Two-Domain DNA Strand Displacement

Luca Cardelli

Microsoft Research

Abstract

We investigate the computing power of a restricted class of DNA strand displacement structures: those that are made of double strands with nicks (interruptions) in the top strand. To preserve this structural invariant, we impose restrictions on the single strands they interact with: we consider only two-domain single strands consisting of one toehold domain and one recognition domain. We study fork and join signal processing gates based on these structures, and we show that these systems are amenable to formalization and to mechanical verification.

Keywords: DNA Computing, Process Algebra.

1 Introduction

Among the many techniques being developed for molecular computing [5], *DNA strand displacement* has been proposed as mechanism for performing computation with DNA strands [8, 3]. In most schemes, single-stranded DNA acts as *signals* and double-stranded (or more complex) DNA structures act as *gates*. Various circuits have been demonstrated experimentally [10]. The strand displacement mechanism is appealing because it is *autonomous* [4]: once signals and gates are mixed together, computation proceeds on its own without further intervention until the gates or signals are depleted (output is often read by fluorescence). The energy for computation is provided by the gate structures themselves, which are turned into inactive waste in the process. Moreover, the mechanism requires *only* DNA molecules: no organic sources, enzymes, or transcription/translation ingredients are required, and the whole apparatus can be chemically synthesized and run in basic wet labs.

The main aims of this approach are to harness computational mechanisms that can operate at the molecular level and produce nano-scale structures under program control, and somewhat separately that can intrinsically interface to biological entities [2]. The computational structures that one may easily implement this way (without some form of unbounded storage) vary from Boolean networks, to state machines, to Petri nets. The last two are particularly interesting because they take advantage of DNA's ability to encode symbolic information: they operate on DNA strands that represent abstract signals.

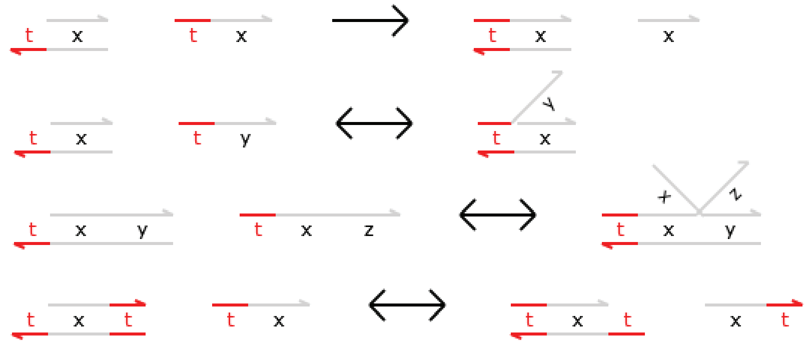


Fig. 1: Toehold-mediated DNA branch migration and strand displacement

The fundamental mechanism in many of these schemes is *toehold mediated branch migration and strand displacement* [10], which implements a basic step of computation. It operates as shown in Figure 1, where each letter and corresponding segment represents a DNA *domain* (a sequence of nucleotides, C,G,T,A) and each DNA strand is seen as the concatenation of multiple domains. Single strands have an orientation; double strands are composed of two single strands with opposite orientation, where the bottom strand is the Watson-Crick, $C - G$, $T - A$, complement of the top strand. The 'short' domains hybridize (bind) *reversibly* to their complements, while the 'long' domains hybridize *irreversibly*; the exact critical length depends on physical condition. Distinct letters indicate domains that do not hybridize with each other.

In the first reaction of Figure 1, a short *toehold* domain t initiates binding between a double strand and a single strand. After the (reversible) binding of the toehold, the x domain of the single strand gradually replaces the top x strand of the double strand by *branch migration*. The branching point between the two top x domains performs a random walk that eventually leads to *displacing* the x strand. The final detachment of the top x strand makes the whole process essentially irreversible, because there is no toehold for the reverse reaction. The second reaction illustrates the case where the top domains do not match: then the toehold binds reversibly and no displacement occurs. The third reaction illustrates the more detailed situation where the top domains match only initially: the branch migration can proceed only up to a certain point and then must revert back to the toehold: hence no displacement occurs and the whole reaction reverts.

The fourth reaction illustrates a *toehold exchange*, where a branch migration (of strand tx) leads to a displacement (of strand xt), but where the whole process is reversible via a reverse toehold binding and branch migration. The first (irreversible) and fourth (reversible) reactions are the fundamental steps that can be composed to achieve computation by strand displacement.

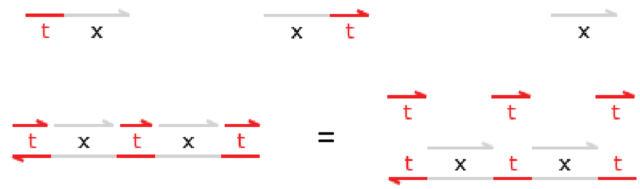


Fig. 2: Examples of allowable single and double strands: $tx, xt, x, \underline{t^\dagger x^\dagger t^\dagger x^\dagger t}$

2 Two-domain Signals and Gates

We now describe some DNA strand displacement structures that emulate, depending on the point of view, either chemical reactions or Petri net transitions. Their function is to *join* input signals and *fork* output signals. To achieve compositionality, so that gates can be composed arbitrarily into larger circuits, it is necessary to first fix the structure of the signals. Any given choice of signal structure requires a different gate architecture, for example for 4-domain signals [9] (signals composed of 4 segments of different function), and 3-domain signals [1]. Here we present a new, streamlined, architecture based on 2-domain signals, where the gates can be combined into arbitrary circuits (including loops), and where the waste products do not interfere with the active gates.

Top-nicked double strands.

Double-stranded DNA (*dsDNA*) can have interruptions (*nicks*) on one strand while remaining connected if the opposite strand has enough hold on the area around the nick. We called such structures *nicked double-stranded DNA* (*ndsDNA*). This excludes any long overhangs or any protrusions from the double-strand. In particular, we work with *top-nicked double-strands*, where all the nicks are on one strand (the top one by convention). A deviation from this simple structure happens fleetingly during branch migration, but all the initial and final species we use are ndsDNA.

We use t for short domains, x,y,z for long domains, and a,b,c for long domains that are meant to be privately used by some construction. We write, e.g., tx for a single-stranded DNA (*ssDNA*) strand consisting of a toehold t followed by a domain x, and similarly for xt. We write, e.g., $\underline{txy}$ for a fully complemented double strand consisting of a continuous strand txy at the top and its Watson-Crick complement at the bottom. Finally, we write $\underline{tx^\dagger y}$ to indicate the same double strand but with a nick at the top between x and y. In the figures, a nick is indicated by an arrowhead and a discontinuity.

Examples of allowable single and double strands are shown in Figure 2. We assume that domains indicated by different letters are distinct, so that, e.g., x does not hybridize with y, zy, yz, ty, or yt. To simplify our notation, we use an implicit equivalence illustrated in the bottom part of the figure. Suppose we start with a regular double strand, and we nick it at the top (bottom left).

