the t_a toehold, which must be unique to each join gate; this is again guaranteed by 'new'.

Fig. 10 shows a trace of this gate with x and y inputs and including all unproductive reactions, showing an almost stable active gate population (xy.zw) which is initially depressed under heavy load, but that recovers from the buffer. The outputs z and w are produced over time (their curves overlap). The script for this system is:

```
def BJ2x2(M,N, tx,x, ty,y, tz,z, tw,w) =
new ta@bind,unbind new a new b new c
new td@bind,unbind new d new tB@bind,unbind
new B
( M * tB^:[B tx^]:[x ty^]:[y ta^]:[a tz^]<z>:[b
tw^]<w>:[c tB^]<B>
```

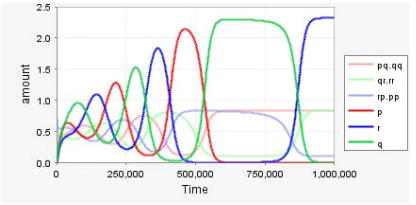

Fig. 13: Oscillator

```
| M * <ta^ a tz^ b tw^ c tB^>
| M * [B]:tx^
| M * [x]:[ty^ d]:[td^ y]:ta^
| M * <d td^>
| M * td^:[y]
| M * ty^:[d]
| N * <c tB^ B> )

( BJ2x2(Buff,S, tx,x, ty,y, tz,z, tw,w)
| S * <hx tx^ x>
| S * <hy ty^ y> )
```

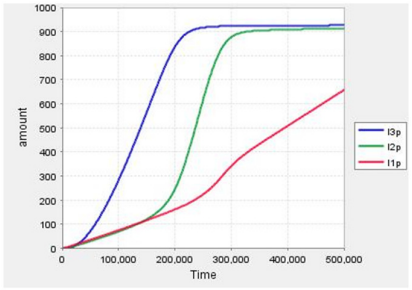

Fig. 14: Ultrasensitivity

3.2 Oscillating System

We compose three buffered join gates into a circuit to obtain a three-way oscillator. Given three signals p,q,r, the gates perform the reactions p+q→q+q, q+r→r+r, r+p→p+p [8]. This is an intrinsically unstable oscillator, but a very simple one to experiment with, requiring only 3 join gates. To build this circuit, we just reuse the previous definition of the buffered join gate, B2x2, and provide an initial sets of signals to start an oscillation:

```
( BJ2x2(Buff,S, tp,p, tq,q, tq,q, tq,q)
| BJ2x2(Buff,S, tq,q, tr,r, tr,r, tr,r)
| BJ2x2(Buff,S, tr,r, tp,p, tp,p, tp,p)
| 3*S * <hp tp^ p>
| 2*S * <hq tq^ q>
| 2*S * <hr tr^ r> )
```

The system without unproductive reactions consists of 186 species and 129 reactions; we show its execution in Fig. 11. The higher amplitude oscillations are three of the signal species (there are two such curves for each signal, because of different history segments). The lower amplitude oscillations are three of the gate structures, which start at 1.0 but are then cyclically depleted by interactions with the signals and replenished from the buffers (here we are intentionally stressing the system: bigger buffers would result in a more even level of active gates). Fig. 12 shows all the species in the system, highlighting the depletion of the buffers and the accumulation of waste products, with the oscillations of Fig. 11 near the bottom. The system with all unproductive reactions consists of 303 species and 246 reactions (Fig. 13), exhibiting a lower amplitude and a slower oscillation.

