ACS Synthetic Biology
Research Article
Cite This: ACS Synth. Biol. 2019, 8, 1530−1547
pubs.acs.org/synthbio

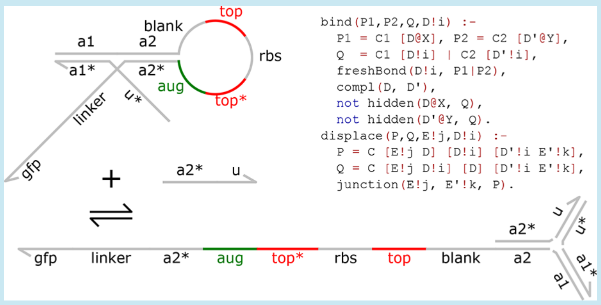
A Logic Programming Language for Computational Nucleic Acid Devices

Carlo Spaccasassi,[†] Matthew R. Lakin,[‡,§] and Andrew Phillips*[,†]

[†]Microsoft Research, Cambridge, CB1 2FB, U.K.
[‡]Department of Computer Science, University of New Mexico, Albuquerque, New Mexico 87131, United States
[§]Center for Biomedical Engineering, University of New Mexico, Albuquerque, New Mexico 87131, United States

Supporting Information

ABSTRACT: Computational nucleic acid devices show great potential for enabling a broad range of biotechnology applications, including smart probes for molecular biology research, *in vitro* assembly of complex compounds, high-precision *in vitro* disease diagnosis and, ultimately, computational theranostics inside living cells. This diversity of applications is supported by a range of implementation strategies, including nucleic acid strand displacement, localization to substrates, and the use of enzymes with polymerase, nickase, and exonuclease functionality. However, existing computational design tools are unable to account for these strategies in a unified manner. This paper presents a logic programming language that allows a broad range of computational nucleic acid systems to be designed and analyzed. The language extends standard logic programming with a novel equational theory to express nucleic acid molecular motifs. It automatically identifies matching motifs present in the full system, in order to apply a specified transformation expressed as a logical rule. The language supports the definition of logic predicates, which provide constraints that need to be satisfied in order for a given rule to be applied. The language is sufficiently expressive to encode the semantics of nucleic strand displacement systems with complex topologies, together with computation performed by a broad range of enzymes, and is readily extensible to new implementation strategies. Our approach lays the foundation for a unifying framework for the design of computational nucleic acid devices.



```
bind(P1,P2,Q,D!i) :-
  P1 = C1 [D@X], P2 = C2 [D'@Y],
  Q  = C1 [D!i] | C2 [D'!i],
  freshBond(D!i, P1|P2),
  compl(D, D'),
  not hidden(D@X, Q),
  not hidden(D'@Y, Q).
displace(P,Q,E!j,D!i) :-
  P = C [E!j D] [D!i] [D'!i E'!k],
  Q = C [E!j D!i] [D] [D'!i E'!k],
  junction(E!j, E'!k, P).
```

KEYWORDS: *strand graph, site graph, process calculus, programming language, DNA computing, molecular programming, biological computation, logic programming*

Devices made of nucleic acids that are capable of performing computation are an active area of research, and also demonstrate strong potential for real-world applications. Such applications include smart probes for molecular biology research,[1,2] *in vitro* assembly and manufacturing of complex compounds,[3,4] high-precision *in vitro* diagnosis of disease[5−8] and, ultimately, computational theranostics inside living cells.[9,10] This diversity of applications has been supported by a range of nucleic acid implementation strategies for carrying out information processing at the nanoscale. These include implementations based solely on nucleic acid hybridization and strand displacement,[11−16] and those that combine nucleic acid strand displacement with localization to substrates.[17−19] Alternative implementation strategies incorporate enzymatic reactions via the Polymerase-Exonuclease-Nickase (PEN) DNA toolbox,[20−23] transcriptionally encoded computation via Genelet systems,[24,25] and Ribocomputing logic circuits that combine complex nucleic acid topologies with translational machinery.[5,7]