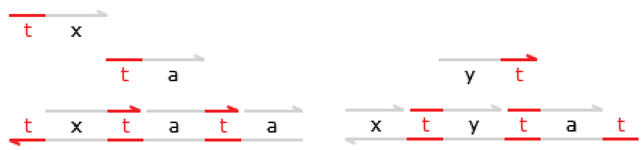


Fig. 3: Transducer $T_{xy} \mid tx \to ty$: initial state plus input tx.

Long segments between nicks remain attached to the bottom strand, while short toehold segments can detach and reattach (bottom right). We regard these reversible states as equivalent; the notation $\underline{x^\dagger t^\dagger y}$ then indicates two equivalent situations, where the top t is either present or absent, and where t is implicitly exchanged with the environment. Hence, we can use $\underline{x^\dagger t^\dagger y}$ to indicate an open toehold between x and y, because the toehold is available (sometime). This way, we do not need to use separate notations for temporarily occluded and temporarily open toeholds, which we would have to regard as equivalent anyway (up to some kinetic occlusion effects).

Two-domain strand displacement gates.

All our gates are top-nicked dsDNA and our signals are two-domain ssDNA. This simple setup is more expressive than it might appear at first. For example (Figure 3), let us consider a single strands tx as encoding a *signal*, with the strand xt as its *cosignal*, and consider the problem of constructing a *signal transducer* T_{xy} from a signal tx to a signal ty, with the *reduction* $T_{xy} \mid tx \to ty$, where $\mid$ is *parallel composition* of components, and final waste is discarded. All signals share the same toehold t, and are distinguished by the long domains x,y,z, etc. As shown in Figure 4, the input tx can initiate a signal/cosignal cascade of strand displacements in the left double-strand that after two toehold exchanges releases a *private* cosignal at (the segment a is privately used by the T_{xy} transducer, with a distinct a for each xy pair). The at cosignal then initiates a backward cascade in the right double strand that releases the desired output signal ty at the fourth reaction. The release of ty is reversible, but the gate is then locked down by the last two reactions. The locking down of the gate is also used to reabsorb the xt and ta strands, by exploiting the $\underline{x}$ end of the right structure and the $\underline{a}$ end of the left structure. In the end, only *unreactive* (no exposed toeholds) dsDNA and ssDNA is left. In Figure 4, the initial structures from Figure 3 are shown inside rounded rectangles, and the final structures inside squared rectangles. The reaction rules are described abstractly in Figure 10.

The structures in Figure 3 can be written in the notation described above as $T_{xy} = \underline{t^\dagger xt^\dagger at^\dagger a} \mid ta \mid \underline{x^\dagger ty^\dagger ta^\dagger t} \mid yt$. The auxiliary signal ta contains the private segment a, uniquely joining the two halves of T_{xy} transducers, and we can therefore assume that it will not interfere with other gates. The auxiliary

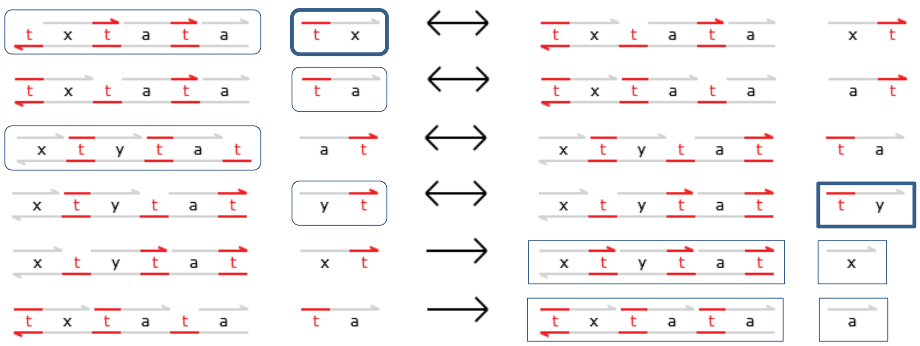


Fig. 4: Transducer $T_{xy} \mid tx \to ty$ reactions.

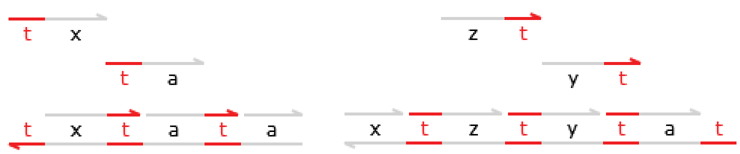


Fig. 5: Fork $F_{xyz} \mid tx \to ty \mid tz$: initial state plus input tx.

cosignal yt however contains a public segment y, which is necessary to release the output signal. It is therefore important to maintain an invariant that no other gate in the whole system spontaneously absorbs yt, or in general any public cosignal, although it may do so in a proper response to inputs. For example, a T_{zy} transducer and a T_{xy} transducer may use "each other's" yt cosignal without problem.

The transducer T_{xy} can be extended easily to a fork gate F_{xyz} such that $F_{xyz} \mid tx \to ty \mid tz$, releasing two outputs from one input. This is shown in Figure 5, where the left half of the structure is the same as in T_{xy}. The fork gate can be extended to a catalytic gate C_{xyz} such that $C_{xyz} \mid tx \mid ty \to ty \mid tz$ (Figure 6). The right half of C_{xyz} is unchanged from F_{xyz}, except that yt is not required because it is produced by the left half. This gate, like the more general join gate discussed next, takes two inputs, but absorbs them only if both inputs are present [9]. If only the first input is present, it is returned to the soup by

Fig. 6: Catalyst $C_{xyz} \mid tx \mid ty \to ty \mid tz$: initial state plus inputs tx and ty.

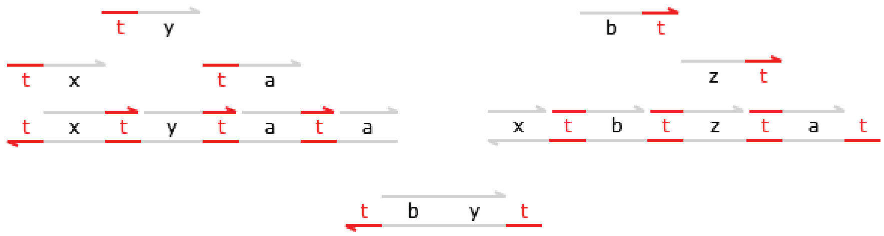


Fig. 7: Join $J_{xyz} \mid tx \mid ty \rightarrow tz$: initial state plus inputs tx, ty.

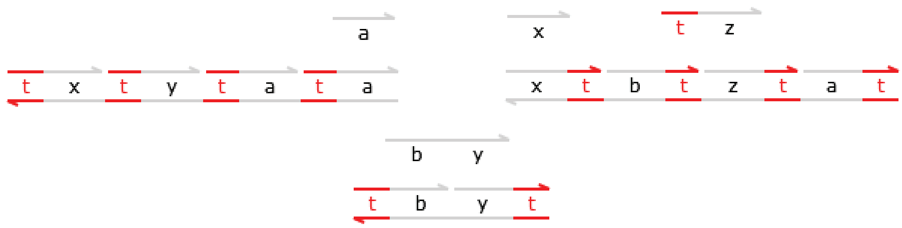


Fig. 8: Join $J_{xyz} \mid tx \mid ty \rightarrow tz$: final state plus output tz.

reversibility of strand displacement between tx and xt.

