Published in final edited form as:

Theor Comput Sci. 2016 June 13; 632: 43–73. doi:10.1016/j.tcs.2015.07.041.

A strand graph semantics for DNA-based computation

Rasmus L. Petersen[1], **Matthew R. Lakin**[2], and **Andrew Phillips**[1]

[1]Microsoft Research, Cambridge, UK

[2]Department of Computer Science, University of New Mexico, Albuquerque, NM, USA

Abstract

DNA nanotechnology is a promising approach for engineering computation at the nanoscale, with potential applications in biofabrication and intelligent nanomedicine. DNA strand displacement is a general strategy for implementing a broad range of nanoscale computations, including any computation that can be expressed as a chemical reaction network. Modelling and analysis of DNA strand displacement systems is an important part of the design process, prior to experimental realisation. As experimental techniques improve, it is important for modelling languages to keep pace with the complexity of structures that can be realised experimentally. In this paper we present a process calculus for modelling DNA strand displacement computations involving rich secondary structures, including DNA branches and loops. We prove that our calculus is also sufficiently expressive to model previous work on non-branching structures, and propose a mapping from our calculus to a canonical *strand graph* representation, in which vertices represent DNA strands, ordered sites represent domains, and edges between sites represent bonds between domains. We define interactions between strands by means of strand graph rewriting, and prove the correspondence between the process calculus and strand graph behaviours. Finally, we propose a mapping from strand graphs to an efficient implementation, which we use to perform modelling and simulation of DNA strand displacement systems with rich secondary structure.

Keywords

strand graph; site graph; process calculus; programming language; DNA computing; molecular programming; biological computation

1 Introduction

Molecular computation is a powerful emerging technology for enabling programmable control of matter at the nanoscale. As our scientific understanding of molecular-scale phenomena continues to increase, new opportunities are being revealed for using nanoscale computation to influence macro-scale systems such as whole organisms [1]. Thanks to the sequence-specific chemistry of DNA interactions, DNA nanotechnology has emerged as a promising technique for implementing programmable nanoscale systems, which have great potential for biofabrication, biosynthesis, and diagnostic and therapeutic applications.

DNA strand displacement [49] is a simple yet powerful approach for implementing nanoscale computing systems, and has recently been used to implement a range of nanoscale computations, including multi-layer logic circuits [41, 35], catalytic amplification cycles [26, 50], artificial neural networks [37], and distributed algorithms [4]. It has also been shown that the quantitative behaviour of arbitrary chemical reaction networks can be implemented using DNA strand displacement [42, 4]. Furthermore, DNA strand displacement systems have been used in practical applications such as logic-based analysis of cell surface markers for cell sorting [39].

Formal modelling and analysis is an important step in the design of DNA strand displacement systems, enabling mechanized verification of circuit behaviour [20, 47] as well as fitting of reaction rate parameters from experimental data [4]. Such formal analysis requires a means of representing a DNA strand displacement system in a formal language, with a well-defined semantics that accurately models the interactions between system components. In previous work, we developed a domain-specific DNA strand displacement (DSD) language for modelling, simulating and analysing DNA strand displacement systems [33, 24]. The language was formalised as a process calculus with a precise syntax and semantics, and implemented as the Visual DSD software tool [25] to enable straightforward use by scientists. Over time, the scope of the DSD language has been extended to increase the diversity of DNA structures that can be modelled. Extensions were made to include structures of theoretical and practical interest, such as linear heteropolymers of unbounded lengths [23] for modelling Turing-powerful molecular computers [34], and strand displacement circuits tethered to a substrate [22] for modelling localised molecular systems [2, 44].

As the state of the art in DNA nanotechnology continues to develop, highly sophisticated molecular devices are being designed and subsequently implemented in the laboratory. Many of these devices involve rich secondary structures such as branches and loops [45, 36], which cannot be represented in the current DSD language, despite the various extensions that have already been incorporated. Furthermore, supporting such secondary structures requires fundamental changes that go beyond simple extensions to the existing language. Instead, it is necessary to completely reformulate the syntactic and semantic basis of the DSD language using a more general framework, capable of encoding arbitrary secondary structures and their interactions.

In this paper we present a language for modelling, simulating and analysing DNA strand displacement systems involving rich secondary structures such as branches and loops. The language is defined as a process calculus (Section 2), with a simple yet expressive syntax and a formal semantics that precisely models the behaviour of systems over time, to enable formal analysis. We define a corresponding graphical representation for the calculus, which can be used to conveniently visualise complex models. To ensure compatibility with previous work [24], we show that the calculus is sufficiently expressive to model all possible behaviours of our previous calculus for DNA strand displacement systems (Appendix C). To enable an efficient implementation of the language, we define a canonical representation based on a formal model of *strand graphs* (Section 3). This allows us to use the expressive power of graph theory both to represent secondary structures and to derive interactions via localized graph rewriting rules. Finally, we present an efficient implementation of the strand graph semantics (Section 4). A brief introduction to DNA strand displacement is also included (Appendix A).

2 Process calculus

In this section we present a language for modelling, simulating and analysing DNA strand displacement systems involving rich secondary structures such as branches and loops. The language is defined in Section 2.1 as a *process calculus* with a formal *syntax* and *semantics*. The formal syntax enables systems to be written down concisely as program code, to facilitate rapid construction and editing of models. The formal semantics enables the possible interactions between DNA strands to be computed automatically, for model simulation and formal analysis. In Section 2.2 we illustrate the syntax and semantics of the calculus through a number of examples, and present a corresponding graphical representation for the syntax, which can be used to conveniently visualise complex models. The graphical representation has a one-to-one correspondence with the textual syntax, such that both can be used interchangeably. To ensure compatibility with previous work, in Appendix C we prove that the calculus is sufficiently expressive to model all behaviours of our previous calculus for DNA strand displacement systems [24, 21], which we term *classic DSD*. We summarise the syntax and semantics of classic DSD in Appendix C.1, and present a translation from classic DSD to our calculus in Appendix C.2. We use the calculus semantics to prove in Appendix C.3 that our calculus is able to fully reproduce all behaviours of classic DSD. The proof illustrates how the calculus semantics can be used to perform formal analysis directly on processes of the calculus, without the need for additional transformations, following the approach of previous process calculi such as the pi-calculus [29, 30].

2.1 Syntax and semantics

The syntax of the process calculus is summarised in Definition 1. A process P is defined as a multiset of strands, where each strand $<S>$ is defined as a sequence S of one or more domains. A domain can be either free or bound, where a free domain is represented as d and a bound domain is represented as $d!i$, following similar notation introduced in [7], where i represents the bond name. A domain name x represents a nucleotide sequence, while a complement domain name x* represents the complement of the sequence x, such that x can bind to x* via Watson-Crick base pairing. Finally, a domain x is defined as a *toehold*, written $x\hat{}$, if the domain is considered to be short enough that it can spontaneously unbind from its complement $x\hat{}$*. In the remainder of this paper, we also refer to a process P as a program.

The semantics of the process calculus is summarised in Definition 2. We note that the rules can be applied to a collection of sequences in a given *context*, as shown in Definition 2. Individual domains within a given sequence are separated by white space, while multiple sequences are separated by a comma. Each rule is labelled with the name of the rule, together with the set of bonds that are modified by the rule. The semantic rules rely on a number of auxiliary functions in order to determine whether a give rule can be applied. Specifically, the function *adjacent*(i, P) returns the set of bonds that are adjacent to bond i in program P. The function *hidden*(i, P) returns true if one end of bond i occurs within a closed loop, and hence is prevented from efficiently binding to its complement. The function *anchored*(i, P) returns true if both ends of bond i are held “close” to each other, meaning that bond i is part of a stable junction. We note that, although these functions take an arbitrary process P as argument, they only examine a part of the context surrounding the bond i, as illustrated graphically in Definition 2. An alternative version of Definition 2 is presented in Appendix B, which allows multiple consecutive domains to be migrated simultaneously, in a single merged reaction. We illustrate the various reduction rules through a collection of example programs, described below.

Definition 1. Syntax of processes, in terms of *domain names x, y, z* and *bonds i, j, k*. We assume that all processes P are *well-formed* in that each bond i in P appears exactly twice and is shared between complementary domains. We consider processes *equal* up to re-ordering of strands. The function *toehold*(x) returns true if x is a toehold domain. For convenience, we also write $x\hat{}$ if *toehold*(x).

Domain $d ::=$	x	Domain name
$\mid$	x^*	Complementary domain name
Possibly bound domain $o ::=$	d	Free domain
$\mid$	$d!i$	Bound domain
Sequence $S ::=$	$o_1 \dots o_N$	Sequence of domains, $N \geq 1$
Process $P ::=$	$<S_1> \mid \dots \mid <S_N>$	Multiset of strands, $N \geq 0$

2.2 Examples

2.2.1 Hairpin toehold exchange—The first example (Fig. 1A) illustrates a hairpin toehold exchange, in which an invader strand opens up a hairpin formed by a template strand. For convenience, we assign names to the strands and provide a corresponding graphical representation, which shows bound domains connected by their bond names. We note that there is a one-to-one correspondence between the graphical representation and the program code.

```
invader = <tˆ x> | template = <x!j uˆ!k y uˆ*!k x*!j tˆ*>
```

We can first apply rule (RB), which allows the domain t on the invader to bind to the complementary domain t^* on the template. The program matches the context $C(t, t^*)$, which means that both the domain t and its complement t^* are not already bound. We now check whether a bond i can be formed between these two domains, written $P' = C(t!i, t^*!i)$, by ensuring that this bond is not hidden, written $\neg$*hidden*(i, P'). Currently a bond is defined as hidden if one end of the bond is inside a closed loop that does not contain the other end of the bond, which is not the case here. Note that this definition could be generalised to check for the presence of additional secondary structures that may prevent the bond from forming.

Since all of the conditions of rule (RB) are satisfied, we can form the new bond *i* between the invader and the template to produce the program *P′*:

```
            <t^ x>   | <x!j u^!k y u^*!k x*!j t^*>


    ->(RB) <t^!i x> | <x!j u^!k y u^*!k x*!j t^*!i>
```

We can now apply rule (RU), which allows the invader to unbind from the template by breaking the bond between domains *t* and *t**. The program matches the context $P = C(t!i, t^*!i)$, which means that there is a bond *i* between domains *t* and *t**. Furthermore, domain *t* represents a toehold since it is annotated with the toehold symbol (ˆ), which means that the domain is short enough that it is able to unbind spontaneously. We also check that the bond *i* is not anchored in the program, written ¬*anchored*(*i, P*). We define a bond as “anchored” if it is part of a junction that holds both ends of the bond close to each other, which is not the case here. Note that this definition could be generalised to check for the presence of other structures that may stabilise the bond. Since all of the conditions of rule (RU) are satisfied, we can break the bond *i* between the invader and the template to produce the program $C(t, t^*)$:

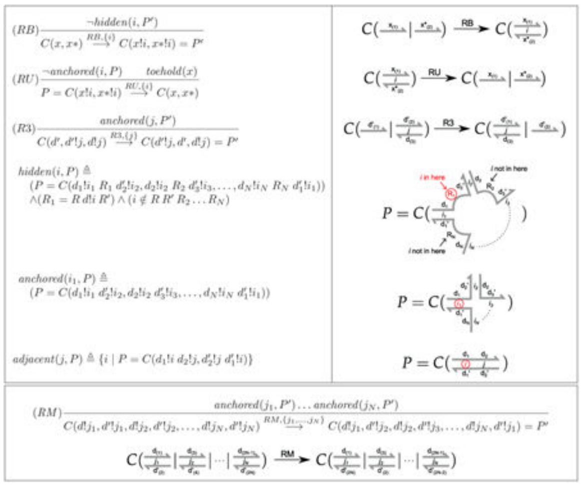
```
            <t^!i x> | <x!j u^!k y u^*!k x*!j t^*!i>


    ->(RU) <t^ x>   | <x!j u^!k y u^*!k x*!j t^*>
```

Definition 2. Reduction semantics of processes, with graphical representations to clarify the exposition. We define a *context* $C(S_1, \ldots, S_N)$ as a process *P* containing sequences $S_1, \ldots, S_N$, where $permute(S_1, \ldots, S_N)$ denotes any possible permutation of sequences $S_1, \ldots, S_N$:

$$\begin{array}{rlll} R ::= & o_1 \ldots o_N & & N \geq 0 \\ C(S_1, \ldots, S_N) ::= & split(permute(S_1, \ldots, S_N)) \mid P & & N \geq 1 \\ split(S_1, \ldots, S_N) ::= & <R\ S_1\ R_1 \ldots S_N\ R_N> & & \\ \mid & split\ (S_1, \ldots, S_i) \mid split(S_{i+1}, \ldots, S_N) & & i \in \{1, \ldots, N-1\} \end{array}$$

The function *comp*(*d*) returns the complement of domain *d*, where $comp(x) = x^*$ and $comp(x^*) = x$. The function *adjacent*(*j, P*) returns the set of bonds that are adjacent to *j* in *P*. The function *hidden*(*i, P*) returns true if one end of bond *i* is hidden within a closed loop. The function *anchored*(*i, P*) returns true if both ends of bond *i* are held close to each other, such that bond *i* is part of a junction.



To enable 3-way branch migration leaks, we relax the constraints on rule (R3) by replacing *anchored*(*j*, *P*′) with |*adjacent*(*j*, *P*)| ≤ 1, which simply checks whether domain *j* has at most one adjacent bond in *P*, where |*B*| denotes the number of elements in set *B*. Note that we use *P* rather than *P*′ to check for adjacent bonds, to ensure that the strand being displaced has at least one end free. Therefore we also require $P = C(d', d'!j, d!j)$ in this version of the rule and have no requirement on *P*′. We refer to this generalised version of rule (R3) as rule (RL). Similarly, to enable hairpin binding leaks we relax the constraints on rule (RB) by removing the condition ¬*hidden*(*i*, *P*′). We refer to this generalised version of rule (RB) as rule (RH).

This takes us back to our initial state, which means that we can re-apply rule (RB) to produce the program $C(t!i, t^*!i)$:

```
           <t^ x>   | <x!j u^!k y u^*!k x* !j t^*>


   ->(RB) <t^!i x> | <x!j u^!k y u^*!k x*!j t^*!i>
```

Now, instead of applying rule (RU) we can apply rule (R3), which allows the domain *x* of the invader to displace the domain *x* of the template, by a process of branch migration. The program matches the context $C(x, x!j, x^*!j)$, which means that there is a bond *j* between a domain *x* and its complement *x**, together with a third domain *x* that is not bound. We now check whether one end of the bond can be moved from the template to the invader, written $P' = C(x!j, x, x^*!j)$, such that the resulting bond is anchored. This is indeed the case, since there is a bond *i* that is immediately adjacent to *j* in *P*′, holding both ends of bond *j* close to each other. Since all of the conditions of rule (R3) are satisfied, we can move one end of bond *j* from the template to the invader, to produce the program *P*′:

```
           <t^!i x>   | <x!j u^!k y u^*!k x*!j t^*!i>


   ->(R3) <t^!i x!j> | <x u^!k y u^*!k x*!j t^*!i>
```

We can now apply rule (RU) again, which allows the hairpin to open. Unlike the previous unbinding reaction, this one results in a hairpin opening rather than in two strands separating completely from each other. The program matches the context $C(u!k, u^*!k)$, the bond *k* is not anchored and the domain *u* is a toehold. Therefore, we can break the bond *k* and open up the hairpin to produce the program *C*(*u, u**):

```
          <t^!i x!j> | x < u^!k y u^*!k x*!j t^*!i>


   ->(RU) <t^! x!j x> | <x u^ y u^* x*!j t^*!i>
```

We can now apply rule (RB) again, which closes the hairpin. Unlike the previous binding interaction, this one involves two domains that are on the same strand, rather than two domains on two different strands. As before, the program matches the context *C*(*u, u**) and the bond *k* that we wish to form is not hidden in *P′* = *C*(*u*!*k, u**!*k*), so the bond can form to close the hairpin and produce the program *P′*:

```
          <t^!i x!j> | <x u^ y u^* x*!j t^*!i>


   ->(RB) <t^!i x!j> | <x u^!k y u^*!k x*!j t^*!i>
```

2.2.2 Branch migration leak

—The second example (Fig. 1B) illustrates a branch migration leak, in which an invader strand opens up a hairpin formed by a template strand, in the absence of an exposed toehold. To enable branch migration leaks, we relax the constraints on rule (R3) so that the condition *anchored*(*j, P′*) is replaced with the condition |*adjacent*(*j, P*)| ≤ 1, which simply checks that there is at most one bond adjacent to the bond *j* that we wish to move, as described in Definition 2. We refer to this generalised version of rule (R3) as rule (RL).

```
   invader = <x> | template = <x!j y x*!j>
```

We can apply rule (RL), which allows the domain *x* of the invader to displace the domain *x* of the template and open the hairpin, by a process of branch migration leak. The program matches the context *C*(*x, x*!*j, x**!*j*) and we now check that there is at most one bond adjacent to the bond *j* that we wish to move. This ensures that at least one end of the bound strand is able to fray, allowing the invader strand to initiate a migration in the absence of a toehold. This is indeed the case, since there are no bonds adjacent to the bond *j* that we wish to move. Since all of the conditions of rule (R3) are satisfied, we can move one end of bond *j* from the template to the invader, to produce the program *C*(*x*!*j, x, x**!*j*):

```
          <x>   | <x!j y x*!j>


   ->(RL) <x!j> | <x y x*!j>
```

We can now apply rule (RL) again to close the hairpin. Unlike the previous branch migration leak, this one involves two domains that are on the same strand. As before, the program matches the context *C*(*x, x*!*j, x**!*j*) and there are no bonds adjacent to the bond *j* that we wish to move. Therefore, we can perform the branch migration leak and close the hairpin, to produce the program *C*(*x*!*j, x, x**!*j*):

```
         <x!j>|<x y x*!j>
```

```
->(RL) <x>   | <x!j y x*!j>
```

2.2.3 Four-way branch migration—The third example illustrates a 4-way branch migration (Fig. 1C), in which two bonds are exchanged between two pairs of complementary domains.

```
<x!i y*!j1> | <y!j1 z*!k> | <z!k y*!j2> | <y!j2 x*!i>
```

The program matches the context $C(y!j_1, y^*!j_1, y!j_2, y^*!j_2)$, which means that there are two pairs of complementary domains y and y^*, with the first pair joined by bond j_1 and the second pair joined by bond j_2. Note that this matches rule (RM) for $N = 2$, where N denotes the number of pairs of complementary domains. We now check whether the bonds j_1 and j_2 can be moved so that they are between complementary domains from different pairs, written $P' = C(y!j_1, y^*! j_2, y!j_2, y^*!j_1)$, by checking that the bonds we wish to form are both anchored in P'. This is indeed the case, since there is a bond i that is adjacent to bond j_2 and a bond k that is adjacent to bond j_1 in P'. Since all of the conditions of rule (RM) are satisfied, we can simultaneously exchange one end of bond j_2 with one end of bond j_1, to produce the program P':

```
        <x!i y*!j1> | <y!j1 z*!k> | <z!k y*!j2> | <y!j2 x*!i>

->(RM) <x!i y*!j2> | <y!j1 z*!k> | <z!k y*!j1> | <y!j2 x*!i>
```

2.2.4 Three-way initiated four-way branch migration—The final example illustrates a 3-way initiated 4-way branch migration (Fig. 1D), as described in [36]. The transitions between the first and second states are standard toehold binding and unbinding reductions, according to rules (RB) and (RU). Once the $T_1\hat{}$ toehold has bound, rule (R3) can be used to derive a branch migration across the A domain. This reduction illustrates the use of the *anchored* predicate to deduce that the bonds i_1, i_2, j_1, j_2 together form a junction that anchors the j_2 bond in place after the strand displacement reduction has occurred. Note that this reduction could not be derived using the previously published DSD semantics, which required the $T_1\hat{}$ and A domains to be on the same invader strand. Once the four-way junction has formed, neither the i_3 bond nor the j_2 bond can spontaneously unbind according to rule (RU), since each of the bonds is anchored in place by the rest of the junction consisting of i_1, i_2, j_1, j_2. Hence, the only possible reduction is the four-way branch migration that exchanges bonds j_1 and i_2 between the two pairs of bound domains X and X^*. This reduction is derived using rule (RM), where the newly formed bond j_1 is anchored by the adjacent bond i_3, and the newly formed bond i_1 is anchored by the adjacent bond j_2. This reduction could also not be derived using the previously published DSD semantics, that do not allow four-way junctions or four-way branch migration reactions. Finally, the only remaining reductions are the reversible toehold unbinding and binding reductions on the $T_2\hat{}$ domain, according to rules (RU) and (RB).