As the complexity of nucleic acid computational devices continues to increase, modeling methods and their corresponding software implementations are beginning to play an important role in the design of such devices. In particular, the Visual DSD language was developed to model DNA strand displacement systems,[26] and subsequent versions incorporated a hierarchy of behavioral abstractions,[27,28] custom reactions,[29] localized interactions,[30] and complex topologies.[31] Alternative methods and tools for generating the behavior of DNA strand displacement systems have also been developed, including Peppercorn[32] and the DyNAMiC Workbench.[33] Methods have also been developed for the computational design of PEN-DNA toolbox systems,[34] together with strategies for automating their implementation.[35] However, existing computational design tools are unable to account for the current range of nucleic acid implementation strategies in a unified manner, or to readily incorporate new implementation strategies.

In this paper we present a logic programming language for designing computational devices made of nucleic acids. Logic programming is often summarized by the slogan "Algorithm = Logic + Control".[36] In other paradigms such as imperative and functional programming, an algorithm describes the control flow

Special Issue: IWBDA 2017

Received: May 29, 2018
Published: October 29, 2018

and operations that compute the desired output given some input. In contrast, in logic programming the output is described in terms of logic formulas (Logic) and inputs are defined in terms of logical properties. Verifying that an input conforms to this specification is the task of an accompanying logical inference system, which computes a solution to the formulas in the process (Control). This division of concerns shifts the burden of programming to the inference system and lets programmers focus on the formal definition of the problem. Logic programming provides a powerful declarative framework in which complex logic can be expressed in a clean and concise manner. This allows users to construct custom extensions that implement new hypotheses required for the specific system being designed.

The syntax of our language is formally defined in the style of process calculi and combines a syntax for nucleic acid strands, which extends previous work on Strand graphs,[31] with a syntax for logic programs, based on Prolog. The semantics follows the *unification modulo equational theory* approach: it adopts the standard semantics of Prolog, plus a modular extension to its unification algorithm by a novel equational theory of DNA strands. A corresponding implementation of our language has been integrated with the Visual DSD system. Our logic programming language can encode previous extensions to the Visual DSD language,[27,29−31] as well as new extensions including enzyme interactions[20] and the encoding of kinetic rate hypotheses,[15] neither of which were previously supported in Visual DSD. More importantly, our approach is extensible in that new nucleic acid implementation strategies can be encoded simply by defining new logic predicates. Thus, our work provides a framework to model the largest set of nucleic acid information processing systems to date.

In the remainder of the paper, we first present the underlying logic inference system of our language, together with its equational theory in terms of processes, patterns, and contexts, using an encoding of DNA strand displacement predicates as an example. We then demonstrate how our language can encode an abstraction hierarchy of behaviors by merging fast reactions, and show how hypotheses for assigning kinetic parameters to reactions can be defined as logical predicates. This allows context-specific hypotheses about rate constants to be expressed, which can also be used to guide parameter inference. We then use our logic programming language to encode nucleic acid implementation strategies that make use of polymerase, exonoclease, and nickase enzymes. These general rules greatly simplify the encoding of enzyme-based systems, for example by avoiding the need to manually encode each enzyme operation, for all possible species. We also encode implementation strategies that rely on localization to a substrate. Finally, we demonstrate how complex nucleic acid topologies can be encoded, including arbitrary secondary structures at the domain level[31] such as branches and pseudoknots, and combined with translational machinery to perform computation.[5]

■ RESULTS

Logic Inference System. We developed an inference system for our logic programming language based on SLDNF resolution, which is widely used in logic programming languages such as Prolog.[36,37] We briefly summarize the semantics of SLDNF resolution using a simple logic program based on the well-known Aristotelian syllogism *"Socrates is human. All humans are mortal. Therefore, Socrates is mortal."*

```
human("Socrates").
mortal(X) :- human(X).
```

This program has two *Horn clauses*: a *fact* stating that Socrates is human, and a *clause* stating that for any *X*, if *X* is human then *X* is also mortal. SLDNF resolution provides a logically sound algorithm to verify that an atomic predicate *A*, also called *goal*, is the logical consequence of a logic program. For our example, to know whether Socrates is mortal we can query the system with the goal `mortal ("Socrates")`. SLDNF resolution then attempts to verify the goal using all declared facts and clauses in the logic program. In this case the verification succeeds, since the goal is indeed a consequence of the program. If the goal also contains logical variables, SLDNF resolution finds all possible instantiations of the variables such that the goal succeeds. For our example, the system can find all mortals with the goal `mortal(X)`. In this case the only solution is `X = "Socrates"`.