3.3 Ultrasensitive System

A phosphate-transfer system [9] consists of a set of layers where the first layer detects a signal and the last layer enacts a response. The layers are connected by a cascade of phosphate transfers; the last layer typically has a linear response up to saturation, while the middle layers have an ultrasensitive ('switching') response. We look at a 3-layer system, which initially

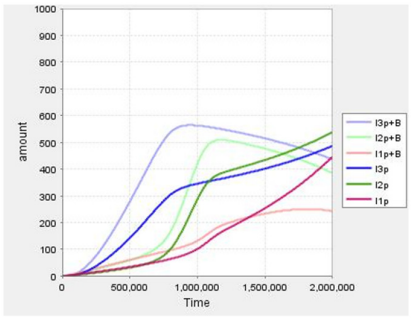

Fig. 15: Ultrasensitivity

contains three reservoirs for species l1, l2, l3, and has reactions:

(1) $l1 \rightarrow l1p$
(2) $l1p + l2 \rightarrow l1 + l2p$
(3) $l2p + l3 \rightarrow l3p + l2$
(4) $l3p \rightarrow l3$

Reaction (1) represents (implicitly) the first layer species l1 being phosphorylated by an external stimulus. Reaction (2) represents the transfer of the phosphate from the first to the second layer, and similarly reaction (3) from the second to the third layer. Reaction (4) represents (implicitly) the work performed by the third layer, transferring the phosphate to some other external system and being reset to l3. We study the response of this system by varying the amount of l1 and looking at the corresponding steady-state amounts of l1p, l2p, l3p. The initial amount of l1 here represents the strength of a sustained stimulation, since this determines the amount of l1p being generated. The initial amount of l3 represents the maximum response. Under certain ranges of reaction rates, the second layer exhibits an ultrasensitive response [10].

Our corresponding strand displacement system consists of four gates for the four reactions: two transducers and two 2x2 joins. The rate of each reaction is controlled by setting the (buffered) amount of each gate species. We use equal amounts (10*S) for reactions (1), (2) and (3), and a smaller amount (1*S) for reaction (4), which needs to be slower to allow the accumulation of l3p but non-zero to obtain a proper ultrasensitive response. To study the steady-state response to varying amounts of l1, we then add a fifth gate (also in quantity 10*S) that slowly and linearly increases the amount of l1 starting from zero: the increase is slow enough to keep the system near steady state. The horizontal axis of the simulation therefore does not represent the time of the reaction: it represents the amount of l1+l1p in the system, which is growing linearly in time. The fifth gate implements the reaction:

(5) $stim \rightarrow stim + l1$

adding l1 at a constant rate.

The initial conditions of the system include reservoirs for l2, l3 (each 1000*S) and stim (10*S). The initial level of l3 determines the point at which the ultrasensitive switch happens, which is the point at which l1+l1p exceeds l3+l3p. The initial level of l2 determines the maximum strength of the l2p response: this level is set equal to l3 here, but could be set much higher than l3 to obtain an ultrasensitive amplification that could be used as an input for some other system, including another ultrasensitive amplifier. Chaining ultrasensitive amplifiers can produce an extremely sharp switch of a high-quantity species from a weak stimulus. The initial level of stim determines the speed at which l1 is added to the system, and again this should be slow enough when studying the steady-state response.

The system being simulated is therefore the following, where we use Buff = 10000 and S = 1.

```
(  BF2(Buff,10*S, tstim,stim, tl1,l1, tstim,stim)          (* reaction (5) *)
|  BTr(Buff,10*S, tl1,l1, tl1p,l1p)                        (* reaction (1) *)
|  BJ2x2(Buff,10*S, tl1p,l1p, tl2,l2, tl1,l1, tl2p,l2p)    (* reaction (2) *)
|  BJ2x2(Buff,10*S, tl2p,l2p, tl3,l3, tl2,l2, tl3p,l3p)    (* reaction (3) *)
|  BTr(Buff,1*S, tl3p,l3p, tl3,l3)                         (* reaction (4) *)
|  1000*S * <hl2 tl2^ l2>                                 (* initial l2 *)
|  1000*S * <hl3 tl3^ l3>                                 (* initial l3 *)
|  10*S * <hstim tstim^ stim>)                            (* initial stim *)
```

Fig. 14 shows that this system produces a typical ultrasensitive response in l2p, and a linear response in l3p (with a slight initial delay due to lag in the gates); this simulation does not include nonproductive reactions. Fig. 15 shows the same system but with all the nonproductive reactions: the behavior is in fact very similar, except that it is stretched on the horizontal axis (at least in part because the stimulus gets diluted by nonproductive reactions), and that each of l1p, l2p, l3p is in equilibrium with a separate buffer species, so the effective total of each available signal is the sum of two curves of similar color.