2.3 Chemical reaction network encoding

The reduction semantics of the previous section considers which reductions are *possible*. This enables qualitative analysis such as computing the state space or state reachability of a system, verifying input/output behaviour, or checking for behavioural equivalence. However, for quantitative analysis such as stochastic and deterministic simulation or probabilistic model checking, we must also consider the *rate* of each reduction. This requires quantifying all of the possible ways in which a given reduction can take place. To achieve this, we follow the approach of [21], by translating processes of the calculus to chemical reaction networks that can then be simulated.

Briefly, a *chemical reaction network* (CRN) is defined as a pair (X, R), where X is a finite set whose elements are referred to as *species*, and R is a finite set of *reactions* of the form $(\mathbf{R}, r, \mathbf{P})$, with $\mathbf{R}, \mathbf{P} \in \mathbb{N}^X$ and $r \in \mathbb{R}$. We let $\mathbb{N}^X$ denote the set of multisets of X and $\mathbb{R}$ denote the set of real numbers, and we use bold font to denote a multiset. For a reaction $(\mathbf{R}, r, \mathbf{P})$ the multiset $\mathbf{R}$ denotes the *reactants*, the multiset $\mathbf{P}$ denotes the *products* and the constant r denotes the *rate* at which the reaction takes place. This can also be written as $\mathbf{R} \xrightarrow{r} \mathbf{P}$. We write a multiset of species as $[I_1, \ldots, I_N]$, where I_i is a species. This can also be written as $I_1 + \ldots + I_N$. We also define a *state* as a multiset of species, where a reaction is *applied* to a state by removing the reactants from the state and adding the products. An example CRN is the following, which corresponds to Fig. 2A:

$$\{I, A, IA, B, IAB, C, IABC, ABC\}, \{I+A \xrightarrow{r_1} IA, IA+B \xrightarrow{r_2} IAB, IAB+C \xrightarrow{r_3} IABC, IABC \xrightarrow{r_4} ABC+I\}$$

The first three reactions are *bimolecular* in that they each have two reactants, while the fourth reaction is *unimolecular*, with a single reactant. The rate of a bimolecular reaction scales with the product of the populations of the two species, since the species are assumed to interact when they encounter each other by a process of diffusion.

Definition 3. Encoding a calculus process to a chemical reaction network. We define a *species* as a process that is a *connected component*, such that any two strands in the process are connected to each other by a path of shared bonds. The function *complex*(P) returns true if P is a connected component. The function *species* (P) returns the multiset of species in P. To enable a concise representation of species, the function *populations*$([I_1, \ldots, I_N])$ converts a multiset of species expressed as a list, to an equivalent multiset expressed as a list of pairs, where the first element of the pair denotes the species and the second element denotes its population. For this we use a notion of *process equivalence* $(\equiv)$ to equate species, where processes are equivalent up to global renaming of bonds. The functions $rate_B\,(R, B, P, P')$ and $rate_U\,(R, B, P, P')$ compute the bimolecular and unimolecular rate constants of a reaction, respectively. The function *freshen*(P) produces a process $P' \equiv P$ in which all bonds are renamed using globally unique names.

$$\text{species}(P) \triangleq [P_1,\dots,P_N] \text{ if } P=(P_1|\dots|P_N) \wedge \text{complex}(P_1) \wedge \dots \wedge \text{complex}(P_N)$$
$$\text{reactions}(I,\{I_1,\dots,I_N\}) \triangleq \text{unary}(I) \cup \text{binary}(I,\text{freshen}(I)) \cup \text{binary}(I,I_1) \cup \dots \cup \text{binary}(I,I_N)$$
$$\text{unary}(I) \triangleq \{([I],r,\mathbf{P}) | [I] \xrightarrow{r} \mathbf{P}\}$$
$$\text{binary}(I_1,I_2) \triangleq \{([I_1,I_2],r,\mathbf{P}) | [I_1,I_2] \xrightarrow{r} \mathbf{P}\}$$
$$\text{populations}(\mathbf{P}) \triangleq \{(P_{i1},m_i) | \mathbf{P}=[P_{11},\dots,P_{1m_1},\dots,P_{N1},\dots,P_{Nm_N}]; i,j \in \{1,\dots,N\}; P_{i1} \equiv \dots \equiv P_{im_i}; i \neq j \Rightarrow P_{i1} \not\equiv P_{j1}\}$$
$$(UR)\frac{P \xrightarrow{R,B} P' \quad \text{complex}(P)}{[P] \xrightarrow{\text{rate}_U(R,B,P,P')} \text{species}(P')}$$
$$(BR)\frac{P_1|P_2 \xrightarrow{R,\{i\}} (C_1(d!i)|C_2(d'!i))=P' \quad \text{complex}(P_1) \quad \text{complex}(P_2) \quad P_1=C_1(d)}{[P_1,P_2] \xrightarrow{\text{rate}_B(R,\{i\},P_1,P_2,P')} \text{species}(P')}$$

In contrast, the rate of a unimolecular reaction scales with the population of the single species. This distinction is essential when computing a CRN from a DSD process, and is achieved by checking whether or not two interacting strands belong to the same connected component.

Definition 3 presents an encoding from the DSD calculus to a chemical reaction network. The function *species*(P) computes the multiset of species in process P, while the function *reactions*(I, X) computes the set of reactions between the species I and each of the species in X. As in [24, 21], we only consider unimolecular and bimolecular reactions, defined by rules (UR) and (BR), respectively. Note that we need to compute the multiset of species after each reduction, since the connected components may change as a result of new bonds being formed or broken. As in [21], the *reactions* function can be used to generate a CRN in *saturating* or *just-in-time* modes. In saturating mode, we assume a fixed set of species X and we compute *reactions*(I, ($X \setminus I$)) for all $I \in X$, where $X \setminus I$ denotes the set X with the element I removed. Thus, the entire set of possible reactions is computed in one go. In just-in-time mode, the possible reactions involving only the initial species are computed, and then the remaining reactions are computed dynamically during a stochastic simulation, by expanding the reaction set whenever a new species is formed as the result of executing a reaction. This allows simulation of systems with potentially unbounded numbers of species, such as polymer chains, and in general can allow more efficient simulation of systems with large but bounded numbers of species. We refer the reader to [21] for additional details of just-in-time simulation.

The functions $rate_B(R, B, P_1, P_2, P')$ and $rate_U(R, B, P, P')$ in Definition 3 compute the bimolecular and unimolecular rate constants of a reaction, respectively. The specific definitions of these functions will depend on the choice of a suitable rate model. The complexity of the rate model is likely to depend on the modelling problem at hand. Rate models may range from a collection of default rates, to an approach in which all rates are expressed as free parameters and fitted from experimental data, to a biophysics-based approach for estimating rate constants. Here we propose an interim rate model, in which the functions $rate_B(R, B, P_1, P_2, P')$ and $rate_U(R, B, P, P')$ use the domain d on which the bonds B are formed or broken, together with the type of reduction R, to determine a rate constant given by *rate*(R, d), similar to the approach of [24]. Specifically, *rate*(RB, d) and *rate*(RU, d) return the rates of binding and unbinding, respectively, involving a domain d and its complement, while *rate*(*R3, d*), *rate*(*RM, d*) and *rate*(*RL, d*) return the rates of branch migration, *n*-way branch migration, and leakage, respectively, involving a domain *d* and its complement. The function $rate_B$ directly returns the bimolecular rate constant based on the type of interaction and the domain sequences, while the function $rate_U$ computes the unimolecular rate constant of two interacting strands that are part of the same molecular complex, by following the approach of [12, 22]. Briefly, for interacting strands that are part of the same complex, this scales all bimolecular rates by the *local concentration* of the interacting domains, given by *conc*(*B,P,P′*), to yield a unimolecular rate. We note that the set of bonds *B* is used to pinpoint a specific interaction, which is required in cases where more than one interaction on a given domain is possible. In our current approach, a default local concentration is used. However, this approach can be generalised to make use of biophysical models based on both the sequence and structure of the interacting DNA complexes, such as those proposed by [12], for accurately computing the local concentration.

We consider the reductions in Fig. 1A to illustrate our interim rate model. The first binding reduction (RB) denotes a bimolecular reaction of the invader binding to the template, which takes place at rate *rate*(*RB, t*). However, the second binding reduction (RB) denotes a unimolecular reaction, which takes place at rate *rate*(*RB, u*) × *c*, where *c* is the local concentration of the interacting domains *u* and *u**. This can be computed using a biophysical model such as in [12]. For the reductions in Fig. 1B, the first leak reduction (RL) denotes a bimolecular reaction of the invader binding to the template in the absence of a toehold, which takes place at rate *rate*(*RL, x*). This rate is typically $1M^{-1}s^{-1}$. However, the second leak reduction (RL) denotes a unimolecular *remote toehold* leak reaction between the template and the bound invader, which takes place at *rate*(*RL, x*) × *c*, where *c* is the local concentration of the interacting domains. As the local concentration approaches 1M, or about 1 molecule per nm^3, the leak reaction rate is scaled to about $1s^{-1}$, which approaches the rate of a normal branch migration reaction when the migrating bond is anchored.

We now make use of the CRN semantics to generate the chemical reaction networks for a number of example systems based on programmed assembly of metastable hairpins to form a variety of structures [45]. Note that in the experimental sequences that were used to implement these systems, almost all individual domains were 6 nucleotides in length, which corresponds to a toehold domain. Therefore, to reduce the notational burden associated with explicitly labelling almost all domains as toeholds, we employ the notational convention that domains with lower-case names are toeholds whereas domains with upper-case names are long domains.

Our first example is the catalytic self-assembly of a three-arm branched junction, as illustrated in Figure 2 of [45], where the computed reactions are summarized in Fig. 2A. Our second example is the cross-catalytic signal amplification circuit from Figure 3 of [45], where the computed reactions are summarized in Fig. 2B. Our third example is the stochastic bipedal walker example from Figure 5 of [45], where the computed reactions are summarized in Fig. 3. Note that for this example we have merged some of the reactions for conciseness, and have also omitted the reverse reactions. Code in the process calculus syntax for the initial species involved in these examples is presented below the corresponding graphical representation.

3 Strand graphs

The process calculus defined in the previous section represents a formal programming abstraction for strand displacement systems with complex secondary structures. While the syntax of the calculus provides a convenient way of writing down arbitrary secondary structures composed of DNA strands, in practice the semantic rules would be challenging to implement directly, because of the complexity of pattern-matching on arbitrary process contexts that is required by the rules. In this section we provide a mapping from the process calculus to a canonical representation based on strand graphs. This enables us to leverage the expressive power of graph theory both to represent secondary structures and to derive reactions via localized graph rewriting rules. Our rewrite rules are local in the sense that they only have to consider a bounded number of connected components.

3.1 Notation

We summarise the definition for strand graphs in Definition 4. Each vertex v of the graph represents a *strand*, and each site s at a vertex represents a *domain*. The sites at a given vertex are ordered according to the sequence of domains in the strand, given the strand's innate directionality. This vertex-specific ordering on sites is a key difference between strand graphs and site graphs. An edge e between two sites represents a *bond* between two domains. We can record the fact that certain domains are complementary and thus able to bind to each other by a set of *admissible* edges (A); all edges that appear throughout the execution of the program will be drawn from this set. Some of these edges denote bindings between toeholds, while the remainder denote bindings between long domains. This information (recorded by the function *toehold*) is needed in order to determine whether edges can be removed by spontaneous unbinding of domains, which can only happen if the bound domains are toeholds. At this point, we can omit information about the specific domains on the strands. All of the information needed to manipulate connectivity is available from the placement of sites on nodes, the set of admissible edges and the annotation of toehold edges. The set E models the current edges in the program and is the only non-static information. The reduction rules change one strand graph into another by changing only this set of current edges. We therefore consider the rest of the information static and only describe the transformations of E. In order to report which nodes correspond to which strands to the user, we assign a colour to each vertex (via the function *colour*). Thus, different copies of the same strand are modelled by different vertices with the same colour. The domains can of course also be included on the strands for display purposes. The colour information also simplifies the definitions and is helpful when introducing the CRN semantics in Section 3.4.

We may draw a strand graph as in Fig. 4B. Each strand type is assigned a colour, for instance the strand type `<L T2^ X* T1^>` has been assigned the colour green (represented by the number 1). The sites are marked as small black circles. To illustrate the sequence of the sites, the nodes have been drawn as circular arrows (with the arrowhead at the 3′ end) rather than circles. Thus, on the green node (node 1), the first site L is the lower left one. We have drawn all the admissible edges; the current edges are drawn in black while the rest are drawn in blue. Further, toehold edges are drawn as dashed.

We define an encoding from a process to a strand graph in Definition 5. The assignment of colours to strand types is made arbitrarily, which means that this assignment should be fixed if different programs are to interact. For the moment we only consider the case where the entire system is described by a single program. Note that *toehold* only has to look at one end of the edge, since its domain, A, is constructed such that edges will either have short domains at both ends or at neither end.

Definition 4. Strand graphs. For $V \subseteq \mathbb{N}$ and $length : V \to \mathbb{N}$, we define the following helper functions:

$legal_S(V, length) = \{(v, n) \in \mathbb{N}^2 \mid v \in V \wedge n \leq length(v)\}$ to be the set of legal sites, where a site $s = (v, n)$ is a pair of natural numbers such that v is a vertex and n is a position on that vertex.

$legal_E(V, length) = \{S \subseteq legal_S(V, length) \mid |S| = 2\}$ is the set of legal edges, where an edge $e = \{s_1, s_2\}$ is a set containing two sites, such that $s_1 \neq s_2$.

We define a *strand graph* $G = (V, length, colour, A, toehold, E)$, where:

$V = \{1, \ldots, N\}$ is the set of vertices; each vertex is a natural number.

$colour : V \to \mathbb{N}$ is a function assigning a colour to each vertex, colours are also natural numbers.

$length : V \to \mathbb{N}$ is a function assigning a length to each vertex, such that $colour(v_1) = colour(v_2) \Rightarrow length(v_1) = length(v_2)$. That is, the length is really a function of the colour.

$A \subseteq legal_E(V, length)$ is a set of admissible edges, such that $colour(e_1) = colour(e_2) \Rightarrow (e_1 \in A \Leftrightarrow e_2 \in A)$ and $(legal_S(V, length), A)$ forms a bipartite graph, which ensures that admissible edges connect complementary domains. For an edge $e = \{(v_1, n_1), (v_2, n_2)\}$ we define $colour(e) = \{(colour(v_1), n_1), (colour(v_2), n_2)\}$.

$toehold : A \to \mathbb{B}$ is a function that returns true on all admissible edges between toehold domains and false on admissible edges between long domains, such that $colour(a_1) = colour(a_2) \Rightarrow toehold(a_1) = toehold(a_2)$.

$E = \{e_1, \ldots, e_M\} \subseteq A$ is a set of current edges, such that $(i \neq j) \Rightarrow e_i \cap e_j = \emptyset$. That is, at any given instant there can be at most one edge connected to any given site.