The matching of predicates and terms in SLDNF resolution is performed by a procedure called *unification*. This is a constraint satisfaction algorithm that works on sets of equality constraints of the form $t_1 \doteq t_2$, and finds an assignment θ of logical variables in terms t_1 and t_2 such that the two terms become identical, with $\theta(t_1) = \theta(t_2)$. It can be viewed as a generalized form of pattern matching. Unification works by iteratively decomposing equalities over composite terms into equalities over their components. For example, the equality $mortal(\text{“}Socrates\text{”}) \doteq mortal(X)$ is decomposed into an equality over predicate names, $mortal \doteq mortal$, and an equality over the arguments, $\text{“}Socrates\text{”} \doteq X$. The first equality is trivially true, while the second equality is solved by the assignment $\theta(X) = \text{“}Socrates\text{”}$. Unification succeeds with a solution θ when all of the equality constraints are satisfied.

Our programming language extends the standard SLDNF resolution algorithm (see Methods), with three main modifications. First, we simplified the treatment of negative predicates to avoid a consistency problem with standard SLDNF known as *floundering*. Floundering occurs when trying to prove a negative predicate containing free variables. Such a predicate does not have a finitely failed tree because it is indeterminate, therefore forcing SLDNF to fail such cases can result in inconsistencies.[36] Unfortunately, floundering is an undecidable property that cannot be verified statically. To circumvent this issue, during program execution we check that negative predicates are fully instantiated and therefore have no free variables. An example of a fully instantiated negative predicate for our logic program is `not mortal ("stone").` Second, we modified the *resolution strategy*, which determines the next node in the search tree to expand during resolution. Standard SLDNF typically aims to find a single solution to a goal in the fastest way possible, and therefore uses a depth-first search (DFS) strategy. However, our language is used to fully explore the possible behaviors of a system, rather than finding a single behavior, so our resolution strategy employs a breadth-first search (BFS) strategy. This has stronger guarantees than DFS since it is *complete*, in the sense that a solution is always found if one exists.[38] As a consequence, the order in which clauses are written in a program is not important, unlike other logic programming languages such as Prolog. Also, since our search strategy is BFS, we do not support extra-logical operators such as cut (!) to curb backtracking during solution finding. Third and most importantly, our language extends the standard unification algorithm with a novel equational theory of nucleic acid strands, using the *unification modulo equational theory* approach. Our equational theory defines *contexts* and *patterns* that allow the identification and sound manipulation of general nucleic acid motifs. The soundness of our method rests on the Double-Push Out approach of graph grammar theory.[38,39] The following section presents the equational theory of our language though an example.

Nucleic Acid Equational Theory. We present the equational theory of our logic programming language by using it to encode the elementary rules of *DNA strand displacement*,[11] whose systematic use was pioneered in ref 40. DNA strand displacement involves an invading single strand of DNA displacing an incumbent strand, hybridized to a template strand. The process is mediated by a short, single-stranded region of DNA referred to as a *toehold*. It can implement a broad range of computation, including any computation that can be expressed as an abstract chemical reaction network,[12] such as oscillations, switches, population protocols, combinatorial logic and sequential logic. Chemical reaction networks are known to be Turing complete with an arbitrarily small probability of error,[41] due to the finite number of species involved. However, DNA strand displacement systems can form potentially unbounded polymers, and are therefore more expressive in that they have been shown to be Turing powerful.[42]