Let us now consider, in Figures 7 and 8, a binary join gate J_{xyz} such that $J_{xyz} \mid tx \mid ty \rightarrow tz$ (the generalization to additional outputs works as in the fork gate). Each distinct combination of xyz requires choosing a distinct private domain connecting the two halves of the gate; this private domain can however be shared among a population of gates with the same input and output signals. The main new feature in this gate is the additional $\underline{t^\dagger b y^\dagger t}$ structure that absorbs a signal and a cosignal together, or neither separately. Without it, and without the bt, $\underline{tb}$ components, the join gate would leave behind a yt residual (all the other single strands, xt, zt, ta, are reclaimed). Hence $\underline{t^\dagger b y^\dagger t}$ is a 'garbage collector' turning undesired active residuals to waste. It is triggered only after the release of a private strand tb, so that the collector does not reclaim an extraneous cosignal yt before the join gate has committed to its inputs. Such an extraneous yt could come from a transducer T_{xy}, or from another join J_{uvy} (before any input) or J_{yuv} (after the first input) causing cross-gate interference, or even from within the same join, as in J_{xyy}. Removing garbage is important because accumulated garbage slows down future reactions by imposing a growing reverse pressure on the desired direction of the reactions. We have designed all gates to remove all active garbage, but until now garbage removal did not require additional double strands. The Join structure is easily generalized to any number of inputs; for example, Figure 9 shows a 3-input Join with collectors.

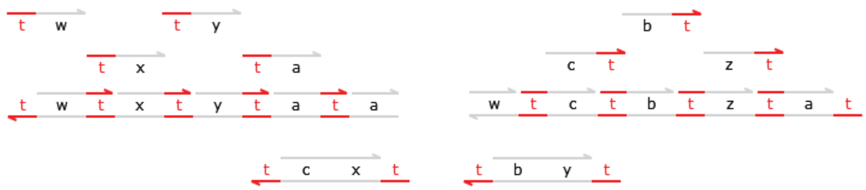


Fig. 9: 3-Join $J_{wxyz} \mid tw \mid tx \mid ty \rightarrow tz$: initial state plus inputs tw, tx, ty.

Discussion: The double strand restrictions.

The restriction of allowing only ndsDNA structures has a number of potential advantages. The absence of any branching seems inherently more trouble-free than complex structures that can interact in unexpected ways through their protruding single-stranded parts. Here all double-stranded structures are quiescent (except for receptive toeholds on the bottom strand) and only single-stranded components have hybridization potential, eliminating the possibility that the gate themselves may polymerize, or may self-interact. These structures also have a simple syntactical representation and simple reduction rules, which simplify formal verification. Nothing prevents us from devising precise syntax and reductions for more general structures [7], and there is no good reason in principle to avoid more complex structures if they work well. However, we have shown that our simplified structures already cover a surprising range of computation (fork and join gates in populations are equivalent to Petri Nets [1]), and hence one can restrict the use of more complex structures to the situations where they are actually needed, or where they somehow perform better.

Discussion: The single strand restrictions.

Our hybridized structures start as ndsDNA, but we have to ensure that they remain ndsDNA through computation. (Except for *transients*, i.e., during branch migrations that either revert harmlessly or lead to strand displacements.) This invariant puts constraints on the allowable single strands. First of all, single strands consisting only of long segments are inert because all the double strands are fully complemented (except for toeholds), and hence they can be ignored. A single strand of the form xty could bind to a double strand of the form $\underline{x^\dagger t^\dagger z}$, leading to a configuration that is stable and is not ndsDNA. Therefore our single strands cannot contain substrands of the form xty, and we are left with single strands of the form, $x^n t^m$ or $t^n x^m$ or $t^n x^m t^p$. The third class could lead to stable configurations with two overlapping competing toeholds ($\underline{t^\dagger x^\dagger t^\dagger y^\dagger t}$ with txt and tyt) and hence are ruled out too. Multiple toeholds in sequence bind as stably as a long domain, so e.g. $xttt$ would be as bad as the former xty, and they can lead to competing toeholds: $\underline{x^\dagger t^\dagger t^\dagger y}$ with xtt and tty. Hence we do not allow consecutive toeholds in the top strands. Similarly, strands with consecutive long segments can lead to stable competition: txy and yzt over

$\underline{t^{\dagger}xyz^{\dagger}t}$. In the end, we are left only with xt or tx, and the only remaining competition is between tx and xt over $\underline{t^{\dagger}x^{\dagger}t}$, where the stable structures are ndsDNA. A final case to consider is tx and yt over $\underline{t^{\dagger}xy^{\dagger}t}$: if a single strand is present it binds only reversibly, and if both are present they both bind stably and release xy, so the stable structures are always ndsDNA. In fact, $\underline{t^{\dagger}xy^{\dagger}t}$ is an important configuration that seems to add some power: without it we can still implement garbage-collecting join gates, but apparently only by using more than one distinct toehold.

Discussion: The double strand restrictions, revisited.

We finally have to make sure that no reactive single strands other than t, tx, xt, plus the unreactive x and xy, are ever released from double strands during computation. This imposes another restriction on double strands: nicks should break the top strand into segments of two domains or less. Otherwise, the double strand $\underline{t^{\dagger}xty^{\dagger}t}$ could release a forbidden single strand xty in presence of tx and yt. (We could still allow $\underline{t^{\dagger}xyz^{\dagger}t}$, but it would be unreactive.) Hence, we are left with allowable double strands that are nicked concatenations of the double-stranded elements $\underline{t}$, $\underline{x}$, $\underline{tx}$, $\underline{xt}$, $\underline{xy}$.

3 Nick Algebra

In this section we provided a formal framework where we can perform calculations about the evolution of systems of top-nicked double strands. *Domains* are taken either from a finite set of *short domains* (*toeholds*) or from an unbounded set of *long domains* ranged over by x,y,z and a,b,c. The set of toeholds must be finite (and in practice quite small) because of its reversible-binding assumption that limits length and hence cardinality. Designs based on a single toehold can be easily adapted to multiple toeholds to increase binding discrimination and efficiency, but the converse is problematic: designs based on distinct toeholds may fail if the toeholds are then identified. Here we require only a single distinguished toehold, always indicated by the constant t, but it would be easy to generalize to multiple toeholds.

An infix operator '.' may be used to concatenate domains into single strands; this is often omitted, particularly because all our single-strands have the form $t.x$ or $x.t$, which are then usually written tx and xt (unless we wish to use long identifiers for domains). Single strands t, x, and $x.y$ remain implicit 'waste', and are not used in the syntax.

Double strands are written underlined. We use an infix operator '$\underline{\dagger}$' to represent a 'nick' on the top strand of a double-stranded sequence, an infix operator '$\underline{.}$' (often omitted) to represent the unbroken concatenation of top and bottom strands, and ϕ for the empty double strand. The segments between nicks are only single or pair combinations of toeholds and domains.