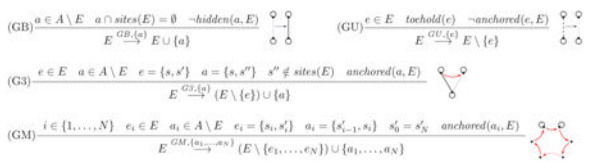
Definition 5. Given a program $P = <S_1> \mid \ldots \mid <S_N>$, we will define a corresponding strand graph $S = (V, length, colour, A, toehold, E)$. For a domain o, we define its type $tp(o)$ by $tp(d) = d$ and $tp(d!i) = d$, that is tp erases all bonds. Given a strand $S = o_1 \ldots o_n$, we define the type to be $tp(S) = tp(o_1) \ldots tp(o_n)$ and the length to be $len(S) = n$. We now number the strand types in order of appearance in the program, obtaining a list $t_1 \ldots t_T$, from which we can define the colour function. We also define domain functions *dom* and *ndom* and a toehold predicate *toe*, all specific to P, such that $dom(i, j)$ is the jth domain in S_i, $ndom(i, j)$ is the name of $dom(i, j)$ (i.e. with any bindings removed) and $toe(i, j)$ is true exactly if $ndom(i, j)$ is a toehold domain. We are now ready to define the strand graph S:

$$V=\{,\ldots,N\}$$
$$\text{length}(v)=\text{len}(S_v)$$
$$\text{colour}(v)=i \iff t_p(S_v)=t_i$$
$$(v_1,n_1)\overset{A}{\longleftrightarrow}(v_2,n_2) \iff \text{ndom}(v_1,n_1)=\text{comp}(\text{ndom}(v_2,n_2))$$
$$\text{toehold}(\{s_1,s_2\}) \iff \text{toe}(s_1)$$
$$(v_1,n_1)\overset{E}{\longleftrightarrow}(v_2,n_2) \iff \exists d,i.\text{dom}(v_1,n_1)=d!i\hat{}\text{dom}(v_2,n_2)=\text{comp}(d)!i$$

3.2 Reduction semantics

The reduction semantics of strand graphs is given in Definition 6. We briefly provide some intuition for the reduction rules below. The rules can then be seen to simply change the colour of the edges. Rule (GB) states that if there is an edge that is admissible but not current, with both end sites unoccupied and not hidden from each other, then this edge can be added to the current set. This reflects that the two domains at the end sites are able to bind. Rule (GU) states that if there is a current edge that denotes a toehold and is not anchored, then it can be removed from the current set. This is because the toehold is free to unbind. To describe rules (G3) and (GM) we introduce the concept of a *displacing path*. In particular, an admissible edge can become a current edge even if one of the end sites is taken, by removing the edge that is in the way. That is we will make the following transformation:

Definition 6. Semantics of strand graphs, assuming a global finite set V of vertices and a global set A of admissible edges. The function *sites*(E) = $\{s \mid \exists\ e \in E.\ s \in e\}$ returns the set of sites in E. The function *adjacent*(e, E) returns the set of edges in E that are adjacent to edge e. Two edges are adjacent if they occur between the same pair of vertices and each site in one edge is adjacent to a site in the other edge. Since paired strands run in opposite directions, the end sites are shifted in opposite directions. The function *hidden*(e, E) returns true if one end of edge e is hidden within a closed loop. The predicate *anchored*(e, E) holds only if edge e occurs between two sites that are held close together, such that e is part of a junction. The reactions are annotated with the set of bonds they form or break.



$$\text{(GB)}\ \frac{a \in A\setminus E \quad a \cap sites(E)=\emptyset \quad \neg hidden(a,E)}{E \xrightarrow{GB,\{a\}} E \cup \{a\}} \qquad \text{(GU)}\ \frac{e \in E \quad toehold(e) \quad \neg anchored(e,E)}{E \xrightarrow{GU,\{e\}} E\setminus\{e\}}$$

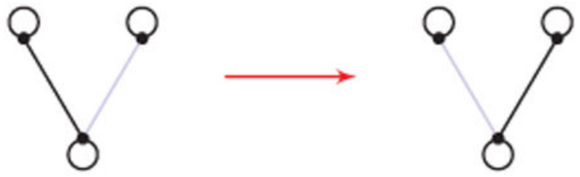
$$\text{(G3)}\ \frac{e \in E \quad a \in A\setminus E \quad e=\{s,s'\} \quad a=\{s,s''\} \quad s'' \notin sites(E) \quad anchored(a,E)}{E \xrightarrow{G3,\{a\}} (E\setminus\{e\})\cup\{a\}}$$

$$\text{(GM)}\ \frac{i \in \{1,\ldots,N\} \quad e_i \in E \quad a_i \in A\setminus E \quad e_i=\{s_i,s_i'\} \quad a_i=\{s_{i-1}',s_i\} \quad s_0'=s_N' \quad anchored(a_i,E)}{E \xrightarrow{GM,\{a_1,\ldots,a_N\}} (E\setminus\{e_1,\ldots,e_N\})\cup\{a_1,\ldots,a_N\}}$$

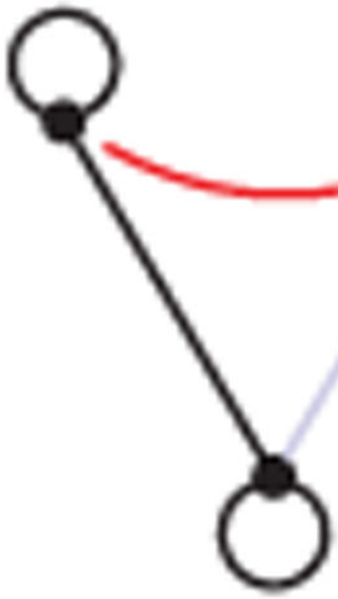
$$\text{adjacent}(\{(v_1,n_1),(v_2,n_2)\},E) \triangleq E \cap \{\{(v_1,n_1-1),(v_2,n_2+1)\},\{(v_1,n_1+1),(v_2,n_2-1)\}$$
$$\text{hidden}(\{(v_1,n),(v',n')\},E) \triangleq \forall i \in \{1,\ldots,N\}.e_i \in E\hat{}e_i=\{(v_i,n_i),(v_i',n_i')\}\hat{}(v_{i+1}'=v_i)\hat{}(n_{i+1}'>n_i)\hat{}(v_1'=v_N)\hat{}(n_1'>n_N)\hat{}(n_1<n<$$
$$\text{anchored}(e_1,E) \triangleq \forall i \in \{1,\ldots,N\}.e_i \in (\{e_1\}\cup E)\hat{}e_i=\{(v_i,n_i),(v_i',n_i')\}\hat{}(v_{i+1}',n_{i+1}')=(v_i,n_i+1)\hat{}(v_1',n_1'$$

As in Definition 2, to enable 3-way branch migration leaks, we relax the constraints on rule (G3) by replacing *anchored*(a, E) with |*adjacent*(e, E) | ≤ 1, which simply checks whether domain e has at most one adjacent edge in E. We refer to this generalised version of rule (G3) as rule (GL). Similarly, to enable hairpin binding leaks we relax the constraints on rule

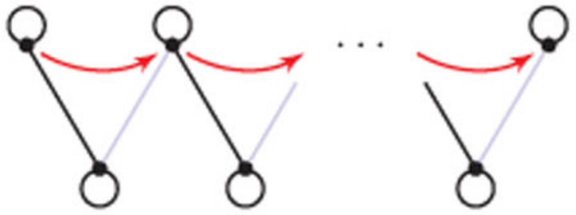
(GB) by removing the condition ¬*hidden*(*a, E*). We refer to this generalised version of rule (GB) as rule (GH).



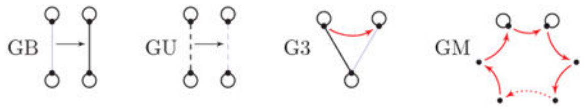
However the transformation will only occur if the *anchored* predicate is true. In that case, we may depict the situation as follows:



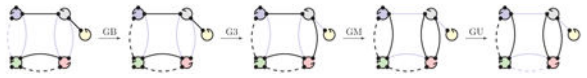
This illustrates that the bond can be seen to swing, rather than one edge disappearing and another appearing. We can then imagine that if the site to which the bond would swing is already occupied then another bond swing would need to happen first. This could form a long chain as follows:



The red arrows then form what we term a displacing path. If such a path forms a loop, it denotes the simultaneous swapping of a list of bonds. If it does not form a loop, it can be unravelled from the end into a series of single bond swaps. The latter situation is captured by rule (G3) and the former by rule (GM). We can summarize the four main rules graphically as follows (also set next to the rules in Definition 6):



We can also illustrate the example of Fig. 1D as follows:



3.3 Correspondence with process calculus semantics

In this section we sketch a proof that the operational semantics of the process calculus and strand graph representations coincide. We will write E_P for the edge graph that was derived from the process P. We begin by stating some lemmas that are straightforward to prove, hence their proofs are omitted.

Lemma 7. $adjacent(i, P) = adjacent(\{(v, n),(v', n')\}, E_P)$, where there exists d such that $dom(v, n) = d!i$ and $dom(v', n') = comp(d)!i$.

Lemma 8. $hidden(i, P) \Leftrightarrow hidden(\{(v, n),(v', n')\}, E_P)$, where there exists d such that $dom(v, n) = d!i$ and $dom(v', n') = comp(d)!i$.

Lemma 9. $anchored(i, P) \Leftrightarrow anchored(\{(v, n), (v', n')\}, E_P)$, where there exists d such that $dom(v, n) = d!i$ and $dom(v', n') = comp(d)!i$.

Lemma 10. $anchored(a, E) \Leftrightarrow anchored(a, E \setminus \{e\}) \Leftrightarrow anchored(a, E \bigcup \{e'\})$, provided e and e' are not among the bonds forming the anchor.

We can now present a proof of the main correspondence theorem.

Theorem 11. There exist R and B such that $P \xrightarrow{R,B} P'$ iff there exists G and E such that $E_P \xrightarrow{G,E} E_{P'}$.

Proof. We first note that the effects of the rules are disjoint, that is for given P, P' only one rule could apply to tranform P into P', likewise for given E_P and $E_{P'}$. This can be seen by observing the changes of bonds: For $X \in R, G$, rules XU will remove one bond, rules XB will add one bond, rules $X3$ will move a single bond and rules XM will move more than one bond. (Rule GM cannot be instantiated with $N = 1$ as this would make $e_1 = a_1$, contradicting that $e_1 \in E$ and $a_1 \in A \setminus E$). It follows that the theorem can be strengthened to conclude that $R = RY$ and $G = GY$, for some $Y \in U, B, 3, M$. We will proceed by cases on Y.

- **Case RB,GB.** There exists an i such that $P = C(x, x^*)$ and $P' = C(x!i, x^*!\ i)$ iff there exists an $a \in A \setminus E_P$ such that $a \bigcap sites\,(E_P) = \emptyset$ and $E_{P'} = E_p \bigcup \{a\}$. In that case, $a = \{s, s'\}$, where $dom(s) = x!i$ and $dom(s') = x^*!i$, and by Lemma 8 we get that

$hidden(i, P') \Leftrightarrow hidden(a, E_{P'})$, i.e., that $\neg hidden(i, P') \Leftrightarrow \neg hidden(a, E_{P'})$. Thus, for $B = \{i\}$ and $E = \{a\}$, we get that $P \xrightarrow{RB,B} P'$ iff $E_P \xrightarrow{GB,E} E_{P'}$.

- **Case RU,GU.** There exists an i such that $P = C(x!i, x^*!i)$ and $P' = C(x, x^*)$ iff there exists an $e \in E_P$ such that $E_{P'} = E_P \setminus \{e\}$. In that case, $e = \{s, s'\}$, where $dom(s) = x!i$ and $dom(s') = x^*!i$, and soby Lemma 9 we get that $\neg anchored(i, P) \Leftrightarrow \neg anchored(e, E_P)$, and by definition we get that $toehold(x) \Leftrightarrow toehold(e)$. Thus, for $B = \{i\}$ and $E = \{e\}$, we get that $P \xrightarrow{RU,B} P'$ iff $E_P \xrightarrow{GU,E} E_{P'}$.
- **Case R3,G3.** There exists j such that $P = C(d', d'!j, d!j)$ and $P' = C(d'!j, d', d!j)$ iff there exists $e \in E_P$ and $a \in A \setminus E_P$ such that $e = \{s, s'\}$ and $a = \{s, s''\}$ and $s'' \notin sites(E_P)$ and $E_{P'} = (E_P \setminus \{e\}) \bigcup \{a\}$. In that case, $dom(s) = d!j$ and $dom(s') = d'!j$, and so by Lemmas 9 and 10 we get that $anchored(j, P) \Leftrightarrow anchored\,(a, E_{P'}) \Leftrightarrow anchored\,(a, E_P)$. Thus, for $B = \{j\}$ and $E = \{a\}$, we get that $P \xrightarrow{R3,B} P'$ iff $E_P \xrightarrow{G3,E} E_{P'}$.
- **Case RM,GM.** There exists $j, \ldots, j_N$ such that $P = C(d!j_1, d'!j_1, \ldots, d!j_N, d'!j_N)$ and $P' = C(d!j_1, d'!j_2, \ldots, d!j_N, d'!j_{N+1})$ iff there exists $e_1, \ldots, e_N \in E_P$ and $a_1, \ldots, a_N \in A \setminus E_P$ such that for all $k \in \{1, \ldots, N\}$, $e_k = \{s_k, s_k'\}$ and $a_k = \{s_{k-1}', s_k\}$, with $s_0' = s_N'$ and such that $E_{P'} = (E_P \setminus \{e_1, \ldots, e_N\}) \bigcup \{a_1, \ldots, a_N\}$. It is then the case that, for all $k \in \{1, \ldots, N\}$, $dom(s_k) = d^*!j_k$ and $\mathrm{dom}(s_k') = d!j_k$. By Lemmas 9 and 10 we get that $anchored(j_k, P') \Leftrightarrow anchored(a_k, E_{P'}) \Leftrightarrow anchored(a_k, E_P)$ for all $k \in \{1, \ldots, N\}$. Thus, for $B = \{j_1, \ldots, j_N\}$ and $E = \{a_1, \ldots, a_N\}$, we get that $P \xrightarrow{RM,B} P'$ iff $E_P \xrightarrow{GM,E} E_{P'}$.
- **Case leaks.** The cases for the leak reaction rules are similar to those above, with the only extra requirement being that we must use Lemma 7 to show that $|adjacent(j, P)| = |adjacent(e, E_P)|$, where e is defined as in the case for rule R3 above. From this, it is trivial to see that $|adjacent(j, P)| \le 1 \Leftrightarrow |adjacent(e, E_P)| \le 1$, as required.

This covered all cases, and therefore completes the proof of Theorem 11.

3.4 CRN semantics

The semantics of Definition 6 can be used to derive the behaviour of a full system, but represents each copy of a species explicitly. We would like to represent each species only once and record its multiplicity during simulation or state space exploration. A species in a strand graph is defined as a *connected component*, that is, a maximal connected sub graph. It is therefore tempting to define a species simply to be a connected strand graph. However, the interaction of species relies on them sharing the notions of admissible edges as well as toeholds, so we shall define species relative to this data structure. First we note that both notions factor through the colour map, meaning that to determine if an edge is admissible or is a toehold, it is enough to know the colour of the vertices at the endpoints (and the sites). Thus, we shall define the notion of an environment $\xi = (C, length, A, toehold)$ to be a tuple where $C = \{1, \ldots, K\}$ is a set of colours, $length : C \to \mathbb{N}$, $A \subseteq legal_E(C, length)$, and *toehold* : $A \to \mathbb{B}$. This is much like for strand graphs, except that everything is now given in colour space. Given an environment ξ, a *species on* ξ, $T = (V, colour, E)$ is then a tuple where $V = \{1, \ldots, N\}$, $colour : V \to C$, $colour(E) \subseteq A$, such that $sg(\xi, T)$ is a connected strand graph. So to decide admissibility of an edge, we first translate it into colour space and then ask the environment. The map *sg* turns an environment and a species on that environment into a strand graph, and is given by $sg(\xi, S) = (V', length', colour', A', toehold', E')$, where

$$\begin{gathered} V'=V \\ \text{length}'=\text{length}\circ\text{colour} \\ \text{colour}'=\text{colour} \\ A'=\{e\in\text{legal}_E(V',\text{length}')|\text{colour}'(e)\in A\} \\ \text{toehold}'=\text{toehold}\circ\text{colour} \\ E'=E \end{gathered}$$

We will write $adm_\xi(T)$ to denote A'. We say that two species, T_1 and T_2, on the same environment ξ are isomorphic if $sg(\xi, T_1) \sim sg(\xi, T_2)$, that is they are isomorphic as strand graphs as per Definition 13. We will write this as $T_1 \sim_\xi T_2$.

We need one more thing for our species semantics, namely the ability to put two species next to each other and consider the resulting strand graph to determine reactions. Since all species have vertices numbered from 1, we need to renumber one of the species. Thus we define a binary operator $|_\xi$ on species by

$$(\{1,\ldots,N\},\text{colour}_1,E_1)|_\xi(\{1,\ldots,M\},\text{colour}_2,E_2)=\text{sg}(\{1,\ldots,N+M\},\text{colour},E)$$

where

$$\text{colour}(n)=\begin{cases}\text{colour}_1(n) & n\le N\\ \text{colour}_2(n-N) & \text{otherwise}\end{cases}$$

and $E = E_1 \bigcup (E_2 + N)$. Here $E_2 + N$ means the set of edges obtained by adding N to all vertices of all the sites in E_2. Note that we allow *sg* to be applied to a graph which is not connected. Finally, we need a function *species* in the other direction, taking a strand graph and returning the connected components restructured into species via a similar renumbering. This will be a multiset rather than a set, since different connected components might be isomorphic. We can now define a CRN semantics, as shown in Definition 12. As with Definition 3, this semantics is parametrised by a rate model, which is used to compute unimolecular and bimolecular rate constants. We adopt a similar interim rate model to the one proposed for the process calculus in Section 2.

4 Implementation

Our implementation takes a given program, transforms it into a strand graph, splits this into connected components, brings each connected component into a canonical form, and then proceeds to enumerate reactions, compute the state space and/or do a just-in-time simulation of the system. We will describe the canonical form below, but first we describe the data structures for representing strand graphs.

4.1 Data structures

Since the vertices in a species are numbered 1 through N, and the colour function maps vertices to natural numbers, both of these data can be stored as a single array of length N of integers (in the implementation, we note that everything is numbered from 0 rather than from 1). The edges of a strand graph are stored as a set of pairs of sites. To avoid complications with storing the same edge in two different ways, these pairs are always stored in canonical form, such that the smallest site (by the lexicographic order on integers) is stored in the first component. Similar conventions are used to store the environment. As an optimisation, additional data structures are computed once during species normalisation. These include data structures for storing the anchored bonds, the set of bound ports and the admissible edges (not in colour space).

4.2 Species isomorphisms

Given the split of information into global and local, a permutation on a strand graph can be described as a permutation of the vertices which preserves the colours and does not change the set of edges. In fact, this is enough to determine isomorphism. That is, we have the following lemma:

Lemma 14. For a connected strand graph $S = sg(\xi, T)$, where $T = (V, colour, E)$, π is a strand graph automorphism on S iff π is a permutation of V such that $colour \circ \pi = colour$ and $\forall v_1, n_1, v_2, n_2.\ (v_1, n_1) \overset{E}{\longleftrightarrow} (v_2, n_2) \iff (\pi\ v_1, n_1) \overset{E}{\longleftrightarrow} (\pi\ v_2, n_2)$.

Definition 12. CRN semantics of strand graphs. The function $species(S)$ returns the multiset $\{(T_1, m_1), \ldots, (T_N, m_N)\}$ of components in S, where m_i denotes the multiplicity of component T_i in S. Importantly, we use a notion of *strand graph isomorphism* ($\sim$) to compare components, as shown in Definition 13. As in Definition 3, the functions $rate_U(R,E,C)$ and $rate_B(R,E,C)$ compute the unimolecular and bimolecular rate constants, respectively, associated with the reduction of type R involving edges E of the multiset C.

$$(UR)\,\frac{sg(T) \xrightarrow{R,E} S}{T \xrightarrow{rate_U(R,E,[T])} species(S)} \qquad (BR)\,\frac{T_1 \mid_\xi T_2 \xrightarrow{R,E} S \quad E \setminus (adm_\xi(T_1) \cup adm_\xi(T_2)) \neq \emptyset}{T_1, T_2 \xrightarrow{rate_B(R,E,[T_1,T_2])} species(S)}$$

$$\begin{aligned} reactions(T, (T_1, \ldots, T_N)) &\triangleq unary(T) \cup binary(T,T) \cup binary(T,T_1) \cup \ldots \cup binary(T,T_N) \\ unary(T) &\triangleq \{[T], r, C \mid T \xrightarrow{r} C\} \\ binary(I_1, I_2) &\triangleq \{([T_1, T_2], r, C), C \mid T_1, T_2 \xrightarrow{r} C\} \end{aligned}$$

Definition 13. Strand graph isomorphisms. Given a strand graph $S = (V, length, colour, A, toehold, E)$, an automorphism $\pi : V \rightarrow V$ is a permutation of the vertices such that

$$\begin{gathered}\forall v \in V.\text{colour}(v){=}\text{colour}(\pi\ v)\\ \forall v_1, n_1, v_2, n_2.(v_1, n_1) \overset{E}{\longleftrightarrow} (v_2, n_2) \iff (\pi\ v_1, n_1) \overset{E}{\longleftrightarrow} (\pi\ v_2, n_2)\\ \forall v_1, n_1, v_2, n_2.(v_1, n_1) \overset{A}{\longleftrightarrow} (v_2, n_2) \iff (\pi\ v_1, n_1) \overset{A}{\longleftrightarrow} (\pi\ v_2, n_2)\\ \forall e \in A.\text{toehold}(e){=}\text{toehold}(\pi\ e)\end{gathered}$$

That is, π is colour- (and therefore length-) preserving and is also permuting E and A in a coherent manner. Since this defines how π permutes edges, we allow ourselves to write $\pi\ e$ for $e \in A$ in the last constraint saying that π preserves toeholds. Given two strand graphs S and S', we say that S is isomorphic to S', written $S \sim S'$, if there exists an automorphism π on S such that $S' = \pi\ S$. Note that this definition relies on the assignment of colours to be coherent across strand graphs.