We represent a nucleic acid system in our language as a *process*, which is defined as a multiset of nucleic acid strands (Table 1). The basic abstraction of our language is the *domain*,

Table 1. Syntax of Processes[a]

d::=	$x \mid x^* \mid x^\wedge \mid x^{\wedge *} \mid X$	domain
k::=	$x \mid int \mid X$	bond, $int \geq 0$
t::=	$int \mid string \mid x \mid x(T_1, ..., T_N)$	tag, $N \geq 1$, $int \geq 0$
l::=	$x \mid int \mid X$	location, $int \geq 0$
s::=	$d\{t\}\ @l \mid d\{t\}\ !\ k@l \mid X$	site, with$\{t\}$ and $@l$ optional
S::=	$s_1 ... s_N$	sequence of sites, $N \geq 1$
P::=	$<S_1> \vert \cdots \vert <S_N>$	process, $N \geq 0$

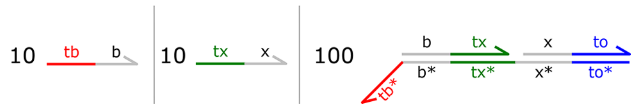
[a]The syntax of processes in our language is based on the syntax of strand graph processes,[31] extended with *logical variables*, *tags*, and *locations*, for which x denotes a lower-case name and *int* denotes an integer. Logical variables X are written in upper case and are replaced with concrete values during logical inference. Tags t model various properties of a domain, such as chemical modifications or tethering to a substrate. Locations l are unique identifiers that pinpoint the occurrence of a particular domain in a process, and are used to distinguish between multiple occurrences of the same domain name. A process is *well-formed* if each bond k occurs exactly twice in the process, and only between complementary domains. Processes are composed using the parallel composition operator (|), which indicates that the order in which processes are specified is not significant. Note that this is distinct from the short vertical bar ($\mid$), which is used in programming language definitions to denote a set of possible grammatical symbols. We consider processes equal up to reordering of strands and renaming of bonds.

which represents a unique nucleic acid sequence. We assume that a domain can bind to its complement but cannot interact with any other domains in the system. In practice, this is achieved by ensuring that distinct domains use noninterfering nucleic acid sequences, for instance by relying on appropriate coding strategies.[13] We represent a domain with a lower-case variable and annotate complementary domains with a star, where x^* denotes the Watson−Crick complement of domain x. Toeholds are domains that are assumed to be short enough to spontaneously unbind from their complement. A toehold is labeled with a caret, written $x^\wedge$, where the complement of $x^\wedge$ is $x^{\wedge *}$. We indicate that two domains x and x^* are *bound* using the notation $x!i$ and $x^*!i$, where i is a unique identifier called a *bond*,

which can be either a variable name or an integer. We define a *site* s as a domain that is either bound or free, and a *sequence* S as a nonempty sequence of sites, ordered from the 5′ end to the 3′ end. A process P is a multiset of strands $<S_1>$ | ... | $<S_N>$, separated by the parallel composition operator (|), where each strand is a sequence enclosed in angle brackets. In addition, we define a *species* as a set of strands bound to each other such that they form a connected component. For convenience, we define additional syntactic sugar that allows a species to be enclosed in square brackets and preceded by a constant, which denotes the species population.

For example, consider the following process, which relies on DNA strand displacement to compute a join operation. A corresponding graphical representation of the process is also shown.

```
( 10 [<tb^ b>]
| 10 [<tx^ x>]
| 100 [ <to^*!1 x*!2 tx^*!3 b*!4 tb^*>
      | <b!4 tx^!3> | <x!2 to^!1> ] )
```



The process consists of a multiset of species, separated by the parallel composition operator, where each species is enclosed in square brackets and preceded by its population. The first species is a single strand `<tb^ b>`, consisting of a toehold `tb^` followed by a domain `b`. Similarly, the second species is a single strand `<tx^ x>`. The third species is a complex consisting of a strand `<to^*!1 x*!2 tx^*!3 b*!4 tb^*>` bound to two shorter strands `<x!2 to^!1>` and `<b!4 tx ^ ! 3>`. The bonds are omitted in the corresponding graphical representation, since they are only used to determine connectivity.