A soup U is a finite multiset of single and double strands, with multiset union indicated by ' | ', and with a notation $(\nu x)U$ for domain isolation. The

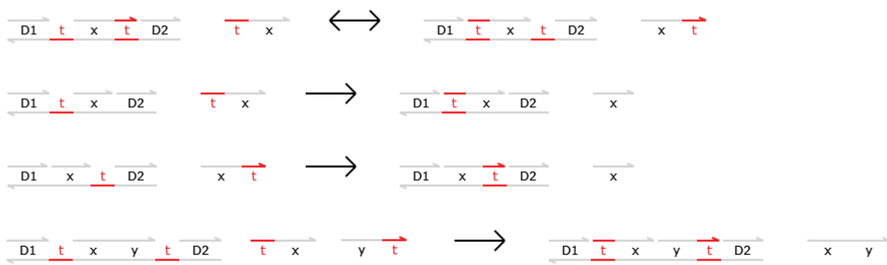


Fig. 10: The basic reactions (D1,D2 are arbitrary or empty double strands).

latter indicates that x is not used outside of U: this allows us to declare private domains locally, and to combine constructions compositionally. In practice, it means simply that all the domains indicated by ν must be chosen distinct when a global system is fixed for execution: the algebraic laws for $(\nu x)U$ encode such a guarantee. We also use U^n as an abbreviation for n copies of U in parallel (|). The resulting algebra is our nick algebra, which is strictly a subset of the DSD (DNA Strand Displacement) language [7].

Definition 1. Term Syntax

$S ::= t.x \mid x.t$	Single strand
$\underline{D} ::= \phi \mid \underline{t} \mid \underline{x} \mid \underline{t.x} \mid \underline{x.t} \mid \underline{x.x} \mid \underline{D^\dagger D}$	Double strand
$U ::= S \mid \underline{D} \mid U\vert U \mid (\nu x)U$	Soup

The set of *public domains* $pd(U)$ is the inductively defined set of those domains not bound by ν in U; in particular $pd(t.x) = pd(x.t) = pd(\underline{x}) = pd(\underline{t.x}) = pd(\underline{x.t}) = \{x\}$, $pd(\underline{x.y}) = \{x, y\}$, $pd(\underline{t}) = pd(\phi) = \{\}$, and $pd((\nu x)U) = pd(U) - \{x\}$. Then, $U\{y/x\}$ is the substitution of y for x in U, with the representative cases $t\{y/x\} = t$, $x\{y/x\} = y$, $z\{y/x\} = z$ for $z \neq x$, $((\nu z)U)\{y/x\} = (\nu z)U\{y/x\}$ for $z \notin \{x, y\}$, $((\nu x)U)\{y/x\} = (\nu x)U$, and $((\nu y)U)\{y/x\} = (\nu z)U\{z/y\}\{y/x\}$ for a $z \notin pd(U) \cup \{x, y\}$.

Algebraic equality (a binary congruence relation over the term syntax) is indicated just by $=$ and is axiomatized below with the monoid laws of $(\phi, \underline{\dagger})$, the commutative monoid laws of $(\phi, \mid)$, and the scoping laws of $(\nu x)U$ [6].

Definition 2. Algebraic Equality

$=$ is an equivalence relation

$$\underline{D_1} = \underline{D_2},\quad \underline{D_3} = \underline{D_4} \quad\Rightarrow\quad \underline{D_1{}^\dagger D_3} = \underline{D_2{}^\dagger D_4}$$
$$U_1 = U_2,\quad U_3 = U_4 \quad\Rightarrow\quad U_1 \mid U_3 \;=\; U_2 \mid U_4$$
$$U_1 = U_2 \quad\Rightarrow\quad (\nu x)U_1 \;=\; (\nu x)U_2$$

$$\underline{D_1{}^\dagger(D_2{}^\dagger D_3)} = \underline{(D_1{}^\dagger D_2)^\dagger D_3}$$
$$\underline{\phi^\dagger D} = \underline{D^\dagger \phi} = \underline{D}$$

$$U_1 \mid (U_2 \mid U_3) = (U_1 \mid U_2) \mid U_3$$
$$U_1 \mid U_2 = U_2 \mid U_1$$
$$\phi \mid U = U \mid \phi = U$$

$(\nu x)U = (\nu y)(U\{y/x\}) \qquad \text{if } y \notin pd(U)$
$(\nu x)\phi = \phi$
$(\nu x)(U_1 \mid U_2) = U_1 \mid (\nu x)U_2 \qquad \text{if } x \notin pd(U_1)$
$(\nu x)(\nu y)U = (\nu y)(\nu x)U$

Note that $(\nu x)(\nu x)U = (\nu x)U$ is derivable. As an example of use of the isolation operation, consider that it is always possible to bring all the ν prefixes to the top level by making all the private domains distinct: $(\nu x)tx \mid (\nu x)tx = (\nu x)tx \mid (\nu y)ty = (\nu x)(\nu y)(tx \mid ty)$. This means that conflicts between local definitions can be resolved globally, while allowing local definition to be combined without consideration of global conflicts.

The *reduction* relation $U_1 \rightarrow U_2$ describes a single step of system evolution; it is the smallest binary relation on U satisfying the rules below, where $\leftrightarrow$ stands for two reduction rules in opposite directions. Its symmetric and transitive closure $U_1 \rightarrow^* U_2$ describes multi-step system evolution. In the reduction rules, the ***single-stranded waste*** (t, x, xy) is automatically removed because it can be immediately identified as waste (as a consequence, the single strands t, x, xy need not be included in the syntax). Alternatively, we could have made the single-stranded waste explicit and introduced separate rules to remove it. The double-stranded waste instead has a special degradation rule because it requires a check over the whole double strand. The four basic reactions (exchange, coverage, cooperation) are depicted in Figure 10.

Definition 3. Reduction

$\underline{D_1{}^\dagger t^\dagger x t^\dagger D_2} \mid tx \leftrightarrow \underline{D_1{}^\dagger t x^\dagger t^\dagger D_2} \mid xt$	Exchange
$\underline{D_1{}^\dagger t^\dagger x^\dagger D_2} \mid tx \rightarrow \underline{D_1{}^\dagger t x^\dagger D_2}$	Left coverage
$\underline{D_1{}^\dagger x^\dagger t^\dagger D_2} \mid xt \rightarrow \underline{D_1{}^\dagger x t^\dagger D_2}$	Right coverage
$\underline{D_1{}^\dagger t^\dagger x y^\dagger t^\dagger D_2} \mid tx \mid yt \rightarrow \underline{D_1{}^\dagger t x^\dagger y t^\dagger D_2}$	Cooperation
$\underline{D} \rightarrow \phi \qquad \text{if } \underline{D} \text{ not } \textit{reactive}$	Waste
$U_1 \rightarrow U_2 \;\Rightarrow\; U_1 \mid U \rightarrow U_2 \mid U$	Dilution
$U_1 \rightarrow U_2 \;\Rightarrow\; (\nu x)U_1 \rightarrow (\nu x)U_2$	Isolation
$U_1 = U_2, U_2 \rightarrow U_3, U_3 = U_4 \;\Rightarrow\; U_1 \rightarrow U_4$	Well-mixing