4 Conclusions

We have shown that automatic tools for compiling higher-level languages to (large sets of) chemical reactions can be useful for investigating DNA gate designs. Even when the set of reactions is finite, it can grow combinatorially with the size of the system, and it is unfeasible to generate it by hand. This is particularly important when including *unproductive* reversible reactions, which have a sometimes mild but usually noticeable effect on the kinetics, and other kinds of reactions like leaks and secondary structure interactions that we have not included in this work.

Simple finite sets of components can interact so intricately that it is not feasible to generate the full set of reactions; other systems (those that can produce polymers) have an infinite set of reactions. Still, all those systems can be described in high-level languages, because a language can represent a huge or even infinite state space (a set of species and reactions) finitely and compactly. Even when the state space cannot be precomputed, it is possible to inspect it by generating the required states incrementally from the high-level description [2]. All this points to a useful role for high-level languages and tools for investigating DNA systems, which are by nature highly combinatorial.

References

[1] A. Phillips, L. Cardelli. A Programming Language for Composable DNA Circuits. Journal of the Royal Society Interface, August 6 2009, 6:S419-S436.

[2] A. Phillips, L. Cardelli. Efficient, Correct Simulation of Biological Processes in the Stochastic Pi-calculus. In Computational Methods in Systems Biology, LNBI 4695, pp 184-199, Springer, 2007.

[3] D. Soloveichik, G. Seelig, E. Winfree. DNA as a Universal Substrate for Chemical Kinetics. In Proc. DNA Computing and Molecular Programming, 14th International Conference, 2008.

[4] L. Cardelli. Strand Algebras for DNA Computing. In DNA Computing and Molecular Programming, Revised Selected Papers. LNCS 5877, pp 12-24, Springer, 2009.

[5] A. Marathe, A.E. Condon, R.M. Corn. On Combinatorial DNA Word Design. J. Comp. Biology 8(3) 201–219, 2001.

[6] CellDesigner: A modeling tool of biochemical networks. http://www.celldesigner.org.

[7] Winfree Lab: personal communication.

[8] L. Cardelli: Artificial Biochemistry. In: A. Condon, D. Harel, J.N. Kok, A. Salomaa, E.Winfree (Eds.). Algorithmic Bioprocesses. Springer, 2009.

[9] P. Thomason, R. Kay. Eukaryotic signal transduction via histidine-aspartate phosphorelay. *J Cell Sci* 113 (Pt 18), 3141-3150, 2000.

[10] C. Gomez-Uribe, G.C. Verghese, L.A. Mirny. Operating Regimes of Signaling Cycles: Statics, Dynamics, and Noise Filtering. PLoS Computational Biology, 3(12) 2007.

[11] D.Y. Zhang, A.J. Turberfield, B. Yurke, E. Winfree. Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA. Science Vol. 318. no. 5853, pp 1121 – 1125, 2007.

5 Auxiliary Material

These are the Visuals DSD scripts used to generate the figures: they can be used to produce stochastic simulations within the Visual DSD tool. For the figures, however, the SBML output of the Visuals DSD scripts is passed to Cell Designer for ODE simulation and plotting: the correspondence between ODE abbreviated legends and Visual DSD structures is noted below. The scaling factor S is useful for stochastic simulations: it allows one to change the (integer) number of molecules while automatically compensating the rate of the binary reactions, therefore preserving the kinetics. The scripts below include adequate values of S for stochastic simulations, but in the ODE simulations, where we can use fractional concentration values, we just use S=1. The binary toehold binding rate is in /nM/s, and hence with S=1 the vertical axis is in nM and the horizontal axis in seconds. In the ultrasensitive system we need a wider range of concentrations, and to keep the highest concentration within reasonable range we convert the toehold binding rate to /0.1 nM/s: the vertical axis is then in 0.1 nM, with the buffer species at 1 µM.