Proof. Left to right is trivial given that *sg* preserves *V, colour,* and *E*. Right to left follows from the fact that, since it preserves *colour,* ξ is a valid environment for $\pi(T)$ and $sg(\xi,\pi(T))$ will produce the same length and toehold functions and the same set of admissible edges.

This means that the local information is enough to establish isomorphism or, as we do in the implementation, to compute a canonical representation.

4.3 Canonical representations of species

During reaction enumeration, just-in-time simulations [21] and state space exploration, all species are kept in canonical form to enable quick equality checking. The canonical form is obtained by finding a canonical labelling of the nodes. In [32] a series of worst-case quadratic algorithms for finding a canonical labelling of site graphs is presented. There, a site graph is first transformed into a graph with coloured edges and then the ordering of the colours is used to define edge enumeration. Here, we already have coloured edges, by lifting the colour of nodes. Because the sites are ordered, we do not need to sort edges by colour during edge enumeration, we can simply order them by outgoing site. A much bigger gain in practice comes from the fact that a canonical labelling is determined as soon as we have decided on a starting vertex. This means that we can restrict ourselves to vertices of a single colour. We can, for instance, choose the lowest colour with a minimal set of nodes. Thus, if there is only a single strand type with only one instance in the species, we can immediately obtain a canonical labelling by enumerating from the lowest such, and no second phase to compare enumerations is necessary.

The worst-case complexity is still quadratic, which occurs when all strands have the same colour but none of them are isomorphic. This would occur, for instance, in a linear polymer. It is an interesting problem to speed up such cases; one solution might be to perform some abstraction, such as forgetting the length, similarly to how list predicates are handled in static analyses. Finally, even when systems grow very large, in many cases the connected components stay small, meaning that even quadratic complexity is tolerable.

4.4 Three-way initiated four-way branch migration example

We have run the implementation on all the examples presented in Section 2. The output graphs that are automatically generated by the implementation are substantial, therefore we have omitted them from the paper. The CRN generated for the example of Fig. 1D is presented in Fig. 5. The rounded boxes represent species and the square boxes represent reactions, each box being labelled by the reaction rate. Bold lines are inputs to reactions and thin arrows are outputs. The bold rounded boxes denote initial species in the system. These species can react via external binding to form a single complex. This complex can either perform a toehold unbind to reverse the reaction or a displacement to kick off the strand A, resulting in four strands that can perform a four-way branch migration followed by a reversible toehold unbinding. We have used the default rates from Visual DSD for binding, unbinding, and migration.

The state space produces analogous behaviour, as shown in Figure 6. The initial state is marked in bold, where we have specified one of each initial molecule. The implementation automatically detects the individual species as connected components of the strand graph corresponding to each state. Both the CRN and the state space were computed from the following program, as described in Fig. 1D.

```
<tether T2^!a X*!b T1^> | <A X!b T2^*!a> | <T1^* X!c RA> | <X* !c A*!d>
| <A!d>
```

We have also performed a stochastic simulation of the system, starting with 1000 molecules of the initial tethered species (Species 1, purple) and 1200 molecules of the other initial species (Species 4, light blue). The time course is presented in Figure 7. We observe the binding of the two initial species form the light green Species 5, which rapidly produces strand “A” (Species 0, dark blue) and another transient product (Species 7, dark brown, mostly covered by light green) which then performs the four-way branch migration followed by a toehold unbinding. This means that the final species enter into a stochastic equilibrium (the large Species 6 in red and one of the products is in dark green with the other product directly underneath). This simulation was run using the following program, where the components are given explicitly in order to assign multiplicities.

```
  1000 * ( <tether T2^!a X*!b T1^> | <A X!b T2^*!a> )

| 1200 * ( <T1^* X!c RA> | <X* !c A*!d> | <A!d> )
```

Note that if one of the explicit components were not connected, it would be split into separate species that would each be assigned the same multiplicity. For comparison, we note that all models tested so far that are compatible with both the new implementation and the most recent release of Visual DSD [25] are faster in the new implementation.

5 Discussion

In this paper we have presented a formal language for modelling domain-level interactions between nucleic acid complexes with arbitrary secondary structures. We have defined our language as a process calculus and have proposed a mapping to a canonical representation based on strand graphs, which is derived from site graphs by imposing an ordering on the sites in each vertex. This is necessary because vertices represent individual strands and sites represent domains that appear in a particular order on the strand. Edges between sites encode the bonds that exist between various domains. By defining local strand graph rewrite rules we have presented an operational semantics for deriving reactions between DNA structures. This significantly extends our previous work on the DSD programming language, by enabling additional classes of structures and interactions to be derived, such as dendritic structures and n-way branch migration reactions with $n \geq 4$. More generally, the approach can handle arbitrary DNA secondary structures. We have also defined an encoding to chemical reaction networks, which allows strand graph systems to be simulated using stochastic simulators based on the Gillespie algorithm [13] or ordinary differential equation solvers. This required computing the connected components of a strand graph, which correspond to the individual chemical species. We have also derived the state space of strand graph systems, which can be analysed by probabilistic model checking [20, 47] using tools such as Prism [15]. We have used our language to model several systems that were beyond the scope of previous implementations such as [25]. These include programmable biomolecular self-assembly pathways based on hairpin opening [45], and a recently-proposed three-way initiated four-way branch migration reaction scheme, which is the foundation for surface-based molecular Turing machines and reusable digital logic circuits [36]. Finally, we have shown that our language is capable of expressing all of the behaviours of previously published DSD semantics [21].

5.1 Related Work

The inspiration for a strand graph approach to modelling nucleic acid secondary structures came from kappa [7], a language for formally modelling interactions between agents via named sites, based on site graphs. Following the approach of kappa, we defined a formal syntax with an accompanying reduction semantics. Unlike kappa, however, our reduction rules were fixed to reflect existing knowledge of DNA chemistry, DNA strand displacement and hybridization interactions. Future work could investigate encodings of these rules directly in kappa, however this presents a number of challenges. First, the strand displacement rules rely on a notion of ordering between sites, which is currently not directly supported in kappa. More importantly, kappa rules allow site-specific interactions to be described, whereas here we wish to describe rules that are independent of the specific sites involved, but rely instead on notions of domain complementarity and relative positioning of sites. Another challenge is the ability to express primitives that require enumeration over an unspecified number of bonds, such as the *anchored* primitive. This requires checking the existence of a path of connected bonds within a junction that may contain an arbitrary numbers of branches. In kappa, a separate primitive would need to be defined for each junction with a specific number of branches. Although the current DSD language cannot be encoded in kappa directly, it would be interesting to consider extensions to kappa that would facilitate such encodings. This would enable DSD to be implemented directly in kappa, without requiring a custom implementation. A more detailed comparison with kappa is provided in Appendix D.

Ours is not the first attempt to model nucleic acid structures with graphs. In [28], a formalism where subgraphs are replaced by other subgraphs as a result of enzymatic reactions is presented. In [16, 17], a set of rewrite rules specifically targeting strand displacement systems is given. These rules correspond to GB, GU and G3, and the paper further treats the problem of efficient enumeration of reactions by computing the possible states of local connectivity of strands. These two formalisms both model backbones explicitly as a series of bonds, whereas here we have an implicit representation given by the ordering of sites. In [14] an approach similar to ours is presented, though not formalised as strand graphs. Rather, the reaction semantics is defined in terms of pseudocode algorithms. Furthermore, instead of deriving branch migration reactions based on displacing paths, a fixed set of rules is defined, including a rule corresponding to branch migration with N fixed at 4. An approach to efficient computation is also presented, based on simplifying the reaction graphs using assumptions of time scale separation. In [31] the backbone is again modelled explicitly, but here with the intention of expressing base stacking. This is currently not expressible in our formalism and represents an interesting opportunity for detailed system modelling, in particular for the derivation of leak reactions that occur via blunt-end stacking and subsequent strand displacement, as discussed in [35]. That work also includes enzymatic reactions such as restriction enzyme digest and ligation, which we have previously only been able to model by manually augmenting a DSD program with explicit chemical reactions [46]. However, our formalism does enable the derivation of more general strand displacement reactions, such as three-way initiated four-way branch migration.

A somewhat less related development is found in [19] and [18], where the expressivity of graph rewriting is explored. There, the relation between subgraphs and species is not as direct. Rather, the question is investigated of whether rewrite rules exist, even for unconstrained systems, that give desired behaviour, along with the question of synthesis of such rules. The intention is that nodes represent programmable particles and that the right programs will then realise the synthesised rules. As the computational power of nucleic acid systems is progressively tamed in laboratories, in particular with regard to the programmable manipulation of matter at the mesoscale using nanoscale interactions, it is possible that these two lines of research could be unified in the future.

In this paper, we have defined a process calculus based on explicitly-named bonds to provide a text-based front-end language for declaring arbitrary strand graph structures. A domain-level "dot-paren"-style language such as that used by NUPACK [48] could be used as an alternative. However, it is worth noting that, while our strand graph-based approach can encode any secondary structure directly, the dot-paren syntax approach requires an unbounded number of different kinds of parentheses to encode all possible secondary structures: for any fixed number of kinds of parentheses, there exist *pseudoknotted* structures that cannot be expressed. Pseudoknots are structures in which loops are formed by strands folding back on themselves in such a way that the loops are not well-nested within larger loops. A simple example would be `<x!i a y!j b x*!i c y*!j>`. An alternative could be to adopt a numeric approach to specifying nucleic acid secondary structure [9] that, like our process calculus syntax, can also encode arbitrary nucleic acid structures.

5.2 Future Work

In this paper we have expanded the set of reactions that we can derive by introducing the concept of "anchored" bonds. This generalizes the 3-way branch migration rules from previously published DSD semantics, which only allowed branch migration to occur when the "anchor" domain and the "displacing" domain were adjacent single-stranded domains on the same invader strand. This generalization is required to handle the three-way initiated four-way branch migration example [36], because the three-way migration involves a toehold and a migration domain that are on different strands but are connected by a double-stranded domain, following the associative toehold scheme [3]. It is also required to derive the final step of the three-way junction assembly example from [45]. Making this generalization raises interesting questions regarding the appropriate level of modelling detail concerning such intramolecular binding reactions. For example, in rule GU, bound complementary toeholds are only allowed to spontaneously unbind as long as the bond is not anchored. In particular, in the three-way initiated four-way branch migration example, this means that the initial toehold t_1^* cannot unbind between the completion of the three-way migration step and initiation of the four-way migration step. Although the complementary toehold domains may unbind, in practice they will rebind quickly as their effective local concentration is very high compared to diffusion-limited binding with another species. Making this simplification prevents the strand graph model from deriving highly unlikely polymerization reactions between multiple structures that have reached this point in the reaction pathway, i.e., whose t_1 and t_1^* toeholds are also exposed. The concept of "anchored" bonds is also used in rule G3: the requirement that the newly formed bond *a* be anchored requires that the migration reaction must produce a junction. These definitions also allow us to express leak reactions concisely: The leaky version of rule G3 does not require that the bond *a* being formed is anchored. Instead it states that the bond *e* being broken can have at most one adjacent domain. This relaxation means that rule GL can be used to initiate a displacement reaction without an anchored toehold binding event.

Previous work [24] presented a method for converting a detailed semantics into a hierarchy of more abstract semantics, by merging reactions that occur on relatively fast timescales, such as multiple strand displacement reactions triggered by a single invader strand [42]. This enabled more tractable models to be generated automatically for high-level analysis. In principle, this approach can also be directly applied to the strand graph systems presented here. Interestingly, three-way initiated four-way migration raises further questions regarding this merging of reactions. Merging multiple strand displacement steps is relatively straightforward when they occur sequentially and on a fast timescale. However, in three-way initiated four-way migration it is not in fact the case that the three-way strand displacement reaction must completely finish before the four-way migration can take place. Indeed, in practice they will most likely occur concurrently to some extent, in the sense that the four-way migration may proceed as far as the three-way migration has completed, and progress on the four-way migration may act as a brake on backward steps of the three-way migration. Thus it may make sense to merge such "concurrent" reactions.

The reduction semantics presented in this paper provides a novel formalism of branch migration reactions in terms of a *displacing path*. This allows the GM rule to derive four-way branch migration reactions as well as similar reactions involving more branches, such as 6- or 8-way branch migration. The concept of displacing paths allows us to derive such reactions without a special-case reaction rule, by requiring that bonds are swapped between neighbouring domains in a sequence that eventually terminates by reaching the first domain. Although this reasoning process is expressed in terms of sequential bond-passing, the bond-swapping reactions occur concurrently in practice. The requirement that $anchored(a_i, E)$ holds in rule GM requires that all of the branches involved in the migration need to have an anchor point. One could relax this constraint such that not all new bonds along a displacing path need to be anchored, and estimate the number of anchors by experiments or biophysical modelling.

The strand graph approach provides considerable expressiveness for defining secondary structures, and for deriving structures via dynamic interactions between existing species. In principle, one may define essentially arbitrary secondary structures without any consideration of their biophysical plausibility. For instance, a strand graph such as the following

```
<a!i b!j0 c!j1 d!j2 a*!i> | <d*!j2 c*!j1 b*!j0>
```

can be defined in the syntax, however the bond between *a* and *a** is not biophysically plausible unless the combined length of the domains in the duplex (*b*, *c* and *d*) is considerably longer than the persistence length of double-stranded DNA. Such bindings are currently not prevented in our syntax, however they could be disallowed in the reduction rules by augmenting the *hidden* predicate to take into account domain lengths and topology. A possible solution would be to integrate the strand graph system proposed in this paper with a biophysical model of nucleic acid structures. This could be achieved using a coarse-grained approach such as OxDNA [8] or Multistrand [40], or using a worm-like chain model [27] to compute the volume swept out by each domain, in order to determine whether two domains can interact. Such biophysical modelling could also be used to compute specific rates of interaction [12, 43], which is currently lacking in our approach and is an important direction for future work. A sufficiently detailed and rigorously parametrised system could in future be used as an underlying formalism to model nucleic acid folding pathways, involving DNA origami [38] and potentially RNA tiles [11].

This work represents an important step towards a general formal language for modelling, simulating and analysing the behaviour of arbitrary nucleic acid systems at the domain level. We envisage that future versions of the Visual DSD software will be based on the theoretical framework outlined in this paper, to facilitate the design of dynamic nucleic acid systems involving complex topologies.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants 1027877, 1028238, and 1318833. M.R.L. gratefully acknowledges support from the New Mexico Cancer Nanoscience and Microsystems Training Center (NIH/NCI grant 5R25CA153825).

Appendices

A An introduction to DNA strand displacement

In this appendix, we present a brief introduction to DNA strand displacement reactions [49]. Figure 9 shows our abstract representation of DNA strands, sequences, and structures. Each single-stranded DNA molecule is represented by a straight line, with parallel lines denoting duplexes where the two strands are bound together to form a double helix. We use domains to represent DNA subsequences, which we assume have been designed such that each domain x binds only to its complementary domain x*. Complementary DNA sequences obey the laws of Watson-Crick base-pairing, that is, A binds to T and C binds to G. Each single DNA strand has an inherent orientation: the two ends of the strand are known in the field of DNA chemistry as the 5′ and 3′ ends. In our diagrams, we denote the 3′ end of the DNA strand using an arrow, as is common in such diagrams of DNA structures. We distinguish between "long domains", written just as x, which we assume are on the order of 25 – 30 nucleotides in length, and "toehold domains", written with the carat symbol as $\hat{t}$, which we assume are on the order just 5 – 8 nucleotides in length. This difference means that long domains, when they bind to their complements, bind sufficiently stably that spontaneous unbinding of the domain is extremely unlikely. Thus, we assume that binding of long domains is irreversible. On the other hand, the binding between complementary toehold domains is relatively weak, so two strands that are only bound together by a toehold domain may spontaneously unbind. Thus, we assume that binding of toehold domains is reversible.

Figure 9 an example of a basic DNA strand displacement reaction. The reactants comprise a strand displacement "gate", which is a two-strand complex, and a single-stranded input strand. In the first reaction, the single-stranded input binds reversibly to the gate via the complementary $\hat{t}$ toehold. Thus, the overhanging toehold provides a nucleation point for the interaction between the input strand and the two-strand gate complex. Once the input strand is bound, the remainder of the input strand (the x domain) is held in place adjacent to the gate, which has the same DNA sequence (the x domain). This initiates a reaction known as a "branch migration", in which bases from the preivously bound strand unbind from the inhibitor and are replaced by bases from the invading input strand. This process is a random walk, and if the junction between the two strands reaches the far end of the complex then the upper strand that was previously strongly bonded to the lower strand of the gate is now only bound by a short toehold domain (the $\hat{u}$ domain). This allows the previously bound strand to reversibly unbind from the gate, leaving the input strand bound to the gate, the previously bound strand free in solution, and the previously bound toehold domain $\hat{u}$ exposed on the gate. We refer to this process as a "strand exchange" reaction because the diffusing input strand is reversibly exchanged for the diffusing output strand. If the secondary toehold domain $\hat{u}$ is omitted from the gate, the strand displacement reaction is reversible because the displaced output strand cannot rebind from the gate if the secondary toehold is absent. Irreversible strand displacement gates of this form, labeled with a fluorophore-quencher pair, are often used in experimental implementations of DNA strand displacement to detect the presence of certain input strands by observing the increase in fluorescence caused when the strand is displaced from the gate, thereby separating the fluorophore from the corresponding quencher molecule.

DNA strand displacement reactions have been used to implement a range of molecular information processing systems, including digital logic circuits [35, 41], artificial neural networks [37], chemical reaction networks [42], entropy-driven catalytic cycles [50], and distributed algorithms [4]. Strand displacement circuits have also been used for diagnostic applications [39].