In addition to specifying the initial conditions of a system as a process, our language allows logic predicates to be defined in order to automatically generate system behavior. This is achieved by extending the syntax of processes with *logical variables* X (Table 1), where the *wildcard* "_" is the logical variable that matches any term. Logical variables can then be combined with *patterns* π (Table 2) to match a specific part of a

Table 2. Syntax of Patterns and Contexts[a]

π::=	$<S> \mid <S \mid S \mid S> \mid S_1> \vert <S_2 \mid \varnothing$	pattern
C_N::=	$[\cdot]_i \mid P \mid < S\ C_N \mid S\ C_N \mid C_N\ S> \mid C_N \vert C_N$	context with N holes, $1 \leq i \leq N$

[a]A *pattern* π represents a specific motif that may occur in a process. A *context* C_N is a "process with N holes",[43] where each hole $[\cdot]_i$ in C_N is associated with a number $i \in N$. A context C_N is *well-formed* if it contains exactly N holes and each hole $[\cdot]_i$ occurs exactly once. We only consider well-formed contexts.

process. The pattern $<S>$ matches a strand with exactly sequence S, while the pattern $<S$ matches a strand with sequence S at its 5′ end, and the pattern $S>$ matches a strand with sequence S at its 3′ end. The pattern S matches a sequence that can be present anywhere in a process, and the pattern $S_1>$ | $<S_2$ matches a nick between two adjacent strands, where $S_1>$ and $<S_2$ represent the two ends of the strands where the nick occurs. For example, the pattern `a!1>` | `<b!2` matches the nick in the double stranded complex `<d a!1>` | `<b*!2 a*!1>` | `<b!2 c>`. Note that the order in which strands are written in a complex is not significant for pattern matching, since processes are identified up to reordering of strands. In general, the pattern $S_1>$ | $<S_2$ matches

Table 3. Syntax of Logic Programs[a]

$T ::=$	$X \mid int \mid float \mid string \mid \pi \mid C_N[\pi_1]...[\pi_N] \mid X[\pi_1]...[\pi_N]$	term
$\mid$	$x(T_1, ..., T_N) \mid [T_1;...; T_N] \mid [T_1;...; T_N \# X]$	
$A ::=$	$x(T_1, ..., T_N)$	atomic predicate
$L ::=$	$A \mid$ `not` A	literal
$H ::=$	$A{:}-\ L_1, ..., L_N$	horn clause

[a]The syntax of our logic programming language extends the standard syntax of Prolog with the patterns and contexts defined in Table 2. As in Prolog, the main data structure is a *term* T, which can be an integer *int*, floating point number *float*, a *string* enclosed in double quotes, or a logical variable X. In addition, a term can be a pattern π or a context C_N applied to N patterns, written $C_N[\pi_1]...[\pi_N]$. The context itself can also be a logical variable X, written $X\ [\pi_i]...[\pi_N]$, which is used to match any process that contains the patterns $\pi_1...\pi_N$. The *structure* $x(T_1...T_N)$ bundles together N terms under an identifier x and defines named compound terms such as $compl(d_1, d_2)$. The list $[T_1;\cdots;T_N]$ denotes a possibly empty list of terms T_i. It is also possible to define lists of undetermined length using the syntax $[T_1;\cdots;\ T_N\#X]$, where the logical variable X stands for the rest of the list. An *atomic predicate* or *atom* A has syntax $x(t_1,...t_N)$, where x is an identifier. It is generally used to describe properties or assign relationships to terms. We assume that the sets of predicate identifiers and structure identifiers are disjoint. A *literal* L is either an atom A or a *negative atom* `not` A. A *Horn clause* or *clause* H has syntax $A{:}-\ L_1,...,L_N$, and can be read "A holds *if* $L_1...L_N$ hold". The list of literals L_i is possibly empty. Atom A is called the *head* of H, and the list of literals is its *body*. A clause with no literals is also known as a *fact*. Finally, a *logic program* is a set of Horn clauses.

any two strands in a process such that the 3′ end of one strand matches S_1, while the 5′ end of the other strand matches S_2. This pattern does not require the strands to be adjacent to each other or bound to a common strand, though this constraint can be encoded explicitly for nicking enzymes (Figure 4). The empty pattern ∅ does not match any strand, and is used to model the creation and deletion of strands.