5.1 Transducer

```
directive sample 5000.0 1000
directive plot <_ tx^ x>; <_ ty^ y>
def S = 1000
def bind = 0.0003/(float_of_int S) (* /nM/s *) (* =3*10^5 /M/s *)
def unbind =  0.1126 (* /s *)
new tx@bind,unbind
new ty@bind,unbind

def Tr(N, tx,x, ty,y) =
  new a new ta@bind,unbind
  ( N * tx^:[x ta^]:[a ty^]<y>
  | N * <ta^ a ty^>
  | N * [x]:ta^
)

( Tr(10*S, tx,x, ty,y)
| S * <h tx^ x>
)
```

5.2 Transducer Composition

```
directive sample 5000.0 1000
directive plot <_ tx^ x>; <_ ty^ y>; <_ tz^ z>
def S = 1000
def bind = 0.0003/(float_of_int S) (* /nM/s *) (* =3*10^5 /M/s *)
def unbind =  0.1126 (* /s *)
new tx@bind,unbind
new ty@bind,unbind
new tz@bind,unbind

def Tr(N, tx,x, ty,y) =
  new a new ta@bind,unbind
```

```
  ( N * tx^:[x ta^]:[a ty^]<y>
  | N * <ta^ a ty^>
  | N * [x]:ta^
)

( Tr(10*S, tx,x, ty,y)
| Tr(10*S, ty,y, tz,z)
| S * <h tx^ x>
)
```

Figure 5: Cell Designer simulation of Visual DSD SBML output with S=1. Legend: x = <h tx^ x>, y = <a ty^ y>, z = <a.1 tz^ z>.

5.3 Buffered Transducer

```
directive sample 30000.0 1000
directive plot sum(<_ tx^ x>); [B]:<h>[tx^]<x>; sum(<_ ty^ y>); <b>[tB^ B]:tx^:[x ta^]:[a ty^]<y>:[b tB^]<B>
def S = 100
def Buff = 100*S
def bind = 0.0003/(float_of_int S) (* /nM/s *) (* =3*10^5 /M/s *)
def unbind =  0.1126 (* /s *)
new tx@bind,unbind
new ty@bind,unbind

def BTr(M,N, tx,x, ty,y) =
new ta@bind,unbind new a new b new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ta^]:[a ty^]<y>:[b tB^]<B>
| M * <ta^ a ty^ b tB^>
| M * [x]:ta^
| M * [B]:tx^
| N * <b tB^ B>
)

(  BTr(Buff,S, tx,x, ty,y)
|  S * <h tx^ x>
)
```

Figure 7: Cell Designer simulation of Visual DSD SBML output with S=1. Legend: x = <h tx^ x>, y = <a ty^ y>, x.y = <b>[tB^ B]:tx^:[x ta^]:[a ty^]<y>:[b tB^]<B>, GB+x = [B]:<h>[tx^]<x>.

5.4 Buffered Fork

```
directive sample 30000.0 1000
directive plot sum(<_ tx^ x>); [B]:<h>[tx^]<x>; sum(<_ ty^ y>); sum(<_ tz^ z>); <c>[tB^ B]:tx^:[x
ta^]:[a ty]<y>:[b tz^]<z>:[c tB^]<B>
def S = 100
def Buff = 100*S
def bind = 0.0003/(float_of_int S) (* /nM/s *) (* =3*10^5 /M/s *)
def unbind =  0.1126 (* /s *)
new tx@bind,unbind
new ty@bind,unbind
new tz@bind,unbind

def BF2(M,N, tx,x, ty,y, tz,z) =
```

```
new ta@bind,unbind new a new b new c new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ta^]:[a ty^]<y>:[b tz^]<z>:[c tB^]<B>
| M * <ta^ a ty^ b tz^ c tB^>
| M * [x]:ta^
| M * [B]:tx^
| N * <c tB^ B>
)

(  BF2(Buff,S, tx,x, ty,y, tz,z)
|  S * <h tx^ x>
)
```