A double strand $\underline{D}$ is *reactive* if it can react in some context; that is, by the first four rules. Hence it must be of the form $\underline{D_1{}^\dagger t^\dagger x t^\dagger D_2}$, $\underline{D_1{}^\dagger t x^\dagger t^\dagger D_2}$, $\underline{D_1{}^\dagger t^\dagger x^\dagger D_2}$, $\underline{D_1{}^\dagger x^\dagger t^\dagger D_2}$, or $\underline{D_1{}^\dagger t^\dagger x y^\dagger t^\dagger D_2}$. Among the unreactive (waste) double strands are thus $\underline{t}$, $\underline{x}$, $\underline{xt}$, $\underline{tx}$, $\underline{xy}$, $\underline{t^\dagger t}$, $\underline{t^\dagger tx}$, $\underline{xt^\dagger t}$, $\underline{xt^\dagger ty}$, $\underline{xt^\dagger t^\dagger ty}$, etc. The waste rule is really a convenience to simplify results of calculations; more generally, as commonly done in process algebra, one would instead eliminate unreactive components via an observational equivalence [6].

4 Correctness

If $U_1 \rightarrow^* U_2$ then U_1 may reduce to U_2, but it may also reduce to something else since $\rightarrow^*$ is a relation. When $U_1 \rightarrow^* U_2$ is used to state a correctness property of system reduction, we say that this is a *may-correctness* property: the system starting from U_1 *may* reduce to U_2, but it may also wander in a different section of state space and never be able to get to U_2 from there. A stronger property is *will-correctness*, indicated by $U_1 \rightarrow^{\forall} U_2$, and defined as $\forall U, U_1 \rightarrow^{*} U \;\;\Rightarrow\;\; U \rightarrow^{*} U_2$. This means that although U_1 may wander to some U in some part of the state space, it *will* always find a path to U_2 from there (it *cannot avoid* finding a path to U_2). If $U_1 \rightarrow^{\forall} U_2$ and U_2 is the only terminal state, then we can say that U_1 *must reduce* to U_2. But will-correctness does not imply that reduction necessarily terminates, and in particular if $U \rightarrow^{\forall} U$ we can say that U is *reversible*. Since $U_1 \rightarrow^{*} U_1$ holds by reflexivity, will-correctness implies may-correctness. (All these properties are really examples of a large class of reachability properties that could be expressed in a temporal logic.)

It is convenient in the next examples and proofs to use a more pictographic notation for nick algebra expressions, to highlight the positions of the toeholds. We use the following abbreviations ($\underline{\dagger}$ is still needed in for $\underline{x^{\dagger}y}$):

Definition 4. Two-Domain Pictograms

$\ulcorner x$	for tx	*Signal*
$x\urcorner$	for xt	*Cosignal*
$\underline{D\ulcorner x}$	for $\underline{D^{\dagger}tx}$ (including $D = \phi$)	Bound signal
$\underline{x\urcorner D}$	for $\underline{xt^{\dagger}D}$ (including $D = \phi$)	Bound cosignal
$\underline{D\smile D'}$	for $\underline{D^{\dagger}t^{\dagger}D'}$ (including $D = \phi$ or $D' = \phi$)	Bottom toehold

For example, the transducer from Figure 3 can be written as:

$\underline{t^{\dagger}xt^{\dagger}at^{\dagger}a} \mid ta \mid \underline{x^{\dagger}ty^{\dagger}ta^{\dagger}t} \mid yt$	explicit notation
$\underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner$	pictogram notation

We now show that the transducer *may* work correctly. Because of their chemical origin, all components come in populations of identical molecules, and any private domain can only be private to a population, and not to an individual molecule. Hence we need to show that a populations of transducers, all sharing the same private domain, *may* map an input population to a desired output population.

Proposition 5. *Transducer T^{n}_{xy} May-Correctness*

Let $T^{n}_{xy} = (\nu a)((\underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner)^{n})$,
then $T^{n}_{xy} \mid \ulcorner x^{n} \rightarrow^{*} \ulcorner y^{n}$.

Proof. Let $T_{xay} = \underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner$ for $a \neq x, y$, so that $T^{n}_{xy} = (\nu a)((T_{xay})^{n})$. We first show that $T_{xay} \mid \ulcorner x \rightarrow^{*} \ulcorner y$.

$T_{xay} \mid \ulcorner x$
$= \underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \ulcorner x$
$\leftrightarrow \underline{\ulcorner x\smile a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid x\urcorner$
$\leftrightarrow \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid x\urcorner \mid a\urcorner$
$\leftrightarrow \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner \mid \ulcorner a$
$\rightarrow \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner$
$\rightarrow \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner$

$\leftrightarrow \underline{x \smile y \urcorner a \urcorner} \mid x \urcorner \mid \ulcorner y$
$\rightarrow \underline{x \urcorner y \urcorner a \urcorner} \mid \ulcorner y$
$\rightarrow \ulcorner y$

Hence $(T_{xay} \mid \ulcorner x)^n \rightarrow^* \ulcorner y^n$ by induction, $(T_{xay})^n \mid \ulcorner x^n \rightarrow^* \ulcorner y^n$ by associativity, $(\nu a)((T_{xay})^n \mid \ulcorner x^n) \rightarrow^* (\nu a) \ulcorner y^n$ by isolation, and $T^n_{xy} \mid \ulcorner x^n \rightarrow^* \ulcorner y^n$ by ν-equivalence and by T^n_{xy} definition. □

We can similarly check the may-correctness of fork and join gates:

Proposition 6. *Fork F^n_{xyz} May-Correctness*

Let $F^n_{xyz} = (\nu a)((\underline{\smile x \urcorner a \urcorner a} \mid \ulcorner a \mid \underline{x \ulcorner z \ulcorner y \ulcorner a \smile} \mid z \urcorner \mid y \urcorner)^n)$,
then $F^n_{xyz} \mid \ulcorner x^n \rightarrow^* \ulcorner y^n \mid \ulcorner z^n$.

Proposition 7. *Join J^n_{xyz} May-Correctness*

Let $J^n_{xyz} = (\nu a)(\nu b)((\underline{\smile x \urcorner y \urcorner a \urcorner a} \mid \ulcorner a \mid \underline{x \ulcorner b \ulcorner z \ulcorner a \smile} \mid b \urcorner \mid z \urcorner \mid \underline{\smile b^\dagger y \smile})^n)$,
then $J^n_{xyz} \mid \ulcorner x^n \mid \ulcorner y^n \rightarrow^* \ulcorner z^n$.

Consider now the difficulties involved in proving more interesting properties. We would like a transducer, for example, to work correctly in 'all possible contexts'. Unfortunately that is just not true, because some context could absorb the $y \urcorner$ strand, which is public, and interfere with the transducer. One would have to consider instead 'all possible contexts that do not interfere with $y \urcorner$'. This is a rather awkward notion: for compositionality one would have, for each component, to keep track of all the elements in the context that the component might be interfering with. Moreover, the transducer interferes with $y \urcorner$, and hence it interferes with (another copy or another population of) itself.