A pattern π is matched with a process P using the notion of a *context* C_N (Table 2), defined as a "process with N holes",[43] where each hole $[\cdot]_i$ in C_N is associated with a number $i \in N$. The matching is performed by *applying* a context C_N to patterns $\pi_1...\pi_N$, written $C_N[\pi_1]...[\pi_N]$. This fills each numbered hole $[\cdot]_i$ in C_N with the corresponding pattern π_i. For example, applying the context $C_2 = <d_1d_2[\cdot]_2 \mid <d_4[\cdot]_1d_6>$ to the patterns $\pi_1 = d_5$ and $\pi_2 = d_3>$ is written $C_2[d_5][d_3>]$ and results in the process $<d_1d_2d_3> \mid <d_4d_5d_6>$. Note that only patterns of the same *kind* can be replaced with each other: for example, a 3′ end pattern cannot be replaced with a nick pattern. The Methods provides more details on the well-formedness conditions for pattern substitution. To allow general logic predicates to be defined, our language embeds these patterns and contexts in a general logic programming language, by extending the standard syntax of Prolog (Table 3).

We now illustrate how this language can be used to define logic predicates that automatically generate the behavior of DNA strand displacement systems. The following logic predicate defines the conditions that need to be satisfied in order for processes `P1` and `P2` to bind, producing the resulting process `Q`:

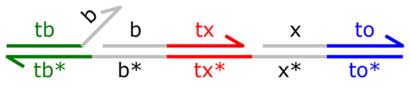
```
bind(P1,P2,Q,D!i) :-
  P1 = C1 [D], P2 = C2 [D′], compl(D, D′),
  Q  = C1 [D!i] | C2 [D′!i],
  freshBond(D!i, P1|P2).
```

The predicate is satisfied if `P1` matches a context `C1[D]` and `P2` matches a context `C2[D′]`, such that `D` is complementary to `D′`, as specified by the built-in predicate `compl(D,D′)`. The resulting process `Q` is obtained by replacing `D` with `D!i` in context `C1`, and replacing `D′` with `D′!i` in context `C2`, written `C1[D!i]|C2[D′!i]`. Furthermore, we require that the bond `i` is *fresh* in the sense that it should not occur anywhere in processes `P1` and `P2`. This is enforced by the built-in predicate `freshBond(D!i, P1|P2)`. This example illustrates how contexts are used to match specific patterns in a process and then update these matched patterns directly in place. The use of contexts in this way is a powerful abstraction for writing rules that generate behavior.[43] If we apply this rule to our example process defined previously, we obtain the following instantiations of the logical variables:

```
P1 = <tb^ b>,
C1 = <[.] b>, D = tb^
P2 = (<to^*!1 x*!2 tx^*!3 b*!4 tb^*>
     | <b!4 tx^!3> | <x!2 to^!1> ),
C2 = (<to^*!1 x*!2 tx^*!3 b*!4 [.] >
     | <b!4 tx^!3> | <x!2 to^!1> ), D′ = tb^*
Q  = C1[tb^!5] | C2[tb^*!5]
```

The resulting process `Q` is the complex that is produced when `P1` binds to `P2` and is represented graphically as follows:

```
[ <tb^!5 b>
| <b!4 tx^!3>
| <x!2 to^!1>
| <to^*!1 x*!2 tx^*!3 b*!4 tb^*!5>]
```

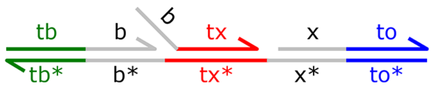


Similarly, the following predicate defines the conditions that need to be satisfied in order for process `P` to perform a strand displacement step, resulting in the process `Q`:

```
displace(P,Q,E!j,D!i) :-
  P = C [E!j D] [D!i] [D′!i E′!j],
  Q = C [E!j D!i] [D] [D′!i E′!j].
```