5.5 Buffered Join

```
directive sample 100000.0 1000
directive plot sum(<_ tx^ x>); sum(<_ ty^ y>); sum(<_ tz^ z>); sum(<_ tw^ w>); <c tB^ B>; <c>[tB^
B]:tx^:[x ty^]:[y ta^]:[a tz^]<z>:[b tw^]<w>:[c tB^]<B>
def S = 100
def Buff = 100*S
def bind = 0.0003/(float_of_int S) (* /nM/s *) (* =3*10^5 /M/s *)
def unbind =  0.1126 (* /s *)
new tx@bind,unbind
new ty@bind,unbind
new tz@bind,unbind
new tw@bind,unbind

def BJ2x2(M,N, tx,x, ty,y, tz,z, tw,w) =
new ta@bind,unbind new a new b new c
new td@bind,unbind new d new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ty^]:[y ta^]:[a tz^]<z>:[b tw^]<w>:[c tB^]<B>
| M * <ta^ a tz^ b tw^ c tB^>
| M * [B]:tx^
| M * [x]:[ty^ d]:[td^ y]:ta^
| M * <d td^>
| M * td^:[y]
| M * ty^:[d]
| N * <c tB^ B>
)

(  BJ2x2(Buff,S, tx,x, ty,y, tz,z, tw,w)
|  S * <hx tx^ x>
|  S * <hy ty^ y>
)
```

Figure 10: Cell Designer simulation of Visual DSD SBML output with S=1. Legend:
x = <h tx^ x>, y = <h ty^ y>, z = <a tz^ z>, w = <b tw^ w>,
xw.zw = <c>[tB^ B]:tx^:[x ty^]:[y ta^]:[a tz^]<z>:[b tw^]<w>:[c tB^]<B>.

5.6 Oscillating System

```
directive sample 500000.0 1000
```

```
directive plot sum(<_ tp^ p>); sum(<_ tq^ q>); sum(<_ tr^ r>);
sum(<c tB^ B>); <c>[tB^ B]:tp^:[p tq^]:[q ta^]:[a tq^]<q>:[b tq^]<q>:[c tB^]<B>
def S = 100
def Buff = 100*S
def bind = 0.0003/(float_of_int S) (* /nM/s *) (* =3*10^5 /M/s *)
def unbind =  0.1126 (* /s *)
new tp@bind,unbind
new tq@bind,unbind
new tr@bind,unbind

def BJ2x2(M,N, tx,x, ty,y, tz,z, tw,w) =
new ta@bind,unbind new a new b new c
new td@bind,unbind new d new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ty^]:[y ta^]:[a tz^]<z>:[b tw^]<w>:[c tB^]<B>
| M * <ta^ a tz^ b tw^ c tB^>
| M * [B]:tx^
| M * [x]:[ty^ d]:[td^ y]:ta^
| M * <d td^>
| M * td^:[y]
| M * ty^:[d]
| N * <c tB^ B> )

(  BJ2x2(Buff,S, tp,p, tq,q, tq,q, tq,q)
|  BJ2x2(Buff,S, tq,q, tr,r, tr,r, tr,r)
|  BJ2x2(Buff,S, tr,r, tp,p, tp,p, tp,p)
|  3*S * <hp tp^ p>
|  2*S * <hq tq^ q>
|  2*S * <hr tr^ r>
)
```