Let us consider a simpler 'progress' property: that the transducer does not deadlock with itself. This can be expressed as a will-correctness property, that for any intermediate state U, if $T^n_{xy} \mid tx^n \rightarrow^* U$ then $U \rightarrow^* ty^n$. This appears to require an induction on all possible intermediate configurations U for any n. Even for a fixed small n, the state space U can grow very large, which suggests that automated state exploration tools should be useful. Note also that an induction on the length of $\rightarrow^*$ is problematic because of the reversible exchange rule: infinite sequences of reductions exist in almost all systems. In a stochastic interpretation of reduction, actual convergence can often be achieved (with measure 1), and this is another challenging property to prove.

We now illustrate how to check a will-correctness property, for a single copy of a transducer:

Proposition 8. *T^1_{xy} Will-Correctness*

$T^1_{xy} \mid \ulcorner x \rightarrow^\forall \ulcorner y$. Moreover, $\ulcorner y$ is the only reachable terminal state.

Proof. We show that if $T^1_{xy} \mid \ulcorner x \rightarrow^* U$ then $U \rightarrow^* \ulcorner y$. We enumerate all distinct states U, up to algebraic equality, arising from $T^1_{xy} \mid tx$ by all possible traces, and then we check that each state can lead to $\ulcorner y$. Assume $x \neq y$; indentation means a branch in the derivation:

01. $(\nu a)\ \underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \ulcorner x$
02. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\smile a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid x\urcorner$
03. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid x\urcorner \mid a\urcorner$
04. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner \mid \ulcorner a$
05. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner$
06. $\rightarrow (\nu a)\ \underline{x\ulcorner y\smile a\urcorner} \mid y\urcorner \mid x\urcorner$
07. $\leftrightarrow (\nu a)\ \underline{x\smile y\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner y$
08. $\leftrightarrow (\nu a)\ \underline{x\urcorner y\urcorner a\urcorner} \mid \ulcorner y$
09. $\rightarrow \ulcorner y$
10. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\smile y\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner y \quad \rightarrow 07$
11. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\urcorner y\urcorner a\urcorner} \mid \ulcorner y \quad \rightarrow 08$
12. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \ulcorner y \quad \rightarrow 09$
13. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\smile y\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner a \mid \ulcorner y \quad \leftrightarrow 10$
14. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\urcorner y\urcorner a\urcorner} \mid \ulcorner a \mid \ulcorner y \quad \leftrightarrow 11$
15. $\leftrightarrow (\nu a)\ \underline{\ulcorner x\ulcorner a\smile a} \mid \ulcorner a \mid \ulcorner y \quad \leftrightarrow 12$

All other states (up to algebraic equality) can be reduced to these states by well-mixing. We can then check that all these states have a path to state 9. The case for $x = y$ is similar: the state graphs is the same because, as can be seen above, there is never both an x redex and a different y redex in the same state, and when two x signals or cosignals can be chosen, it does not matter which one is chosen, by well-mixing. □

For transducer composition, the may-correctness property $T^n_{xy} \mid T^n_{yz} \mid \ulcorner x^n \rightarrow^* \ulcorner z^n$ follows simply from Proposition 5, but even just the will-correctness property $T^1_{xy} \mid T^1_{yz} \mid \ulcorner x \rightarrow^{\forall} \ulcorner z$ (including $x = z$ and $y = z$ and $x = y = z$) does not follow from Proposition 8, and requires the analysis of a product state space. For example, $T^1_{xy} \mid T^1_{yx}$ can absorb the inputs $\ulcorner x \mid \ulcorner y$ sequentially (converting $\ulcorner x$ to a second $\ulcorner y$ and then $\ulcorner y$ to $\ulcorner x$) or in parallel (each transducer starting to process an input before producing an output). In fact, consider the following transducer that uses a public 'a' domain instead of a private one:

$$T_{xay} = \underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner$$

T_{xay} by itself satisfies may and will-correctness as shown above for T^1_{xy}, and so does T_{yax}. But the two together do not satisfy the will-correctness property of just producing $\ulcorner x$ on input $\ulcorner x$, because the following 'crosstalk' derivation is possible, where in the third step $a\urcorner$ goes to the 'wrong' gate:

$T_{xay} \mid T_{yax} \mid \ulcorner x$
$= \underline{\smile x\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\ulcorner x\ulcorner a\smile} \mid x\urcorner \mid \ulcorner x$
$\leftrightarrow \underline{\ulcorner x\smile a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\ulcorner x\ulcorner a\smile} \mid x\urcorner \mid x\urcorner$
$\leftrightarrow \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\ulcorner x\ulcorner a\smile} \mid x\urcorner \mid x\urcorner \mid a\urcorner$
$\leftrightarrow \underline{\ulcorner x\ulcorner a\smile a} \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\ulcorner x\smile a\urcorner} \mid x\urcorner \mid x\urcorner \mid \ulcorner a$
$\rightarrow \underline{\ulcorner x\ulcorner a\ulcorner a} \mid \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\ulcorner x\smile a\urcorner} \mid x\urcorner \mid x\urcorner$
$\rightarrow \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\ulcorner x\smile a\urcorner} \mid x\urcorner \mid x\urcorner$
$\leftrightarrow \underline{x\ulcorner y\ulcorner a\smile} \mid y\urcorner \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\smile x\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner x$
$\rightarrow \underline{x\ulcorner y\ulcorner a\smile} \mid \underline{\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{y\urcorner x\urcorner a\urcorner} \mid x\urcorner \mid \ulcorner x$

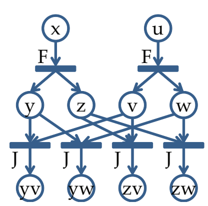
$\rightarrow \underline{x \ulcorner y \ulcorner a \smile} \mid \underline{\smile y \urcorner a \urcorner a} \mid \ulcorner a \mid x \urcorner \mid \ulcorner x$

The last state is final (no further progress can be made), and is not just the expected $\ulcorner x$ (which can be obtained by a different derivation). Moreover, no $\ulcorner y$ is ever produced. The system is deadlocked in a state where the output $\ulcorner x$ has been produced, but many other active components have been left to interfere with future operation. However, that last state, if supplied with an additional $\ulcorner y$, then unblocks and reduces just to $\ulcorner y \mid \ulcorner x$. Hence, although $T_{xay} \mid T_{yax} \mid \ulcorner x \nrightarrow^{\forall} \ulcorner x$, we have that $T_{xay} \mid T_{yax} \mid \ulcorner x \mid \ulcorner y \rightarrow^{\forall} \ulcorner x \mid \ulcorner y$. That means that a large population of such gates in practice does not deadlock easily over an input population of $\ulcorner x$: each pair of stuck gates can be unblocked by another gate correctly producing a $\ulcorner y$, and it is very unlikely that a large fraction of gates ends up being blocked. This can be seen in stochastic simulations of large populations, and also in Ordinary Differential Equation simulations with unit concentration of $T_{xay} \mid T_{yax}$, where the concentration of the residual $\ulcorner a$ tends asymptotically to zero. Hence, another interesting property of these system is that, even though small populations may deadlock, large populations may converge to an almost-correct solution with high probability.