B Vectorised Semantics

In this appendix, we present a "vectorised" version of the reduction semantics for processes, which enables multiple sequential migration reactions to be derived as a single merged reaction step.

$$(RB)\,\frac{\neg hidden(i, P')}{C(x, x*) \overset{RB,\{i\}}{\longrightarrow} C(x!i, x*!i) = P'} \qquad (RU)\,\frac{\neg anchored(i, P) \qquad toehold(x)}{P = C(x!i, x*!i) \overset{RU,\{i\}}{\longrightarrow} C(x, x*)}$$

$$(R3)\,\frac{anchored(\bar{j}, P') \qquad \bar{d}' = comp(\bar{d})}{C(\bar{d}', \bar{d}'!\bar{j}, \bar{d}!\bar{j}) \overset{R3,\{\bar{j}\}}{\longrightarrow} C(\bar{d}'!\bar{j}, \bar{d}', \bar{d}!\bar{j}) = P'}$$

$$(RM)\,\frac{anchored(\bar{j}_1, P') \ldots anchored(\bar{j}_N, P') \qquad \bar{d}' = comp(\bar{d})}{C(\bar{d}!\bar{j}_1, \bar{d}'!\bar{j}_1, \bar{d}!\bar{j}_2, \bar{d}'!\bar{j}_2, \ldots, \bar{d}!\bar{j}_N, \bar{d}'!\bar{j}_N) \overset{RM,\{\bar{j}_1,\ldots,\bar{j}_N\}}{\longrightarrow} C(\bar{d}!\bar{j}_1, \bar{d}'!\bar{j}_2, \bar{d}!\bar{j}_2, \bar{d}'!\bar{j}_3, \ldots, \bar{d}!\bar{j}_N, \bar{d}'!\bar{j}_1) = P'}$$

Definition 15. Vectorised reduction semantics of processes. The semantics is identical to the non-vectorised semantics of Definition 2, except that each domain d is replaced with a vector of domains $\bar{d} = d_1,\ldots, d_N$, and each bond i is replaced with a vector of bonds $\bar{\imath} = i_1,\ldots, i_N$ in rules (R3) and (RM). We write $\bar{d}!\bar{j}$ as short for $d_1!j_1,\ldots, d_N ! j_N$ and $\bar{d}'!\bar{j}$ as short for $d_N*!j_N,\ldots, d_1*!j_1$. We define corresponding vectorised functions as follows, with $adjacent(j_1, \ldots, j_N, P) = (adjacent(j_1, P) \bigcup adjacent(j_N, P)) \setminus \{j_1,\ldots, j_N\}$ and $anchored(j_1, \ldots, j_N, P) = anchored(j_1, P)$.

C Encoding the Classic DSD Semantics

C.1 Classic DSD syntax and semantics

Here we choose as our reference point the classic DSD syntax and semantics published in our previous work on stochastic simulation of biological modelling languages [21]. Without loss of generality, we assume that any "new" quantifiers, module definitions, and module instantiations have been expanded away, so that the classic DSD syntax can be written down as shown in Definition 16.

Definitions 17–20 recapitulate the reduction semantics of the classic DSD system that was initially published in [21].

C.2 A translation from classic DSD syntax into process calculus syntax

Now, we define a translation function $\lceil - \rceil$; that maps a classic DSD program Q into a corresponding program P in our process calculus syntax. The translation is defined by induction on the structure of classic DSD programs Q, as follows.

Name $X, Y, Z ::= N$	Long domain
$N^{\wedge}$	Short domain

Domain $D ::= X$	Domain
X^*	Complemented domain
Non-empty sequence $S ::= D_1 \dots D_N$	Sequence of domains $N \geq 1$
Possibly empty sequence $L, R ::= D_1 \dots D_N$	Sequence of domains $N \geq 0$
Strand $A ::= <S>$	Upper strand
$\{S\}$	Lower strand
Segment $C ::= \{L'\}<L>[S]<R>\{R'\}$	Gate segment
Gate $G ::= C$	Single segment
$C{:}G$	Lower strand concatenation
$C{::}G$	Upper strand concatenation
Classic DSD Program $Q ::= A$	Strand
G	Gate
$Q_1 \parallel Q_2$	Parallel composition of classic species

Definition 16. Syntax for classic DSD programs.

$$
\begin{gathered}
\dots\{\mathrm{L'N\hat{}*R'}\}\dots|<\mathrm{LN\hat{}R}> \xrightarrow{RB} \dots\{\mathrm{L'}\}<\mathrm{L}>[\mathrm{N\hat{}}]<\mathrm{R}>\{\mathrm{R'}\}\dots \\
\dots\{\mathrm{L'}\}<\mathrm{L}>[\mathrm{N\hat{}}]<\mathrm{R}>\{\mathrm{R'}\}\dots \xrightarrow{RU} \dots\{\mathrm{L'N\hat{}*R'}\}\dots|<\mathrm{LN\hat{}R}> \\
:::[\mathrm{S}]<\mathrm{N\hat{}R}>\{\mathrm{N\hat{}*}\}\dots \xrightarrow{RC} :::[\mathrm{SN\hat{}}]<\mathrm{R}>\dots \\
:::[\mathrm{S1}]<\mathrm{SR1}>:<\mathrm{L2}>[\mathrm{SS2}]::: \xrightarrow{RM} :::[\mathrm{S1S}]<\mathrm{R1}>:<\mathrm{L2S}>[\mathrm{S2}]::: \\
:::[\mathrm{S1}]<\mathrm{S2R1}>:<\mathrm{L2}>[\mathrm{S2}]<\mathrm{R2}>::: \xrightarrow{RD} :::[\mathrm{S1S2}]<\mathrm{R1}>\dots|<\mathrm{L2S2R2}> \\
:::[\mathrm{S1}]<\mathrm{R1N\hat{}R}>\{\mathrm{R1'}\}|\{\mathrm{LN\hat{}*L2'}\}<\mathrm{L2}>[\mathrm{S2}]::: \xrightarrow{GB} :::[\mathrm{S1}]<\mathrm{R1}>\{\mathrm{R1'}\}::\{\mathrm{L}\}[\mathrm{N\hat{}}]<\mathrm{R}>:\{\mathrm{L2'}\}<\mathrm{L2}>[\mathrm{S2}]::: \\
:::[\mathrm{S1}]<\mathrm{R1}>\{\mathrm{R1'}\}::\{\mathrm{L}\}[\mathrm{N\hat{}}]<\mathrm{R}>:\{\mathrm{L2'}\}<\mathrm{L2}>[\mathrm{S2}]::: \xrightarrow{GU} :::[\mathrm{S1}]<\mathrm{R1N\hat{}R}>\{\mathrm{R1'}\}|\{\mathrm{LN\hat{}*L2'}\}<\mathrm{L2}>[\mathrm{S2}]::: \\
:::[\mathrm{S1}]<\mathrm{SR1}>:<\mathrm{L2}>[\mathrm{S}]\{\mathrm{R1'}\}::\{\mathrm{L2'}\}<\mathrm{R2}>[\mathrm{S2}]::: \xrightarrow{GD} :::[\mathrm{S1S}]<\mathrm{R1}>\{\mathrm{R1'}\}|\{\mathrm{L2'}\}<\mathrm{L2SR2}>[\mathrm{S2}]:::
\end{gathered}
$$

Definition 17. Elementary reduction rules for classic DSD programs, reproduced from Definition 23 of [21]. In rule (RM) we assume that the first domain in R2 is distinct from the first domain in S2, so that branch migration is maximal along a given sequence and the rules (RM) and (RD) are mutually exclusive. We write $:::$ [S] as an abbreviation for the connection of S to another species via concatenation to either the upper or the lower strand. Similarly we write … { S}… for the connection of a lower strand to other species via lower-strand concatenation, and G… for the connection of G to another gate via concatenation to the lower strand.

$$
\begin{gathered}
\lceil Q_1 \| Q_2 \rceil \triangleq \lceil Q_1 \rceil | \lceil Q_2 \rceil \\
\lceil <S> \rceil \triangleq <S> \\
\lceil \{S\} \rceil \triangleq <\mathrm{rev}(S)> \\
\lceil G \rceil \triangleq \mathrm{tr}(G, [], [])
\end{gathered}
$$

The cases for parallel composition and upper strands are trivial, as is the case for lower strands, except that we must reverse the domain sequence when converting from the classic lower strand syntax to the standardized upper strand syntax of the process calculus representation. The non-trivial case of this definition is the case for a gate G, where we require an auxiliary function *tr* to recursively convert the segment-based classic DSD syntax to the strand-based process calculus syntax. Note that, in the call to *tr*, the arguments [] are empty lists of domains from the DSD process calculus syntax, not elements of the classic DSD syntax. The auxiliary function *tr* can be defined by recursion over the structure of classic DSD gates *G*, as follows.

$$\begin{gathered}
\mathrm{rotate}(<\mathrm{S}>) \triangleq \{\mathrm{rev}(\mathrm{S})\} \\
\mathrm{rotate}(\{\mathrm{S}\}) \triangleq <\mathrm{rev}(\mathrm{S})> \\
\mathrm{rotate}(\{\mathrm{L}'\}<\mathrm{L}>[\mathrm{S}]<\mathrm{R}>\{\mathrm{R}'\}) \triangleq \{\mathrm{rev}(\mathrm{R})\}<\mathrm{rev}(\mathrm{R}')>[\mathrm{S}*]<\mathrm{rev}(\mathrm{L}')>\{\mathrm{rev}(\mathrm{L})\} \\
\mathrm{rotate}(\mathrm{G1{:}G2}) \triangleq \mathrm{rotate}(\mathrm{G2}){:}:\mathrm{rotate}(\mathrm{G1}) \\
\mathrm{rotate}(\mathrm{G1}{:}:\mathrm{G2}) \triangleq \mathrm{rotate}(\mathrm{G2}){:}\mathrm{rotate}(\mathrm{G1}) \\
\mathrm{rev}(<\mathrm{S}> \triangleq <\mathrm{rev}(\mathrm{S})> \\
\mathrm{rev}(\{\mathrm{S}\}) \triangleq \{\mathrm{rev}(\mathrm{S})\} \\
\mathrm{rev}(\{\mathrm{L}'\}<\mathrm{L}>[\mathrm{S}]<\mathrm{R}>\{\mathrm{R}'\}) \triangleq \{\mathrm{rev}(\mathrm{R}')\}<\mathrm{rev}(\mathrm{R})>[\mathrm{rev}(\mathrm{S})]<\mathrm{rev}(\mathrm{L})>\{\mathrm{rev}(\mathrm{L}')\} \\
\mathrm{rev}(\mathrm{G1{:}G2}) \triangleq \mathrm{rev}(\mathrm{G1}){:}\mathrm{rev}(\mathrm{G2}) \\
\mathrm{rev}(\mathrm{G1}{:}:\mathrm{G2}) \triangleq \mathrm{rev}(\mathrm{G2}){:}:\mathrm{rev}(\mathrm{G1})
\end{gathered}$$

Definition 18. Rotating and reverisng strands and gates in the classic DSD calculus, reproduced from Definition 24 of [21]. We write `rev( S)` for the domain list whose order is reversed from that in `S`.

$$\begin{gathered}
\mathrm{G} \equiv \mathrm{rotate}(\mathrm{G}) \\
\mathrm{A} \equiv \mathrm{rotate}(\mathrm{A}) \\
:::[\mathrm{S1}]<\mathrm{R1}>\{\mathrm{RS}\}{:}\{\mathrm{L}\}<\mathrm{L2}>[\mathrm{S2}]::: \equiv :::[\mathrm{S1}]<\mathrm{R1}>\{\mathrm{R}\}{:}\{\mathrm{SL}\}<\mathrm{L2}>[\mathrm{S2}]::: \\
:::[\mathrm{S1}]<\mathrm{RS}>\{\mathrm{R1}\}:::\{\mathrm{L2}\}<\mathrm{L}>[\mathrm{S2}]::: \equiv :::[\mathrm{S1}]<\mathrm{R}>\{\mathrm{R1}\}::\{\mathrm{L2}\}<\mathrm{SL}>[\mathrm{S2}]::: \\
\mathrm{D} \equiv \mathrm{D}' \Rightarrow \mathrm{D}|\mathrm{D2} \equiv \mathrm{D}'|\mathrm{D2}
\end{gathered}$$

Definition 19. Structural congruence rules for the classic DSD calculus, reproduced from Definition 25 of [21]. In addition to these rules, we assume that parallel composition of classic DSD species is commutative and associative, and that the structural congruence relation is reflexive, symmetric and transitive (i.e., is an equivalence relation).

$$\begin{gathered}
\mathrm{D} \xrightarrow{R} \mathrm{D}' \Rightarrow \mathrm{rev}(\mathrm{D}) \xrightarrow{R} \mathrm{rev}(\mathrm{D}') \\
\mathrm{D} \xrightarrow{R} \mathrm{D}' \Rightarrow \mathrm{com}(\mathrm{D}) \xrightarrow{R} \mathrm{com}(\mathrm{D}') \\
\mathrm{D} \xrightarrow{R} \mathrm{D}' \Rightarrow \mathrm{D}|\mathrm{D2} \xrightarrow{R} \mathrm{D}'|\mathrm{D2} \\
\mathrm{D2} \equiv \mathrm{D} \xrightarrow{R} \mathrm{D}' \equiv \mathrm{D2}' \Rightarrow \mathrm{D2} \xrightarrow{R} \mathrm{D2}'
\end{gathered}$$

Definition 20. Inductive reduction rules for the classic DSD calculus, reproduced from Definition 26 of [21]. We write `rev( D)` for the function that reverses all of the strands and complexes in process `D`, and `com( D)` for the function that complements all of the domains in process `D`.

$$\begin{aligned}
\mathrm{tr}(\{L'\}{<}L{>}[d_1\cdots d_n]{<}R{>}\{R'\},\overline{d}_{\text{upper}},\overline{d}_{\text{lower}}) &\triangleq {<}\overline{d}_{\text{upper}}Ld_1!i_1\cdots d_n!i_nR{>}|{<}\mathrm{rev}(\overline{d}_{\text{lower}}L'd_1^*!i_1\cdots d_n^*!i_nR'){>}\text{ where }i_1\\
\mathrm{tr}(\{L'\}{<}L{>}[d_1\cdots d_n]{<}R{>}\{R'\}{:}G,\overline{d}_{\text{upper}},\overline{d}_{\text{lower}}) &\triangleq {<}\overline{d}_{\text{upper}}Ld_1!i_1\cdots d_n!i_nR{>}|(\mathrm{tr}(G,[],(\overline{d}_{\text{lower}}L'd_1^*!i_1\cdots d_n^*!i_nR')))\text{ wher}\\
\mathrm{tr}(\{L'\}{<}L{>}[d_1\cdots d_n]{<}R{>}\{R'\}{::}\,G,\overline{d}_{\text{upper}},\overline{d}_{\text{lower}}) &\triangleq {<}\mathrm{rev}(\overline{d}_{\text{lower}}L'd_1^*!i_1\cdots d_n^*!i_nR'){>}|(\mathrm{tr}(G,(\overline{d}_{\text{upper}}Ld_1!i_1\cdots d_n!i_nR),[]))\text{ w}
\end{aligned}$$

The arguments $\bar{d}_{\text{upper}}$ and $\bar{d}_{\text{lower}}$ to the *tr* function are lists of domains from the DSD process calculus syntax. These lists are used to record the upper and lower domains observed in the segments to the left: hence, they are initialized to be empty when the function is called from $\lceil G\rceil$. When a gate concatenation is encountered, one of these strands will be terminated, depending on whether the concatenation was on the upper or lower strand. The terminated strand is constructed by taking the appropriate stored domains from $\bar{d}_{\text{upper}}$ or $\bar{d}_{\text{lower}}$, and concatenating them with the appropriate (upper or lower) domains from the current segment. For the recursive call, that list is reset to empty and the domains from the continuing strand are transferred to the other list for storage. The bonds between domains that were previously represented implicitly by being inside the [−] duplex component of the segment, are now represented explicitly using named bonds in the process calculus syntax: we assume that the new bond names are chosen freshly for each double-stranded domain.

It is straightforward to see that the cases of the translation for parallel composition, upper strands, and lower strands are correct. The case for gates is almost as simple: we observe that the order of domains in strands is preserved by the translation, including across domain boundaries, and that the bonds between domains are also duplicated correctly.

C.3 Correspondence between classic DSD semantics and process calculus semantics

We will use as our starting point the DSD semantics from [21]. We write $Q \Rightarrow Q'$ to mean that the classic DSD program Q can be reduced to the classic DSD program Q' using the semantic rules from [21], and we write $P \rightarrow P'$ for a reduction from the process calculus program P to the process calculus program P' using the process calculus semantics. Our aim here is to prove the following proposition.

Proposition 21. For any classic DSD programs Q and Q', if $Q \Rightarrow Q'$ then $\lceil Q\rceil \rightarrow \lceil Q'\rceil$.

Before we can prove the correspondence, we require some preliminary Lemmas.

Lemma 22. For all gates G, $\lceil G\rceil = \lceil \texttt{rotate}(G)\rceil$. Similarly, for all strands A, $\lceil A\rceil = \lceil \texttt{rotate}(A)\rceil$.

Proof. Straightforward, given the definition of `rotate` from Definition 18.

Lemma 23. For all classic DSD programs Q, if $\lceil Q\rceil = {<}S_1{>} \cdots {<}S_n{>}$ then $[\ \texttt{rev}(Q)] = {<}\texttt{rev}(S_1){>} \cdots {<}\texttt{rev}(S_n){>}$. Furthermore, $\texttt{rev}(\lceil Q\rceil) = \lceil \texttt{rev}(Q)\rceil$.

Proof. The first claim is straightforward, by considering each species from Q on a per-species basis and using the definition of `rev` from Definition 18. The second claim follows from the first.

Lemma 24. For all classic DSD programs Q, if $\lceil Q \rceil = <S_1> \cdots <S_n>$ then $\lceil$ `com`$(Q) \rceil = <$ `com`$(S_1)> \cdots <$ `com`$(S_n)>$. Furthermore, `com`$(\lceil Q \rceil) = [$ `com`$(Q)]$.

Proof. The first claim is straightforward, by considering each species from Q on a per-species basis. The second claim follows from the first.

Lemma 25. For all classic DSD programs Q and Q', if $Q = Q'$ then $\lceil Q \rceil = \lceil Q' \rceil$.

Proof. The proof proceeds by induction on the height of the derivation of $Q \equiv Q'$, which is defined in Definition 19. The cases are as follows.