Figures 11,12,13: Cell Designer simulations of Visual DSD SBML output with S=1, and with 'unproductive' flag set for Figure 13. Legend for Figures 11,13:
p = <a.10 tp^ p>, q = <a tq^ q>, r = <a.2 tr^ r>, pq.qq = <c>[tB^ B]:tp^:[p tq^]:[q ta^]:[a tq^]<q>:[b tq^]<q>:[c tB^]<B>, qr.rr = <c.4>[tB.7^ B.8]:tq^:[q tr^]:[r ta.1^]:[a.2 tr^]<r>:[b.3 tr^]<r>:[c.4 tB.7^]<B.8>, rp.pp = <c.12>[tB.15^ B.16]:tr^:[r tp^]:[p ta.9^]:[a.10 tp^]<p>:[b.11 tp^]<p>:[c.12 tB.15^]<B.16>.

5.7 Ultrasensitive System

```
directive sample 500000.0 1000
directive plot sum(<_ tl1p^ l1p>); sum(<_ tl2p^ l2p>); sum(<_ tl3p^ l3p>)
def S = 1
def Buff = 10000*S
def bind = 0.00003/(float_of_int S) (* /.1 nM/s *) (* =3*10^5 /M/s *)
def unbind =  0.1126 (* /s *)
new tstim@bind,unbind
new tl1@bind,unbind  new tl1p@bind,unbind
new tl2@bind,unbind  new tl2p@bind,unbind
new tl3@bind,unbind  new tl3p@bind,unbind

def BTr(M,N, tx,x, ty,y) =
new ta@bind,unbind new a new b new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ta^]:[a ty^]<y>:[b tB^]<B>
```

```
| M * <ta^ a ty^ b tB^>
| M * [x]:ta^
| M * [B]:tx^
| N * <b tB^ B>
)

def BF2(M,N, tx,x, ty,y, tz,z) =
new ta@bind,unbind new a new b new c new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ta^]:[a ty^]<y>:[b tz^]<z>:[c tB^]<B>
| M * <ta^ a ty^ b tz^ c tB^>
| M * [x]:ta^
| M * [B]:tx^
| N * <c tB^ B>
)

def BJ2x2(M,N, tx,x, ty,y, tz,z, tw,w) =
new ta@bind,unbind new a new b new c
new td@bind,unbind new d new tB@bind,unbind new B
( M * tB^:[B tx^]:[x ty^]:[y ta^]:[a tz^]<z>:[b tw^]<w>:[c tB^]<B>
| M * <ta^ a tz^ b tw^ c tB^>
| M * [B]:tx^
| M * [x]:[ty^ d]:[td^ y]:ta^
| M * <d td^>
| M * td^:[y]
| M * ty^:[d]
| N * <c tB^ B>
)

(  BF2(Buff,10*S, tstim,stim, tl1,l1, tstim,stim)
|  BTr(Buff,10*S, tl1,l1, tl1p,l1p)
|  BJ2x2(Buff,10*S, tl1p,l1p, tl2,l2, tl1,l1, tl2p,l2p)
|  BJ2x2(Buff,10*S, tl2p,l2p, tl3,l3, tl2,l2, tl3p,l3p)
|  BTr(Buff,1*S, tl3p,l3p, tl3,l3)
|  1000*S * <hl2 tl2^ l2>
|  1000*S * <hl3 tl3^ l3>
|  10*S * <hstim tstim^ stim>
)
```

Figure 14: Cell Designer simulation of Visual DSD SBML output. Legend:
l1p = <a.2 tl1p^ l1p>, l2p = <b.8 tl2p^ l2p>, l3p = <b.14 tl3p^ l3p>.

Figure 15: Cell Designer simulation of Visual DSD SBML output with 'unproductive' flag set. Legend:
l1p = <a.2 tl1p^ l1p>, l2p = <b.8 tl2p^ l2p>, l3p = <b.14 tl3p^ l3p>,
l1p+B = [B.11]:<a.2>[tl1p^]<l1p>, l2p+B = [B.19]:<b.8>[tl2p^]<l2p>, l3p+B = [B.24]:<b.14>[tl3p^]<l3p>.