5 Testing

Gate and circuits designs have been tested with the DSD tool [7]. We give a simple example here, testing a combination of two fork and four join gates in the following configuration, where yv, yw, zv, zw are four output domains (i.e., yv does not mean $y.v$ in this section).

$F^n_{x,y,z} \mid$
$F^n_{u,v,w} \mid$
$J^n_{y,v,yv} \mid$
$J^n_{y,w,yw} \mid$
$J^n_{z,v,zv} \mid$
$J^n_{z,w,zw} \mid$
$\ulcorner x^m \mid \ulcorner u^m$ (input, $m \leq n$)



Since fork and join gates accept inputs and produce outputs in a specific order, one should not expect identical rates of production of yv,yw,zv,zw. (If desired, one can mix populations of symmetric gates, to achieve symmetric behavior.) In Figure 11 we see an Ordinary Differential Equations simulation with unit rates for toehold binding and unbinding, and with concentrations of 1.0 for the input signals and 10.0 for the gates; hence 10% of each gates is consumed during the computation. The system has a total of 54 single strand species, 108 double strand species, and 172 reactions, and therefore 162 ODEs. At time 3 (left), yv is ahead out of the gates, with zw trailing last. At time 30 (middle left) yv and yw are closer, and zv and zw are closer. At time 300 (middle right) the computation has reached 90% completion with similar output quantities approaching the expected 0.5 concentration. The higher curve of the fourth graph shows the total accumulation of the four $\underline{\ulcorner D^{\dagger} D' \urcorner}$ garbage species

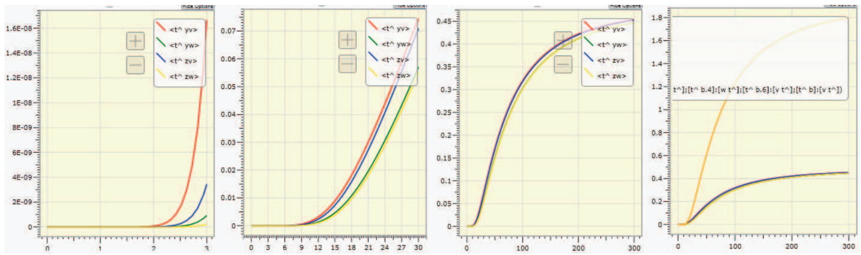


Fig. 11: Testing a fork/join circuit.

for the join gates, indicating that all the gates are being converted to waste. One can further examine the trajectories of all the species in the system to check that no deadlock occurs, and that all the structures are turned to output or to waste.

6 Conclusions

We have shown how to implement fork and join gates via simple two-domain structures, and how to implement them in a 'clean' way that automatically removes all active garbage. In essence, we have given an implementation of the higher-level strand algebra of [1]. But is this implementation correct? We have provided a formal framework where we can perform calculations and study such questions, and we have discussed some simple correctness definitions and some complex behavioral properties. A formal proof of absence of gate interference under all possible combinations and numbers of gates and inputs will require an extensive amount of case analysis, which likely needs to be automated, as well as the identification of appropriate invariants. Alternatively, one may gain confidence in the designs by simulation testing.

Acknowledgments

Figures were prepared with the DSD tool [7]. I would like to thank the members of the Molecular Programming Project at Caltech and U.Washington for many tutorials and discussions.

References

[1] L. Cardelli. Strand Algebras for DNA Computing. In DNA Computing and Molecular Programming. LNCS 5877, Springer, October 2009, pp 12-24.

[2] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, E. Shapiro. Programmable and Autonomous Computing Machine made of Biomolecules. Nature, 414(22), November 2001.

[3] W. Fontana. Pulling Strings. Science 314(8), 2006.

[4] S. J. Green, D. Lubrich, A. J. Turberfield. DNA Hairpins: Fuel for Autonomous DNA Devices. Biophysical Journal 91, October 2006, 2966–2975.

[5] M. Hagiya. Towards Molecular Programming. In G. Ciobanu, G. Rozenberg, (Eds.) Modelling in Molecular Biology. Springer, 2004.

[6] R. Milner. Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, 1999.

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

[8] G. Seelig, D. Soloveichik, D.Y. Zhang, E. Winfree. Enzyme-Free Nucleic Acid Logic Circuits. Science 314(8), 2006.

[9] D. Soloveichik, G. Seelig, E. Winfree. DNA as a Universal Substrate for Chemical Kinetics. PNAS 107 no. 12, 5393-5398.

[10] B. Yurke, A.P. Mills Jr. Using DNA to Power Nanostructures. Genetic Programming and Evolvable Machines archive 4(2), 111 - 122, Kluwer, 2003.

[11] D. Y. Zhang, A. J. Turberfield, B. Yurke, E. Winfree. Engineering Entropy-driven Reactions and Networks Catalyzed by DNA. Science, 318:1121-1125, 2007.

7 Appendix

7.1 May-Correctness of binary Fork and Join gates

Proposition 9. F^n_{xyz} *May-Correctness (Proposition 6)*

Let $F^n_{xyz} = (\nu a)((\underline{{\smile}x{\urcorner}a{\urcorner}a} \mid {\ulcorner}a \mid \underline{x{\ulcorner}z{\ulcorner}y{\ulcorner}a{\smile}} \mid z{\urcorner} \mid y{\urcorner})^n)$,
then $F^n_{xyz} \mid {\ulcorner}x^n \rightarrow^* {\ulcorner}y^n \mid {\ulcorner}z^n$.

Proof. Let $F_{xayz} = \underline{{\smile}x{\urcorner}a{\urcorner}a} \mid {\ulcorner}a \mid \underline{x{\ulcorner}z{\ulcorner}y{\ulcorner}a{\smile}} \mid z{\urcorner} \mid y{\urcorner}$ for $a \neq x, y, z$, so that$F^n_{xyz} = (\nu a)((F_{xayz})^n)$. We first show that $F_{xayz} \mid {\ulcorner}x \rightarrow^* {\ulcorner}y \mid {\ulcorner}z$.