- **First rule.** We assume that $G \equiv$ `rotate`(G). Then, the result follows from Lemma 22.
- **Second rule.** We assume that $A \equiv$ `rotate` (A). Then, the result also follows from Lemma 22.
- **Third rule.** We assume that $Q \equiv Q'$, where

  $$Q {=} : : :[S_1]{<}R_1{>}\{RS\}{:}\{L\}{<}L_2{>}[S_2] : : :$$
  $$Q^{'} {=} : : :[S_1]{<}R_1{>}\{R\}{:}\{SL\}{<}L_2{>}[S_2] : : :$$

  both hold. Then, by straightforward calculations we get that

  $$\lceil Q \rceil {=} {<}\hat{L} S_1 ! \overrightarrow{i}\, R1{>} | {<}L_2 S_2 ! \overrightarrow{j}\, \hat{R}{>} | {<}\widehat{R'} S_2^* ! \overrightarrow{j}\, \mathrm{rev}(L)\mathrm{rev}(S)\mathrm{rev}(R) S_1^* ! \overrightarrow{i}\, \widehat{L'}{>} {=} \lceil Q^{'} \rceil$$

  for some L, $\hat{R}$, $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$, as required.
- **Fourth rule.** We assume that $Q \equiv Q'$, where

  $$Q {=} : : :[S_1]{<}RS{>}\{R_1\}{:}\{L_2\}{<}L{>}[S_2] : : :$$
  $$Q^{'} {=} : : :[S_1]{<}R{>}\{R_1\}{:}\{L_2\}{<}SL{>}[S_2] : : :$$

  both hold. Then, by straightforward calculations we get that

  $$\lceil Q \rceil {=} {<}\hat{L} S_1 ! \overrightarrow{i}\, RSLS_2 ! \overrightarrow{j}\, \hat{R}{>} | {<}\widehat{R'} S_2^* ! \overrightarrow{j}\, \mathrm{rev}(L_2){>} | {<}\mathrm{rev}(R_1) S_1^* ! \overrightarrow{i}\, \widehat{L'}{>} {=} \lceil Q^{'} \rceil$$

  for some L, $\hat{R}$, $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$, as required.
- **Fifth rule.** We assume that $Q = Q_1 \| Q_2$ and $Q^{'} {=} Q_1^{'} \| Q_2$, and hence that $Q_1 \| Q_2 \equiv Q_1^{'} \| Q_2^{'}$. Then, $\lceil Q \rceil = \lceil Q_1 \rceil \mid \lceil Q_2 \rceil$ and $\lceil Q^{'} \rceil {=} \lceil Q_1^{'} \rceil | \lceil Q_2 \rceil$. By assumption we have $Q_1 \equiv Q_1^{'}$ and by induction it follows that $\lceil Q_1 \rceil {=} \lceil Q_1^{'} \rceil$. Thus we get that $\lceil Q_1 \rceil | \lceil Q_2 \rceil {=} \lceil Q_1^{'} \rceil | \lceil Q_2 \rceil$, and hence $\lceil Q_1 \| Q_2 \rceil {=} \lceil Q_1^{'} \| Q_2 \rceil$, i.e., $\lceil Q \rceil = \lceil Q' \rceil$, as required.

- **Parallel composition is commutative and associative.** This case follows from the fact that systems in the process calculus syntax are identified up to reordering of strands.
- **Structural congruence is an equivalence relation.** This case follows from the fact that equality on translated classic DSD programs is also an equivalence relation.

Thus, we have covered all required cases, which completes the proof of Lemma 25.

We are now in a position to prove the correspondence theorem.

Theorem 26. For any classic DSD programs Q and Q', if $Q \Rightarrow Q'$ then $\lceil Q \rceil \to \lceil Q' \rceil$.

Proof. The proof proceeds by induction on the height of the derivation of $Q \Rightarrow Q'$. The base cases, which correspond to the rules from Definition 17, are as follows.

- **Rule RB.** In this case, we get that

$$\lceil Q \rceil = \lceil \ldots \{L' N\hat{} * R'\} \ldots \| {<} L N\hat{} R {>} \rceil = {<} \widehat{R'} \mathrm{rev}(R') N\hat{} * \mathrm{rev}(L') \widehat{L'} {>} | {<} L N\hat{} R {>} | {<} \hat{P} {>}$$
$$\lceil Q' \rceil = \lceil \{L'\} {<} L {>} [N\hat{} *] {<} R {>} \{R'\} \rceil = {<} L N\hat{} !i R {>} | {<} \widehat{R'} \mathrm{rev}(R') N\hat{} * !i\, \mathrm{rev}(L') \widehat{L'} {>} | {<} \hat{P} {>}$$

  both hold, for some $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$, and for some fresh *i*. We note that the newly-formed bond *i* is not within a hairpin, and therefore, using rule RB of the semantics of the process calculus, we can show that $\lceil Q \rceil \to \lceil Q' \rceil$ holds, as required.

- **Rule RU.** The argument for this case is similar to that for RB: the calculations are the same, with the only additional requirement being that we must show that in the reduction context in the process calculus semantics there is no bond adjacent to bond *i*. However, this is trivial because the domains in L and R (if they exist) are not bound to the lower strand, so there are no adjacent bonds.

- **Rule RC.** In this case, we get that

$$\lceil Q \rceil = \lceil : : : [S] {<} N\hat{} R {>} \{N\hat{} *\} \ldots \rceil = {<} \hat{L} S! \overrightarrow{i}\, N\hat{} R {>} | {<} \widehat{R'} N\hat{} * S^*! \overrightarrow{i}\, \widehat{L'} {>} | \hat{P}$$
$$\lceil Q' \rceil = \lceil : : : [S N\hat{}] {<} R {>} \ldots \rceil = {<} \hat{L} S! \overrightarrow{i}\, N\hat{} !j R {>} | {<} \widehat{R'} N\hat{} * !j S^*! \overrightarrow{i}\, \widehat{L'} {>} | \hat{P}$$

  both hold, for some $\hat{L}$, $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$, and for some fresh *i*. We note that the newly-formed bond *i* is not within a hairpin, and therefore, using rule RB of the semantics of the process calculus, we can show that $\lceil Q \rceil \to \lceil Q' \rceil$ holds, as required. (Note that we cannot use rule RU derive the reverse reaction $\lceil Q' \rceil \to \lceil Q \rceil$ in the process calculus in this case, because the domains from S and S^* are in a duplex so the resulting context is not a valid unbinding context.)

- **Rule RM.** To do. In this case, we get that

$$\lceil Q \rceil = \lceil : : : [S_1] {<} S R_1 {>} : {<} L_2 {>} [S S_2] : : : \rceil = {<} \hat{L} S_1! \overrightarrow{i}\, S R_1 {>} | {<} L_2 S! \overrightarrow{j}\, S_2! \overrightarrow{k}\, \hat{R} {>} | {<} \widehat{R'} S_2^*! \overrightarrow{k}\, S^*! \overrightarrow{j}\, S_1^*! \overrightarrow{i}\, \widehat{L'} {>} | \hat{P}$$
$$\lceil Q' \rceil = \lceil : : : [S_1 S] {<} R_1 {>} : {<} L_2 S {>} [S_2] : : : \rceil = {<} \hat{L} S_1! \overrightarrow{i}\, S! \overrightarrow{j}\, R_1 {>} | {<} L_2 S S_2! \overrightarrow{k}\, \hat{R} {>} | {<} \widehat{R'} S_2^*! \overrightarrow{k}\, S^*! j S_1^*! \overrightarrow{i}\, \widehat{L'} {>} | \hat{P}$$

both hold, for some L, $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$. Then, we can use the process calculus rule R3 to conclude that $\lceil Q \rceil \rightarrow \lceil Q' \rceil$, as required. (Note that the process calculus reduction rules do not in fact *require* the migration reaction to be maximal, unlike the rules from [21].)

- **Rule RD.** In this case, we get that

$$\lceil Q\rceil = \lceil :::[S_1]<S_2R_1>:<L_2>[S_2]<R_2>\ldots\rceil = <\hat{L}S_1!\overrightarrow{i}\,S_2R_1>|<L_2S_2!\overrightarrow{j}\,R_2>|<\widehat{R'}S_2^*!\overrightarrow{j}\,S_1^*!\overrightarrow{i}\,\widehat{L'}>|\hat{P}$$
$$\lceil Q'\rceil = \lceil :::[S_1S_2]<R_1>\ldots\,\|<L_2S_2R_2>\rceil = <\hat{L}S_1!\overrightarrow{i}\,S_2!\overrightarrow{j}\,R_1>|<\widehat{R'}S_2^*!\overrightarrow{j}\,S_1^*!\overrightarrow{i}\,\widehat{L'}>|<L_2S_2R_2>|\hat{P}$$

both hold, for some L, $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$. Then, we can use the process calculus rule R3 to conclude that $\lceil Q \rceil \rightarrow \lceil Q' \rceil$, as required.

- **Rule GB**. In this case, we get that

$$\lceil Q\rceil = \lceil :::[S_1]<R_1N\hat{}\,R>\{R_1'\}\|\{LN\hat{}*L_2'\}<L_2>[S_2]:::\rceil = <\hat{L}S_1!\overrightarrow{i}\,R_1N\hat{}\,R>|<\mathrm{rev}(R_1')S_1^*!\overrightarrow{i}\,\widehat{L'}>|<L_2S_2!\overrightarrow{j}\,\hat{R}>|<\widehat{R'}S_2^*!$$
$$\lceil Q'\rceil = \lceil :::[S_1]<R_1>\{R_1'\}::\{L\}[N\hat{}]<R>:\{L_2'\}[S_2]:::\rceil = <\hat{L}S_1!\overrightarrow{i}\,R_1N\hat{}!kR>|<\mathrm{rev}(R_1')S_1^*!\overrightarrow{i}\,\widehat{L'}>|<L_2S_2!\overrightarrow{j}\,\hat{R}>|<\widehat{R'}S_2^*!$$

both hold, for some L, $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$, and for some fresh k. We note that the newly-formed bond k is not within a hairpin, and therefore, using rule RB of the semantics of the process calculus, we can show that $\lceil Q \rceil \rightarrow \lceil Q' \rceil$ holds, as required.

- **Rule GU.** The argument for this case is similar to that for GB: the calculations are the same, with the only additional requirement being that we must show that in the reduction context in the process calculus semantics there is no bond adjacent to bond k. However, this is trivial because the rule specifies that the domains in R_1 and R (if they exist) are not bound to the lower strand, so there are no adjacent bonds.

- **Rule GD.** In this case, we get that

$$\lceil Q\rceil = \lceil :::[S_1]<SR_1>:<L_2>[S]\{R_1'\}::\{L_2'\}<R_2>[S_2]:::\rceil = <\hat{L}S_1!\overrightarrow{i}\,SR_1>|<\mathrm{rev}(R_1')S^*!\overrightarrow{j}\,S_1^*!\overrightarrow{i}\,\widehat{L'}>|<L_2S!\overrightarrow{j}\,R_2S_2!\overrightarrow{k}\,\hat{R}>$$
$$\lceil Q'\rceil = \lceil :::[S_1S]<R_1>\{R_1'\}\|\{L_2'\}<L2SR2>[S_2]:::\rceil = <\hat{L}S_1!\overrightarrow{i}\,S!\overrightarrow{j}\,R_1>|<\mathrm{rev}(R_1')S*!\overrightarrow{j}\,S_1^*!\overrightarrow{i}\,\widehat{L'}>|<L_2SR_2S_2!\overrightarrow{k}\,\hat{R}>|<$$

both hold, for some L, $\widehat{L'}$, $\widehat{R'}$, and $\hat{P}$. Then, we can use the process calculus rule R3 to conclude that $\lceil Q \rceil \rightarrow \lceil Q' \rceil$, as required.

The inductive cases, which correspond to the rules from Definition 20, are as follows.

- **First rule.** We assume that $\mathtt{rev}(Q) \Rightarrow \mathtt{rev}(Q')$, and by assumption we get that $Q \Rightarrow Q'$. By induction we get $\lceil Q \rceil \rightarrow \lceil Q' \rceil$. Then, since the rules of the process calculus are closed under the rev operation (in the case of rule R3, to see this we note that the same context can be used to derive 3-way migrations in both the 3′ and 5′ directions), we get that $\mathtt{rev}(\lceil Q \rceil) \rightarrow \mathtt{rev}(\lceil Q' \rceil)$. Finally, by Lemma 23 we get $\lceil \mathtt{rev}(Q) \rceil \rightarrow \lceil \mathtt{rev}(Q') \rceil$, as required.

- **Second rule.** We assume that $\texttt{com}(Q) \Rightarrow \texttt{com}(Q')$, and by rule RC we get that $Q \Rightarrow Q'$. By induction we get $\lceil Q \rceil \to \lceil Q' \rceil$. Since the reduction rules of the process calculus do not specify which domains are complemented, only their "relative complementarity", it follows that the process semantics is closed under the com operation. Hence we get $\texttt{com}(\lceil Q \rceil) \to \texttt{com}(\lceil Q' \rceil)$, and by Lemma 24 it follows that $\lceil \texttt{com}(Q) \rceil \to \texttt{com}(Q') \rceil$, as required.

- **Third rule.** We assume that $Q = Q_1 \| Q_2$ and $Q' = Q_1' \| Q_2'$, and that $Q \Rightarrow Q'$. Then, by assumption we get that $Q_1 \Rightarrow Q_1'$, and by induction we get that $\lceil Q_1 \rceil \to \lceil Q_1' \rceil$ holds. Since we can add additional strands to the context in any process calculus reduction, it follows that $\lceil Q_1 \rceil | \lceil Q_2 \rceil \to \lceil Q_1' \rceil | \lceil Q_2 \rceil$ holds, i.e., that $(\lceil Q \rceil) \to \lceil Q' \rceil$, as required.

- **Fourth rule.** We assume that $Q \Rightarrow Q'$. Then, by assumption there exist Q'' and Q''' such that $Q \equiv Q''$ and $Q'' \equiv Q'''$ and $Q''' \equiv Q'$ all hold. By induction we get that $\lceil Q'' \rceil \to \lceil Q''' \rceil$ holds. Then, by Lemma 25 we get that $\lceil Q \rceil = \lceil Q'' \rceil$ and $\lceil Q''' \rceil = \lceil Q' \rceil$. Thus it follows that $\lceil Q \rceil \to \lceil Q' \rceil$ holds, as required.

Thus, we have covered all required cases, which completes the proof of Lemma 25.

C.4 Comments on the semantic correspondence

If we were able to delimit the subset of process calculus programs that correspond to a classic DSD program, then we could define a reverse translation $\lfloor P \rfloor$ that turns a process calculus program back into the corresponding classic DSD program, if such a classic DSD program exists. Then, we could try to prove the reverse version of Theorem 26, i.e, if P is a process calculus program that corresponds to a classic DSD program, then $P \to P'$ implies $\lfloor P \rfloor \Rightarrow \lfloor P' \rfloor$. However, this would entail showing that if P corresponds to a classic DSD program and $P \to P'$ then P' also corresponds to a classic DSD program. This property does not hold for arbitrary process calculus programs P, because the process calculus semantics allows additional reactions, e.g. reactions between single-stranded overhangs that would form tree-like structures. The classic DSD semantics neglects such reactions, even though they might be possible in practice, because the classic DSD syntax cannot encode such structures. Therefore, the reverse direction of Theorem 26 does not hold in general. Thus, the new semantics is not a conservative extension of the classic DSD semantics but rather a proper extension.

D Comparison with Kappa

In this appendix we use a simple example to illustrate how a DSD program can be encoded into kappa [10]. We outline some of the challenges involved, together with potential areas for future work. The example we consider is the three-way initiated four-way branch migration example described in Fig. 1D. The DSD code for the example is as follows:

```
<L T2^!i2 X*!i1 T1^> | <A X!i1 T2^*!i2> |
<T1^* X!j1 R> | <X*!j1 A*!j2> | <A!j2>
```

The code defines five strands that represent the initial state of the system. From this initial state, the full state space is automatically generated by repeated application of the DSD rules in Definition 2, as illustrated in Fig. 1D.

We refer the reader to [10] for a description of the kappa language, and to [7, 6] for technical definitions. Briefly, kappa is defined in terms of *site graphs*, where the vertices of the graph are also referred to as *agents.* When encoding a DSD program into kappa, an agent represents a DNA strand and a site represents a domain. By definition, each agent has a unique name and the sites belonging to a given agent are also unique. To represent our DSD example in kappa, we define unique agent names $X_1,\ldots, X_5$ to represent the strands. Since a given domain may occur multiple times within the same strand, we cannot use the domain itself as the site name. Instead, we define unique site names $s_1,\ldots, s_4$ and annotate each site with the domain, using the kappa notation for an *internal state*, where a site s with internal state L is written $s \sim L$. Thus, the internal state of a site is used to represent the domain at that site. Finally, we note that kappa does not directly support the notion of complementary states. Instead, we use a naming convention such that the state x' is assumed to be complementary to x. Using this approach, we can encode the DSD example of Fig. 1D into kappa as follows:

```
X1(s1~L, s2~T2!i2, s3~X′!i1, s4~T1), X2(s1~A, s2~X!i1, s3~T2′!i2),

X3(s1~T1′, s2~X!j1, s3~R), X4(s1~X′!j1, s2~A′!j2), X5(s1~A!j2)
```

Although the encoding of the DSD syntax into kappa syntax is straightforward, a number of difficulties arise when trying to encode the DSD semantics (Definition 2) as kappa rules. In particular, to express the binding rule (RB) we need a way of stating that any site in any agent can bind to any other site in any other agent, provided the two sites have complementary states. Although there are extensions to kappa which permit a given rule to be defined once and applied to multiple agents [5], additional generalisations are still needed to allow a given rule to be applied to multiple sites and to define predicates over states, such as complementarity. In the absence of such generalisations, binding rules tailored to the example under consideration are required. For our specific example we end up with the following binding rule:

```
X1(s4~T1), X3(s1~T1′) -> X1(s4~T1!i3), X3(s1~T1′!i3)
```

This rule states that site s4 of agent X1 can bind to site s1 of agent X3. However, if we change the example slightly so that the complementary domains T1 and T1′ are on different sites, the rule is no longer applicable and a different rule needs to be defined. Thus, the binding rule is specific to the example.

Similarly, the unbinding rule (RU) for our example is defined in kappa as

```
X1(s4~T1!i3, s3~X′!i1), X3(s1~T1′!i3,s2~X!j1) –> X1(s4~T1, s3~X′!i1),
X3(s1~T1′,s2~X!j1)
```

This rule states that site s4 of agent X1 can unbind from site s1 of agent X3, provided site s3 of agent X1 is not bound to site s2 of agent X3. In other words, bond j1 and i1 must be different. This condition is necessary because, according to rule (RU), a bond between two sites can only be broken if there are no adjacent bonds. Again, the kappa rule here is specific to our example, since there is currently no general way in kappa to specify that no adjacent bonds are present.

The branch migration rule (R3) for this example is defined in kappa as follows:

```
X1(s3~X′!i1, s4~T1!i3), X2(s1~A, s2~X!i1),

X3(s1~T1′!i3, s2~X!j1), X4(s1~X′!j1, s2~A′!j2), X5(s1~A!j2)

–>

X1(s3~X′!i1, s4~T1!i3), X2(s1~A!j2, s2~X!i1),

X3(s1~T1′!i3, s2~X!j1), X4(s1~X′!j1, s2~A′!j2), X5(s1~A)
```

This illustrates another limitation of kappa, in that there is no way to define a generalised version of the *anchored* predicate, which checks whether a bond is part of a junction of arbitrary size. As a result, the junction needs to be represented explicitly. In this particular example the junction is made up of bonds i1, i3, j1, j2.