The predicate states that if the initial process `P` matches a context in which the sequence `D′ !iE′ !j` is bound to the sequence `E!jD` on bond `j` and to the sequence `D!i` on bond `i`, then the unbound domain `D` can replace the bound domain `D!i`. The sites `E!j` and `D!i` are included as arguments of the predicate, to record the bound domains at the beginning and end of the displacement, respectively. Applying this predicate to the above process results in the following, where the bound domain `b` has been displaced:

```
[ <tb^!5 b!4>
| <b tx^!3>
| <x!2 to^!1>
| <to^*!1 x*!2 tx^*!3 b*!4 tb^*!5>]
```



Note that this predicate only allows displacement to take place in the 5′ to 3′ direction. As a result, we also need to define a symmetric *displaceL* predicate for the 3′ to 5′ direction (Figure 1).

Finally, the following predicate defines the conditions for unbinding:

```
unbind(P,Q,D!i) :-
  P = C [D!i] [D′!i], toehold(D),
  Q = C [D]   [D′], not adjacent(D!i,_,P).
```

This encodes the assumption that only the toehold domains are short enough to unbind, as specified by the built-in predicate `toehold(D)`, and relies on an additional predicate to check that there are no bound domains adjacent to domain `D`:

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

Note that this predicate takes the site `D!i` as an argument, which contains both the domain `D` and its corresponding bond `i`. Since the bond can only occur twice in a well-formed process, this allows the inference system to pinpoint the specific domain on which unbinding occurs in order to test for adjacent bound domains, even if there are multiple occurrences of domain `D` in the system.

We now combine the various predicates to automatically generate the behavior of a nucleic acid system. In our language we choose to represent this behavior as a *chemical reaction network* (CRN), defined as a set of *reactions*, where each reaction consists of a multiset of reactant species, a reaction rate and a multiset of product species. To achieve this we follow the approach presented in ref 44, which defines a method for converting a process in an arbitrary biological programming language to a CRN. The method requires the definition of a *species* predicate, which converts a process in the language to a multiset of species, together with the definition of a *reaction* predicate, which generates the reactions that can occur involving one or more species. We adapted this method to our logic programming language, by defining a built-in *species* predicate that converts a process P to a multiset of species, where a species is a set of one or more strands that form a connected component, and by allowing the programmer to define one or more custom *reaction* predicates of the form *reaction*($[P_1;\cdots;P_N]$, R, Q). The *reaction* predicate takes as input a list of one or more processes $[P_1;\cdots;P_N]$ denoting the reactant species, together with a reaction rate R, and produces as output the product of the reaction, specified as a process Q. The system then takes care of splitting the process Q into individual products species, using the built-in *species* predicate. Applying this approach to our example, we can encode the rules of DNA strand displacement by defining *reaction* predicates for binding, displacement, and unbinding of strands, using our previously defined predicates.

```
reaction([P1;P2], "bind",Q) :- bind(P1,P2,Q,_).
reaction([P],"displace",Q) :- displace(P,Q,_,_).
reaction([P],"unbind",Q) :- unbind(P,Q,_).
```

In this case we define fixed rates for bind, displace, and unbind, however the predicates can be further refined so that the rates depend on the specific domains involved (Section S1.1). If we apply these three predicates to the Join example, we automatically generate a chemical reaction network of the system behavior (Figure 1), where the strand `<x to^>` is produced only if both strands `<tb^ b>` and `<tx^ x>` are present.