$$\begin{array}{l}
F_{xayz} \mid {\ulcorner}x \\
= \underline{{\smile}x{\urcorner}a{\urcorner}}a \mid {\ulcorner}a \mid \underline{x{\ulcorner}z{\ulcorner}y{\ulcorner}a{\smile}} \mid z{\urcorner} \mid y{\urcorner} \mid x{\urcorner} \\
\leftrightarrow \underline{{\ulcorner}x{\smile}a{\urcorner}a} \mid {\ulcorner}a \mid \underline{x{\ulcorner}z{\ulcorner}y{\ulcorner}a{\smile}} \mid z{\urcorner} \mid y{\urcorner} \mid x{\urcorner} \\
\leftrightarrow \underline{{\ulcorner}x{\ulcorner}a{\smile}a} \mid \underline{x{\ulcorner}z{\ulcorner}y{\ulcorner}a{\smile}} \mid z{\urcorner} \mid y{\urcorner} \mid x{\urcorner} \mid a{\urcorner} \\
\leftrightarrow \underline{{\ulcorner}x{\ulcorner}a{\smile}a} \mid \underline{x{\ulcorner}z{\ulcorner}y{\smile}a{\urcorner}} \mid z{\urcorner} \mid y{\urcorner} \mid x{\urcorner} \mid {\ulcorner}a \\
\rightarrow \underline{{\ulcorner}x{\ulcorner}a{\ulcorner}a} \mid \underline{x{\ulcorner}z{\ulcorner}y{\smile}a{\urcorner}} \mid z{\urcorner} \mid y{\urcorner} \mid x{\urcorner} \\
\rightarrow \underline{x{\ulcorner}z{\ulcorner}y{\smile}a{\urcorner}} \mid z{\urcorner} \mid y{\urcorner} \mid x{\urcorner} \\
\leftrightarrow \underline{x{\ulcorner}z{\smile}y{\urcorner}a{\urcorner}} \mid z{\urcorner} \mid x{\urcorner} \mid {\ulcorner}y \\
\leftrightarrow \underline{x{\smile}z{\urcorner}y{\urcorner}a{\urcorner}} \mid x{\urcorner} \mid {\ulcorner}y \mid {\ulcorner}z \\
\rightarrow \underline{x{\urcorner}z{\urcorner}y{\urcorner}a{\urcorner}} \mid {\ulcorner}y \mid {\ulcorner}z \\
\rightarrow \ \mid {\ulcorner}y \mid {\ulcorner}z
\end{array}$$

Hence $(F_{xayz} \mid \ulcorner x)^n \to^* (\ulcorner y \mid \ulcorner z)^n$ by induction, $(F_{xayz})^n \mid \ulcorner x^n \to \ulcorner y^n \mid \ulcorner z^n$ by associativity, $(\nu a)((F_{xayz})^n \mid \ulcorner x_n) \to^* (\nu a)(\ulcorner y^n \mid \ulcorner z^n)$ by isolation, and $F^n_{xyz} \mid \ulcorner x^n \to^* \ulcorner y^n \mid \ulcorner z^n$ by ν-equivalence and by F^n_{xyz} definition. □

Proposition 10. *J^n_{xyz} May-Correctness (Proposition 7)*

Let $J^n_{xyz} = (\nu a)(\nu b)((\underline{\smile x\urcorner y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner b\ulcorner z\ulcorner a\smile} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile})^n)$,
then $J^n_{xyz} \mid \ulcorner x^n \mid \ulcorner y^n \to^* \ulcorner z^n$.

Proof. Let $J_{xyaz} = \underline{\smile x\urcorner y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner b\ulcorner z\ulcorner a\smile} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile}$ for $a \neq x, y, z$, so that $J^n_{xyz} = (\nu a)((J_{xyaz})^n)$. We first show that $J_{xyaz} \mid \ulcorner x \mid \ulcorner y \to^* \ulcorner z$.

$J_{xyaz} \mid \ulcorner x \mid \ulcorner y$
$= \underline{\smile x\urcorner y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner b\ulcorner z\ulcorner a\smile} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile} \mid \ulcorner x \mid \ulcorner y$
$\leftrightarrow \underline{\ulcorner x\smile y\urcorner a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner b\ulcorner z\ulcorner a\smile} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile} \mid \ulcorner y \mid x\urcorner$
$\leftrightarrow \underline{\ulcorner x\ulcorner y\smile a\urcorner a} \mid \ulcorner a \mid \underline{x\ulcorner b\ulcorner z\ulcorner a\smile} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile} \mid x\urcorner \mid y\urcorner$
$\leftrightarrow \underline{\ulcorner x\ulcorner y\ulcorner a\smile a} \mid \underline{x\ulcorner b\ulcorner z\ulcorner a\smile} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile} \mid x\urcorner \mid y\urcorner \mid a\urcorner$
$\leftrightarrow \underline{\ulcorner x\ulcorner y\ulcorner a\smile a} \mid \underline{x\ulcorner b\ulcorner z\smile a\urcorner} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile} \mid x\urcorner \mid y\urcorner \mid \ulcorner a$
$\to \underline{\ulcorner x\ulcorner y\ulcorner a\ulcorner a} \mid \underline{x\ulcorner b\ulcorner z\smile a\urcorner} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile} \mid x\urcorner \mid y\urcorner$
$\to \underline{x\ulcorner b\ulcorner z\smile a\urcorner} \mid b\urcorner \mid z\urcorner \mid \underline{\smile b^\dagger y\smile} \mid x\urcorner \mid y\urcorner$
$\leftrightarrow \underline{x\ulcorner b\smile z\urcorner a\urcorner} \mid b\urcorner \mid \underline{\smile b^\dagger y\smile} \mid x\urcorner \mid y\urcorner \mid \ulcorner z$
$\leftrightarrow \underline{x\smile b\urcorner z\urcorner a\urcorner} \mid \underline{\smile b^\dagger y\smile} \mid x\urcorner \mid y\urcorner \mid \ulcorner z \mid \ulcorner b$
$\to \underline{x\urcorner b\urcorner z\urcorner a\urcorner} \mid \underline{\smile b^\dagger y\smile} \mid y\urcorner \mid \ulcorner z \mid \ulcorner b$
$\to \underline{\smile b^\dagger y\smile} \mid y\urcorner \mid \ulcorner z \mid \ulcorner b$
$\to \ulcorner z$

Hence $(J_{xyaz} \mid \ulcorner x \mid \ulcorner y)^n \to^* \ulcorner z^n$ by induction, $(J_{xyaz})^n \mid \ulcorner x^n \mid \ulcorner y^n \to^* \ulcorner z^n$ by associativity, $(\nu a)((J_{xyaz})^n \mid \ulcorner x^n \mid \ulcorner y^n) \to^* (\nu a)\ulcorner z^n$ by isolation, and $J^n_{xyz} \mid \ulcorner x^n \mid \ulcorner y^n \to^* \ulcorner z^n$ by ν-equivalence and by J^n_{xyz} definition. □

7.2 DSD Script for Figure 11

This script can be run from a browser in DSD [7] using 'deterministic' simulation.

http://research.microsoft.com/en-us/projects/dna/default.aspx

```
directive sample 300.0 1000
directive plot <t^ yv>; <t^ yw>; <t^ zv>; <t^ zw>; sum([t^ _]:[_ t^])
new t@1.0,1.0

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

def J(N, x, y, z) =
new a new b
( N* <t^ a>
| N* <b t^>
```

```
| N* <z t^>
| N* t^:[x t^]:[y t^]:[a t^]:[a]
| N* [x]:[t^ b]:[t^ z]:[t^ a]:t^
| N* t^:[b y]:t^ )

( F(10, x, y, z)
| F(10, u, v, w)
| J(10, y, v, yv)
| J(10, y, w, yw)
| J(10, z, v, zv)
| J(10, z, w, zw)
| 1 * <t^ x>
| 1 * <t^ u> )
```