Finally, the N-way branch migration rule (RM) for this example is defined in kappa as follows:

```
X1(s3~X′!i1, s4~T1!i3), X2(s1~A!j2, s2~X!i1),

X3(s1~T1′!i3, s2~X!j1), X4(s1~X′!j1, s2~A′!j2)

–>

X1(s3~X′!j1, s4~T1!i3), X2(s1~A!j2, s2~X!i1),

X3(s1~T1′!i3, s2~X!j1), X4(s1~X′!i1, s2~A′!j2)
```

Here again, the specific junction needs to be represented explicitly, and there is no way of encoding a branch migration for a junction of arbitrary size N.

Furthermore, we note that the result of the four-way branch migration can also perform an unbind. However, since this unbinding takes place on a different site to the first unbinding, an additional kappa rule needs to be defined:

```
X1(s1~L, s2~T2!i2, s3~X′!j1), X2(s2~X!i1, s3~T2′!i2)

->

X1(s1~L, s2~T2, s3~X′!j1), X2(s2~X!i1, s3~T2′)
```

Thus, the kappa rule set is specific for a given DSD program and could potentially be substantial, since it needs to account for all of the different ways in which binding, unbinding and migration can occur in that example. This contrasts with a fixed set of only four rules in the DSD semantics.

We summarise the extensions to kappa that would be required in order to implement the DSD semantics of Definition 2 in a way that is independent of the specific example being considered:

- The ability to define meta rules that can be applied to all agents and all sites
- The ability to define predicates on states, in order to specify that any two sites can bind provided they have complementary states.
- The ability to define predicates such as adjacency in order to specify that two sites can only unbind if there are no bonds between corresponding adjacent sites.
- The ability to define predicates using enumeration $i \in \{1,\dots, N\}$ for arbitrary values of N, for predicates such as *anchored*, to specify that a given bond is part of a junction of arbitrary size, and *hidden*, to specify that one end of a bond is hidden within a closed loop of arbitrary size.

Without these extensions, each example will require a manually encoded set of kappa rules. The power of kappa is that it allows a custom set of rules to be defined for each specific system under consideration. In contrast, with DSD we define a small set of general rules that capture the behaviour of a broad range of strand displacement systems, so that custom rules are not required to model each specific system under consideration: the same rules apply to all systems. This difference in approach requires the ability to define rules of a much more general nature than is currently possible in kappa.

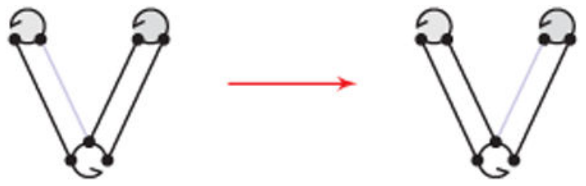
D.1 Relation to site graphs

The fact that domains are on a strand in a specificorder distinguishes strand graphs from site graphs. As an example, consider a simple strand displacement reaction

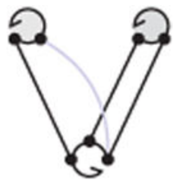
```
          <a!i b> | <c*!k b* !j a*!i> | <b!j c!k>


->(R3) <a!i b!j> | <c*!k b* !j a*!i> | <b c!k>
```

which can be depicted as the following strand graph transformation:



If we do not have an order of the sites of a node, the the left-most situtation could also represent the program `<a!i b> | <b*!j c*!k a*!i> | <c!k b!j>`, which as a strand graph would be



and should not be possible. As site graphs, if one imagines the nodes drawn as just circles, the reactive and the non-reactive graphs are indistinguishable, even though the sites are named.

References

1. Amir, Yaniv; Ben-Ishay, Eldad; Levner, Daniel; Ittah, Shmulik; Abu-Horowitz, Almogit; Bachelet, Ido. Universal computing by DNA origami robots in a living animal. Nature Nanotechnology. May; 2014 9(5):353–7.
2. Chandran, Harish; Gopalkrishnan, Nikhil; Phillips, Andrew; Reif, John. Localized hybridization circuits. In: Cardelli, Luca; Shih, William, editors. Proceedings of the 17th International Conference on DNA Computing and Molecular Programming, volume 6937 of Lecture Notes in Computer Science. Springer-Verlag; 2011. p. 64-83.
3. Chen, Xi. Expanding the rule set of DNA circuitry with associative toehold activation. Journal of the American Chemical Society. 2012; 134:263–271. [PubMed: 22129141]
4. Chen, Yuan-Jyue; Dalchau, Neil; Srinivas, Niranjan; Phillips, Andrew; Cardelli, Luca; Soloveichik, David; Seelig, Georg. Programmable chemical controllers made from DNA. Nature Nanotechnology. 2013; 8:755–762.
5. Danos, Vincent; Feret, Jérôme; Fontana, Walter; Harmer, Russell; Krivine, Jean. Rule-based modelling and model perturbation. Transactions on Computational Systems Biology. 2009; 11:116–137.
6. Danos, Vincent; Feret, Jérôme; Fontana, Walter; Krivine, Jean. Abstract interpretation of cellular signalling networks. In: Logozzo, Francesco; Peled, Doron; Zuck, Lenore D., editors. Proceedings, volume 4905 of Lecture Notes in Computer Science; Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008; San Francisco, USA. January 7-9, 2008; Springer; 2008. p. 83-97.
7. Danos, Vincent; Laneve, Cosimo. Formal molecular biology. Theoretical Computer Science. 2004; 325:69–110.
8. Doye JPK, Ouldridge TE, Louis AA, Romano F, Sulc P, Matek C, Snodin BEK, Rovigatti L, Schreck JS, Harrison RM, Smith WP. Coarse-graining DNA for simulations of DNA nanotechnology. Physical Chemistry Chemical Physics. 2013; 15:20395–20414. [PubMed: 24121860]

9. Leigh Fanning, M.; Macdonald, Joanne; Stefanovic, Darko. Proceedings of the 2nd ACM Conference on Bioinformatics, Computational Biology and Biomedicine. ACM; 2011. ISO: numeric representation of nucleic acid form; p. 404-408.

10. Feret J, Danos V, Krivine J, Harmer R, Fontana W. Internal coarse-graining of molecular systems. Proceedings of the National Academy of Sciences of the USA. 2009; 106:6453–6458. [PubMed: 19346467]

11. Geary, Cody; Rothemund, Paul WK.; Andersen, Ebbe S. A single-stranded architecture for cotranscrip-tional folding of RNA nanostructures. Science. 2014; 345(6198):799–804. [PubMed: 25124436]

12. Genot, Anthony J.; Yu Zhang, David; Bath, Jonathan; Turberfield, Andrew J. Remote toehold: a mechanism for flexible control of DNA hybridization kinetics. Journal of the American Chemical Society. Feb; 2011 133(7):2177–2182. [PubMed: 21268641]

13. Gillespie D. Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry. 1977; 81(25):2340–2361.

14. Grun, Casey; Sarma, Karthik; Wolfe, Brian; Woo Shin, Seung; Winfree, Erik. A domain-level DNA strand displacement reaction enumerator allowing arbitrary non-pseudoknotted secondary structures. Verification of Engineered Molecular Devices and Programs (VEMDP). 2014

15. Hinton, A.; Kwiatkowska, M.; Norman, G.; Parker, D. PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H.; Palsberg, J., editors. Proc 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), volume 3920 of LNCS. Springer; 2006. p. 441-444.

16. Ibuki, Kawamata; Fumiaki, Tanaka; Masami, Hagiya. Abstraction of DNA graph structures for efficient enumeration and simulation. IPSJ SIG Notes. Jul; 2011 2011(12):1–6.

17. Kawamata, Ibuki; Aubert, Nathanael; Hamano, Masahiro; Hagiya, Masami. Abstraction of graph-based models of bio-molecular reaction systems for efficient simulation. In: Gilbert, David R.; Heiner, Monika, editors. Proceedings, volume 7605 of Lecture Notes in Computer Science; Computational Methods in Systems Biology - 10th International Conference, CMSB 2012; London, UK. October 3-5, 2012; Springer; 2012. p. 187-206.

18. Klavins E. Programmable self-assembly. Control Systems, IEEE. Aug; 2007 27(4):43–56.

19. Klavins E, Ghrist R, Lipsky D. A grammatical approach to self-organizing robotic systems. IEEE Transactions on Automatic Control. Jun; 2006 51(6):949–962.

20. Lakin, Matthew R.; Parker, David; Cardelli, Luca; Kwiatkowska, Marta; Phillips, Andrew. Design and analysis of DNA strand displacement devices using probabilistic model checking. Journal of the Royal Society Interface. 2012; 9(72):1470–1485.

21. Lakin, Matthew R.; Paulevé, Loïc; Phillips, Andrew. Stochastic simulation of multiple process calculi for biology. Theoretical Computer Science. 2012; 431:181–206.

22. Lakin, Matthew R.; Petersen, Rasmus; Gray, Kathryn E.; Phillips, Andrew. Abstract modelling of tethered DNA circuits. In: Murata, Satoshi; Kobayashi, Satoshi, editors. Proceedings of the 20th International Conference on DNA Computing and Molecular Programming, volume 8727 of Lecture Notes in Computer Science. Springer International Publishing; 2014. p. 132-147.

23. Lakin, Matthew R.; Phillips, Andrew. Modelling, simulating and verifying Turing-powerful strand displacement systems. In: Cardelli, Luca; Shih, William, editors. Proceedings of DNA17, volume 6937 of Lecture Notes in Computer Science. Springer-Verlag; 2011. p. 130-144.

24. Lakin, Matthew R.; Youssef, Simon; Cardelli, Luca; Phillips, Andrew. Abstractions for DNA circuit design. Journal of the Royal Society Interface. 2012; 9(68):470–486.

25. Lakin, Matthew R.; Youssef, Simon; Polo, Filippo; Emmott, Stephen; Phillips, Andrew. Visual DSD: a design and analysis tool for DNA strand displacement systems. Bioinformatics. 2011; 27(22):3211–3213. [PubMed: 21984756]

26. Bingling, Li; Ellington, Andrew D.; Chen, Xi. Rational, modular adaptation of enzyme-free DNA circuits to multiple detection methods. Nucleic Acids Research. 2011; 39(16):e110. [PubMed: 21693555]

27. Mazur, Alexey K. Wormlike chain theory and bending of short DNA. Physical Review Letters. 2007; 98:218102. [PubMed: 17677812]

28. McCaskill, John S.; Niemann, Ulrich. Graph replacement chemistry for DNA processing. In: Condon, Anne; Rozenberg, Grzegorz, editors. DNA Computing, volume 2054 of Lecture Notes in Computer Science. Springer; Berlin Heidelberg: 2001. p. 103-116.

29. Milner, Robin. Communicating and mobile systems - the Pi-calculus. Cambridge University Press; 1999.

30. Milner, Robin; Parrow, Joachim; Walker, David. A calculus of mobile processes, I. Inf Comput. 1992; 100(1):1–40.

31. Mokhtar, Reem; Garg, Sudhanshu; Chandran, Harish; Bui, Hieu; Song, Tianqi; Reif, John. A graph rewriting system for modeling DNA nanodevices. DNA. 2013

32. Oury, Nicolas; Pedersen, Michael; Petersen, Rasmus. Canonical labelling of site graphs. In: Petre, Ion, editor. Electronic Proceedings in Theoretical Computer Science; Proceedings Fourth International Workshop on Computational Models for Cell Processes; Turku, Finland. 11th June 2013; Open Publishing Association; 2013. p. 13-28.

33. Phillips, Andrew; Cardelli, Luca. A programming language for composable DNA circuits. Journal of the Royal Society Interface. Aug; 2009 6(S4):419–436.

34. Qian, Lulu; Soloveichik, David; Winfree, Erik. Efficient Turing-universal computation with DNA polymers. In: Sakakibara, Yasubumi; Mi, Yongli, editors. Proceedings of DNA16, volume 6518 of Lecture Notes in Computer Science. Springer-Verlag; 2011. p. 123-140.

35. Qian, Lulu; Winfree, Erik. Scaling up digital circuit computation with DNA strand displacement cascades. Science. 2011; 332:1196–1201. [PubMed: 21636773]

36. Qian, Lulu; Winfree, Erik. Parallel and scalable computation and spatial dynamics with dna-based chemical reaction networks on a surface. In: Murata, Satoshi; Kobayashi, Satoshi, editors. Proceedings of the 20th International Conference on DNA Computing and Molecular Programming, volume 8727 of Lecture Notes in Computer Science. Springer International Publishing; 2014. p. 114-131.

37. Qian, Lulu; Winfree, Erik; Bruck, Jehoshua. Neural network computation with DNA strand displacement cascades. Nature. 2011; 475:368–372. [PubMed: 21776082]

38. Rothemund, Paul WK. Folding DNA to create nanoscale shapes and patterns. Nature. 2006; 440:297–302. [PubMed: 16541064]

39. Rudchenko, Maria; Taylor, Steven; Pallavi, Payal; Dechkovskaia, Alesia; Khan, Safana; Butler, Vincent P., Jr; Rudchenko, Sergei; Stojanovic, Milan N. Autonomous molecular cascades for evaluation of cell surfaces. Nature Nanotechnology. 2013; 8:580–586.

40. Schaeffer, JM. Master's thesis. California: Institute of Technology; 2012. Stochastic simulation of the kinetics of multiple interacting nucleic acid strands.

41. Seelig, Georg; Soloveichik, David; Yu Zhang, David; Winfree, Erik. Enzyme-free nucleic acid logic circuits. Science. 2006; 314:1585–1588. [PubMed: 17158324]

42. Soloveichik D, Seelig G, Winfree E. DNA as a universal substrate for chemical kinetics. Proceedings of the National Academy of Sciences of the USA. 2010; 107(12):5393–5398. [PubMed: 20203007]

43. Srinivas, Niranjan; Ouldridge, Thomas E.; Sulc, Petr; Schaeffer, Joseph M.; Yurke, Bernard; Louis, Ard A.; Doye, Jonathan PK.; Winfree, Erik. On the biophysics and kinetics of toehold-mediated dna strand displacement. Nucleic Acids Res. Sep.2013

44. Teichmann, Mario; Kopperger, Enzo; Simmel, Friedrich C. Robustness of localized DNA strand displacement cascades. ACS Nano. 2014; 8(8):8487–8496. [PubMed: 25089925]

45. Yin, Peng; Choi, Harry MT.; Calvert, Colby R.; Pierce, Niles A. Programming biomolecular self-assembly pathways. Nature. 2008; 451:318–322. [PubMed: 18202654]

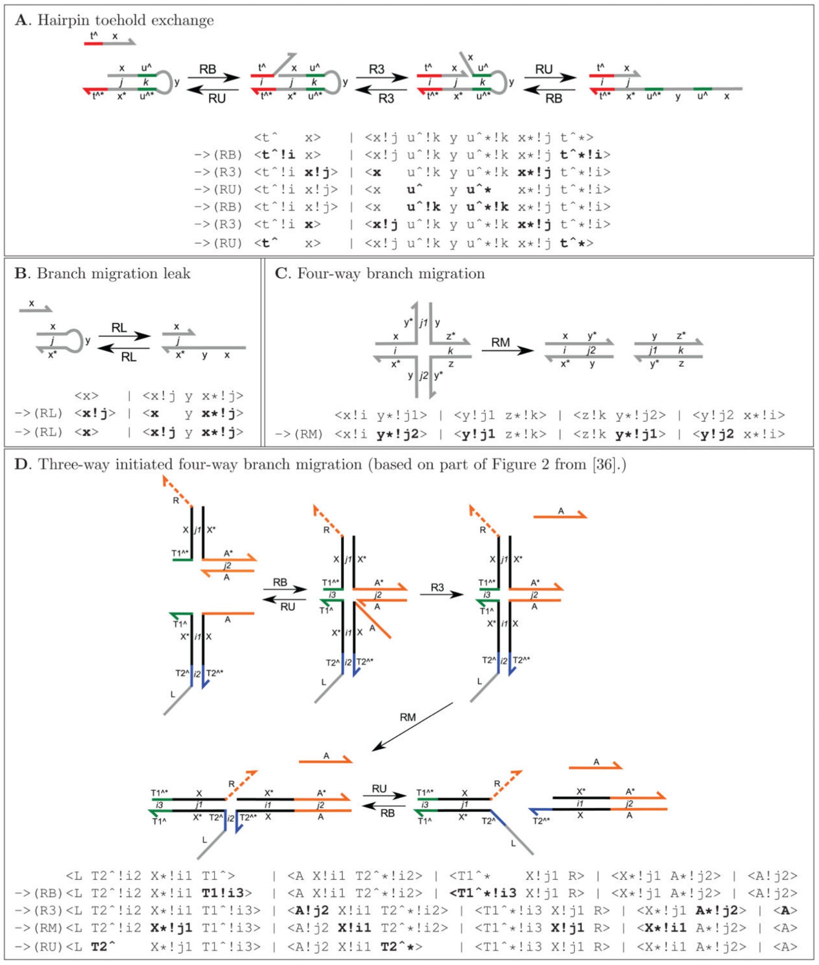
46. Yordanov, Boyan; Kim, Jongmin; Petersen, Rasmus L.; Shudy, Angelina; Kulkarni, Vishwesh V.; Phillips, Andrew. Computational design of nucleic acid feedback control circuits. ACS Synthetic Biology. 2014; 3(8):600–616. [PubMed: 25061797]

47. Yordanov, Boyan; Wintersteiger, Christoph M.; Hamadi, Youssef; Phillips, Andrew; Kugler, Hillel. Proceedings of DNA19, volume 8141 of Lecture Notes in Computer Science. Springer-Verlag; 2013. Functional analysis of large-scale DNA strand displacement circuits. In David Soloveichik and Bernard Yurke, editors; p. 189-203.

48. Zadeh JN, Steenberg CD, Bois JS, Wolfe BR, Pierce MB, Khan AR, Dirks RM, Pierce NA. NUPACK: analysis and design of nucleic acid systems. Journal of Computational Chemistry. 2011; 32:170–173. [PubMed: 20645303]

49. Yu Zhang, David; Seelig, Georg. Dynamic DNA nanotechnology using strand-displacement reactions. Nature Chemistry. 2011; 3:103–113.

50. Yu Zhang, David; Turberfield, Andrew J.; Yurke, Bernard; Winfree, Erik. Engineering entropy-driven reactions and networks catalyzed by DNA. Science. 2007; 318:1121–1125. [PubMed: 18006742]



A. Hairpin toehold exchange

```
        <t^   x>   | <x!j u^!k y u^*!k x*!j t^*>
->(RB)  <t^!i x>   | <x!j u^!k y u^*!k x*!j t^*!i>
->(R3)  <t^!i x!j> | <x   u^!k y u^*!k x*!j t^*!i>
->(RU)  <t^!i x!j> | <x   u^   y u^*   x*!j t^*!i>
->(RB)  <t^!i x!j> | <x   u^!k y u^*!k x*!j t^*!i>
->(R3)  <t^!i x>   | <x!j u^!k y u^*!k x*!j t^*!i>
->(RU)  <t^   x>   | <x!j u^!k y u^*!k x*!j t^*>
```

B. Branch migration leak

```
        <x>   | <x!j y x*!j>
->(RL)  <x!j> | <x   y x*!j>
->(RL)  <x>   | <x!j y x*!j>
```

C. Four-way branch migration

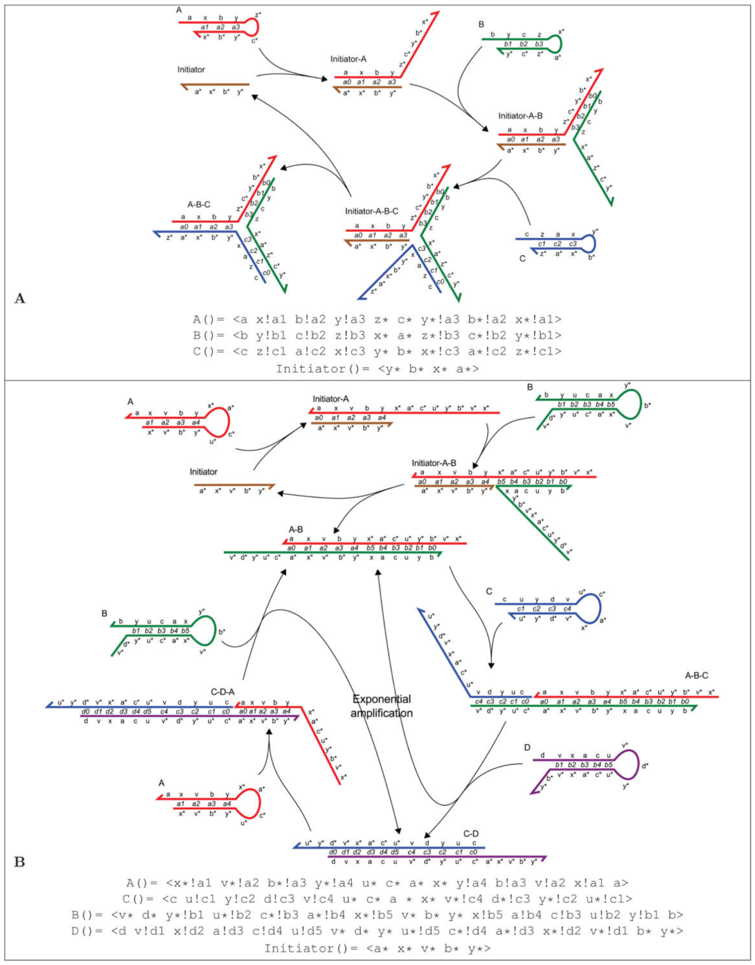
```
        <x!i y*!j1> | <y!j1 z*!k> | <z!k y*!j2> | <y!j2 x*!i>
->(RM)  <x!i y*!j2> | <y!j1 z*!k> | <z!k y*!j1> | <y!j2 x*!i>
```

D. Three-way initiated four-way branch migration (based on part of Figure 2 from [36].)

```
      <L T2^!i2 X*!i1 T1^>     | <A X!i1 T2^*!i2> | <T1^*     X!j1 R> | <X*!j1 A*!j2> | <A!j2>
->(RB)<L T2^!i2 X*!i1 T1!i3>   | <A X!i1 T2^*!i2> | <T1^*!i3 X!j1 R> | <X*!j1 A*!j2> | <A!j2>
->(R3)<L T2^!i2 X*!i1 T1^!i3> | <A!j2 X!i1 T2^*!i2> | <T1^*!i3 X!j1 R> | <X*!j1 A*!j2> | <A>
->(RM)<L T2^!i2 X*!j1 T1^!i3> | <A!j2 X!i1 T2^*!i2> | <T1^*!i3 X!j1 R> | <X*!i1 A*!j2> | <A>
->(RU)<L T2^    X*!j1 T1^!i3> | <A!j2 X!i1 T2^*>    | <T1^*!i3 X!j1 R> | <X*!i1 A*!j2> | <A>
```

Figure 1.

State space for a collection of simple examples. There is a one-to-one correspondence between the program code and the graphical representation. For each example, the first line of code denotes the inital state, which is represented graphically on the left. Importantly, only the initial state needs to be written explicitly by the user, since the remaining states are generated automatically using the reduction rules of Definition 2.

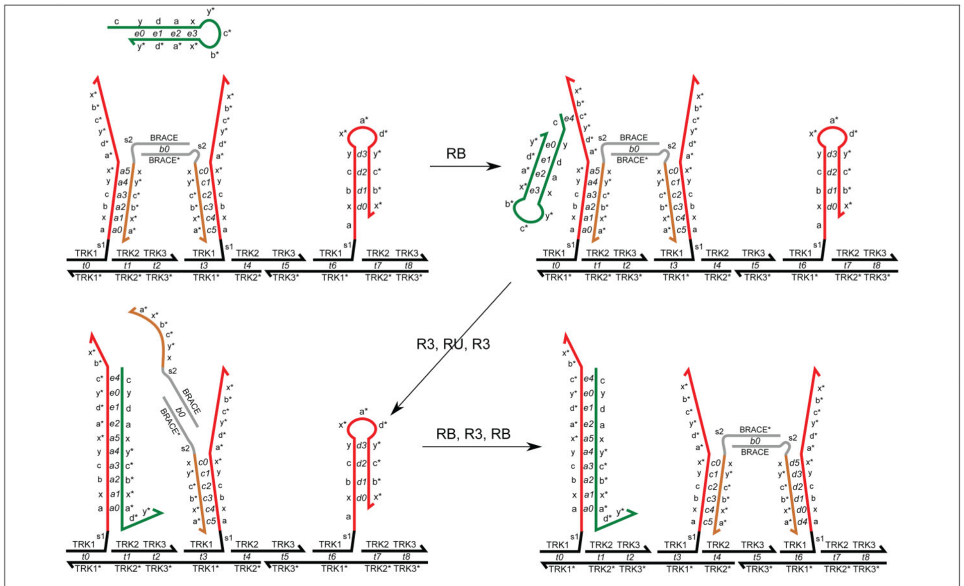


```
A()= <a x!a1 b!a2 y!a3 z* c* y*!a3 b*!a2 x*!a1>
B()= <b y!b1 c!b2 z!b3 x* a* z*!b3 c*!b2 y*!b1>
C()= <c z!c1 a!c2 x!c3 y* b* x*!c3 a*!c2 z*!c1>
Initiator()= <y* b* x* a*>
```

```
A()= <x*!a1 v*!a2 b*!a3 y*!a4 u* c* a* x* y!a4 b!a3 v!a2 x!a1 a>
C()= <c u!c1 y!c2 d!c3 v!c4 u* c* a * x* v*!c4 d*!c3 y*!c2 u*!c1>
B()= <v* d* y*!b1 u*!b2 c*!b3 a*!b4 x*!b5 v* b* y* x!b5 a!b4 c!b3 u!b2 y!b1 b>
D()= <d v!d1 x!d2 a!d3 c!d4 u!d5 v* d* y* u*!d5 c*!d4 a*!d3 x*!d2 v*!d1 b* y*>
Initiator()= <a* x* v* b* y*>
```

Figure 2.

CRN for a catalytic three-armed junction (A, from Figure 2 of [45]) and for a cross-catalytic amplifier (B, from Figure 3 of [45]). The program code for the initial species is written by the user and is shown explicitly. The full CRN is generated automatically from the initial species, according to Definition 3.



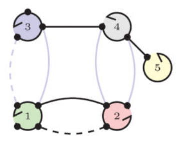
```
Fuel() = <c y!e0 d!e1 a!e2 x!e3 y* c* b* x*!e3 a*!e2 d*!e1 y*!e0>
TrackState1() = <TRK1!t0 s1 a!a0 x!a1 b!a2 c!a3 y!a4 x*!a5 a* d* y* c* b* x*>
  | <TRK2!t1 TRK3!t2> | <TRK1!t3 s1 a!c5 x!c4 b!c3 c!c2 y!c1 x*!c0 a* d* y* c* b* x*>
  | <TRK2!t4 TRK3!t5> | <TRK1!t6 s1 a x!d0 b!d1 c!d2 y!d3 x* a* d* y*!d3 c*!d2 b*!d1 x*!d0>
  | <TRK2!t7 TRK3!t8> | <TRK3*!t8 TRK2*!t7 TRK1*!t6 TRK3*!t5>
  | <TRK2*!t4 TRK1*!t3 TRK3*!t2 TRK2*!t1 TRK1*!t0>
  | <BRACE!b0 s2 x!a5 y*!a4 c*!a3 b*!a2 x*!a1 a*!a0>
  | <BRACE*!b0 s2 x!c0 y*!c1 c*!c2 b*!c3 x*!c4 a*!c5>
```

Figure 3.

Partial set of chemical reactions for the bipedal walker example from Figure 5 of [45].

$$\begin{aligned}
V &= \{1,2,3,4,5\}\\
\mathit{length} &= \{1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 1\}\\
\mathit{colour} &= \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5\}\\
A &= \{\{(1,2),(2,3)\},\{(1,3),(2,2)\},\{(1,3),(3,2)\},\\
&\quad \{(1,4),(3,1)\},\{(2,1),(4,2)\},\{(2,1),(4,1)\},\\
&\quad \{(3,2),(4,1)\},\{(4,2),(5,1)\}\}\\
\mathit{toehold} &= \{\{(1,2),(2,3)\} \mapsto \text{true}, \{(1,4),(3,1)\} \mapsto \text{true},\\
&\quad \mathit{other} \mapsto \text{false}\}\\
E &= \{\{(1,2),(2,3)\},\{(1,3),(2,2)\},\{(3,2),(4,1)\}\}
\end{aligned}$$

A



B

```
① <L T2^!i2 X*!i1 T1^>
② <A X!i1 T2^*!i2>
③ <T1^* X*!j1 R>
④ <X*!j1 A*!j2>
⑤ <A!j2>
```

C

Figure 4.

Textual representation (A) and graphical depiction (B) of a strand graph corresponding to the start state of Fig. 1D. The correspondence between the calculus and strand grand representations is illustrated in (C), where each calculus strand is shown next to its colour.

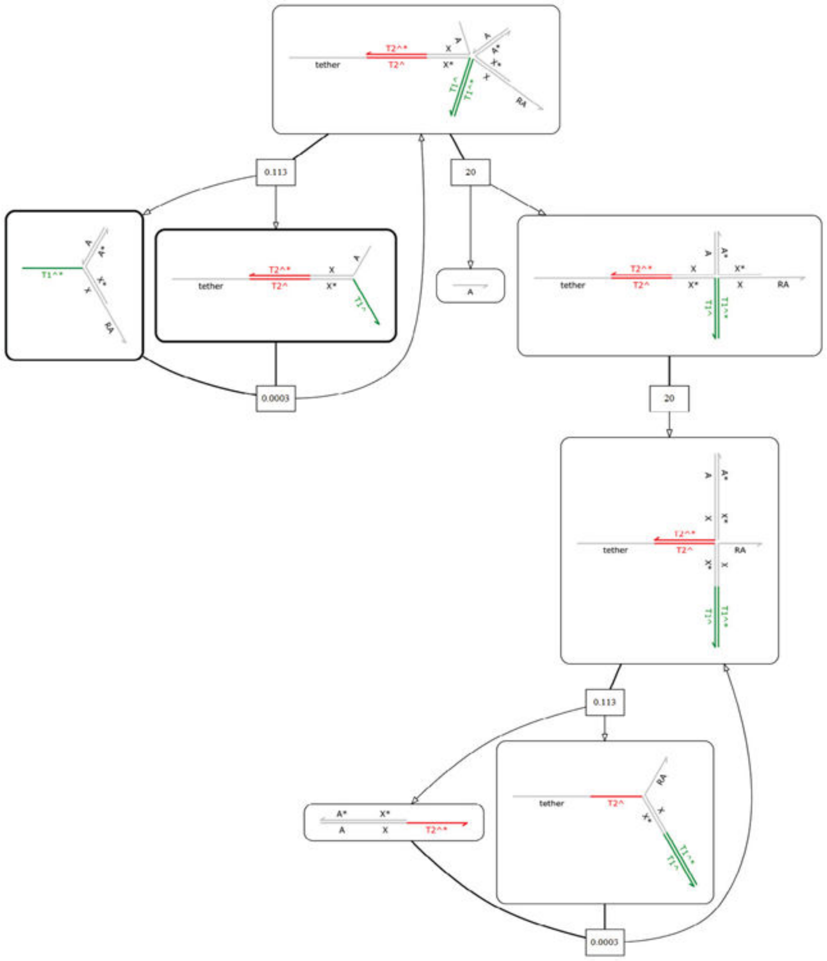


Figure 5.

Inferred CRN for three-way initiated four-way branch migration. Thick lines go between reactant species and reactions, and thin arrows go from reactions to product species.

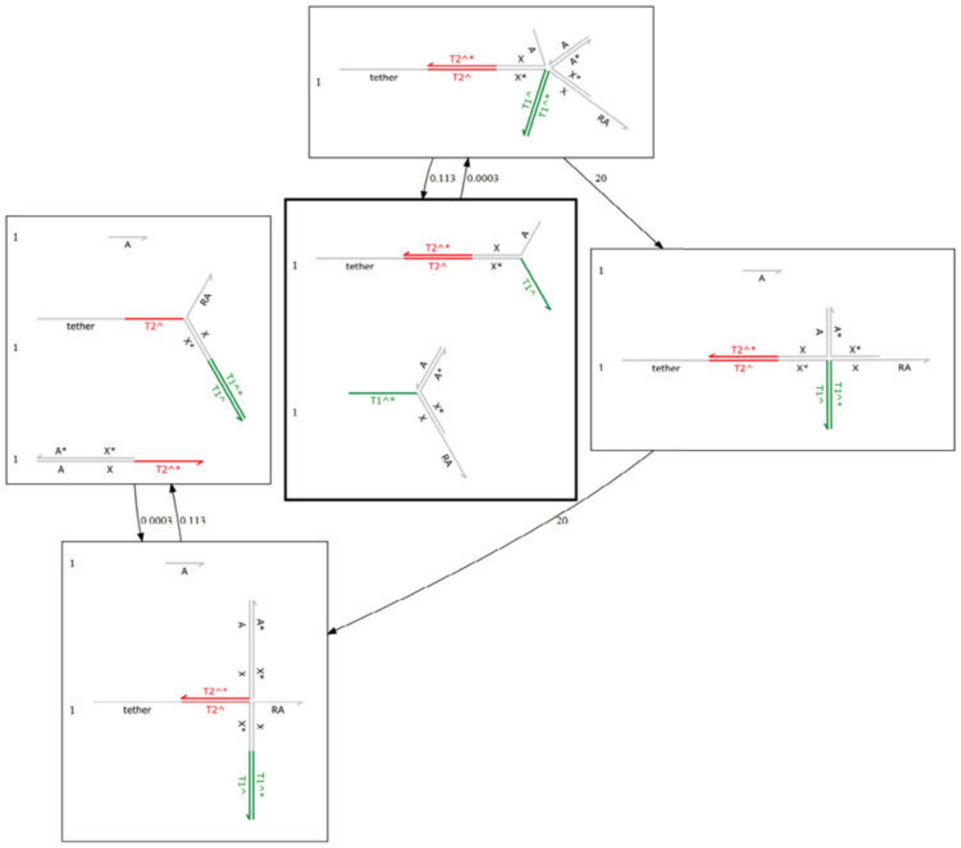


Figure 6.
Inferred state space for three-way initiated four-way branch migration. State transitions are marked by arrows, which are labelled with the rate of the reaction that would incur the transition.

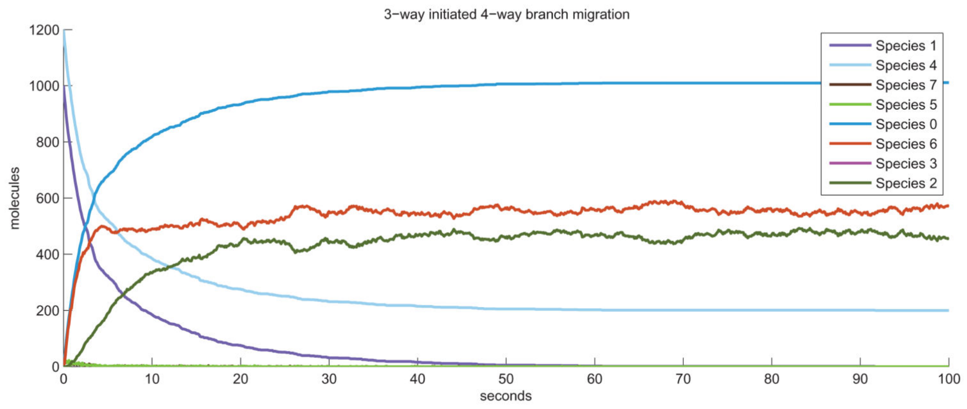


Figure 7.
Time course from a stochastic simulation of the three-way initiated four-way branch migration. The legend refers to the species from Figure 5.

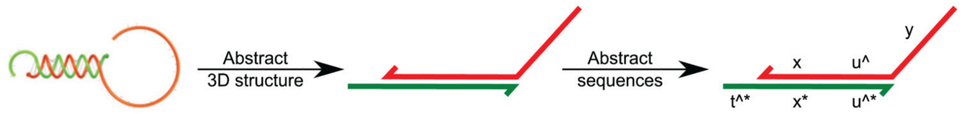


Figure 8.

DNA structure and sequence abstractions. We abstract away from the 3D atomic structure of the DNA strands to produce a simple 2D representation. We then subdivide the DNA strand sequences into domains, which allows us to design structures at a higher level of abstraction than the individual nucleotide sequences.

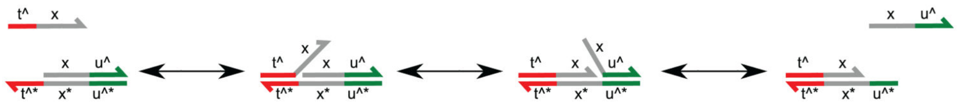


Figure 9.
DNA strand displacement example reaction. The input strand binds reversibly to the gate and initiates a reversible branch migration reaction across the domain *x* that is shared between the input strand and the strand already bound to the gate. When the branch migration reaction reaches the far end of the *x* domain, the other strand originally bound to the gate is now only bound via the complementary *u*̂ toehold domain. Thus, it can reversibly unbind from the gate, leaving the input strand bound to the gate, the previously bound strand free in solution, and the previously bound toehold domain *u*̂ exposed on the gate. We refer to this process as a “strand exchange” reaction.