Although the rules in Figure 1 are sufficient to accurately generate the desired behavior of our Join example, additional predicates are needed in order for the same set of rules to generate the complete behavior of a broad range of DNA strand displacement systems. For instance, we also need to account for the case in which adjacent complementary domains can bind to each other, by defining an appropriate *cover* predicate (section S1.1). In addition, since the division of a DNA sequence into domains can be done arbitrarily, in order to maintain biological accuracy we need to ensure that the *displace*, *cover*, and *bind* predicates are extended to occur on a maximal sequence of consecutive domains. We achieve this by defining corresponding *displaces*, *covers*, and *binds* predicates, using a recursive encoding. For example, the following *displaces* predicate defines a maximal sequence of consecutive displacements:

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

There are two cases for the predicate: the *base case* and the *recursive case*. The base case `displaces(P,Q,E!j,[D])` holds if the single displacement predicate `displace(P,Q,E!j,D!i)` also holds, where `E!j` and `D!i` denote the sites at the beginning and end of the displacement, respectively, and no further displacement is possible starting from site `D!i`. The list `[D]` contains the consecutive domains on which the displacement takes place, where the list contains only a single domain `D` in the base case. The recursive case `displaces(P,R,E!j,[D#L])` holds if process `P` can perform a single displacement to become process `Q`, beginning at site `E!j` and ending at site `D!i`, and furthermore if process `Q` can itself perform multiple displacements starting at site `D!i` along the list of domains `L` to produce process `R`. The list `[D#L]` adds the domain `D` to the list of domains `L`. We define similar recursive predicates for the symmetric case of the displace rule and for the cover and bind rules (section S1.1). In the case of unbinding, since only toehold domains are assumed to be short enough to unbind, we do not consider unbinding along a maximal sequence of consecutive domains. However, in some cases multiple consecutive toeholds may still be short enough to unbind. To support this functionality, we allow the user to specify unbinding rates for specific sequences of consecutive domains (section S1.1).

Abstraction Hierarchy. Our logic programming language also allows an *abstraction hierarchy* of behaviors to be defined, by encoding assumptions about which reactions can be considered sufficiently fast to be merged into a single step. We illustrate this by encoding one of the semantic abstractions from ref 27. To define a semantic abstraction, we label reactions as either *fast* or *slow* and define the following predicate to merge fast reactions:

```
merge(P,P,V) :- not fast([P],_,_).
merge(P,R,V) :-
  fast([P],_,Q), not member(Q,V),
  merge(Q,R,[Q#V]).
```

This predicate assumes that only unimolecular reactions involving a single species can be considered fast, since bimolecular reactions are limited by the relatively slow rate of molecular diffusion. The list `V` stores the list of processes corresponding to each step of the merge. This is used to ensure that we do not revisit the same process twice, in order to avoid getting trapped in an infinite cycle of fast reactions. We then update the *reaction* predicate to merge each slow reaction with a maximal sequence of consecutive fast reactions as follows:

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

In this way, changing which reactions are fast or slow changes the semantic abstraction. If all reactions are labeled as slow then this corresponds to the *Detailed* semantics of ref 27. For example, the generated CRN in Figure 1 corresponds to the *Detailed* semantics applied to the Join example. If all unimolecular reactions are fast then this corresponds to the *Infinite* semantics

A

```
directive rules {
bind(P1,P2,Q,D!i) :-
  P1 = C1 [D], P2 = C2 [D'], compl(D, D'),
  Q = C1 [D!i] | C2 [D'!i], freshBond(D!i, P1|P2).

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

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

unbind(P,Q,D!i) :-
  P = C [D!i] [D'!i], toehold(D),
  Q = C [D] [D'], not adjacent(D!i,_,P).

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

reaction([P1;P2], "bind",Q) :- bind(P1,P2,Q,_).
reaction([P],"displace",Q) :- displace(P,Q,_,_).
reaction([P],"displace",Q) :- displaceL(P,Q,_,_).
reaction([P],"unbind",Q) :- unbind(P,Q,_).
}
directive parameters [
  bind = 0.003; displace = 1; unbind = 0.1
]
( 10 [<tb^ b>]
| 10 [<tx^ x>]
| 100 [<to^*!1 x*!2 tx^*!3 b*!4 tb^*>
      | <x!2 to^!1> | <b!4 tx^!3>] )
```

B

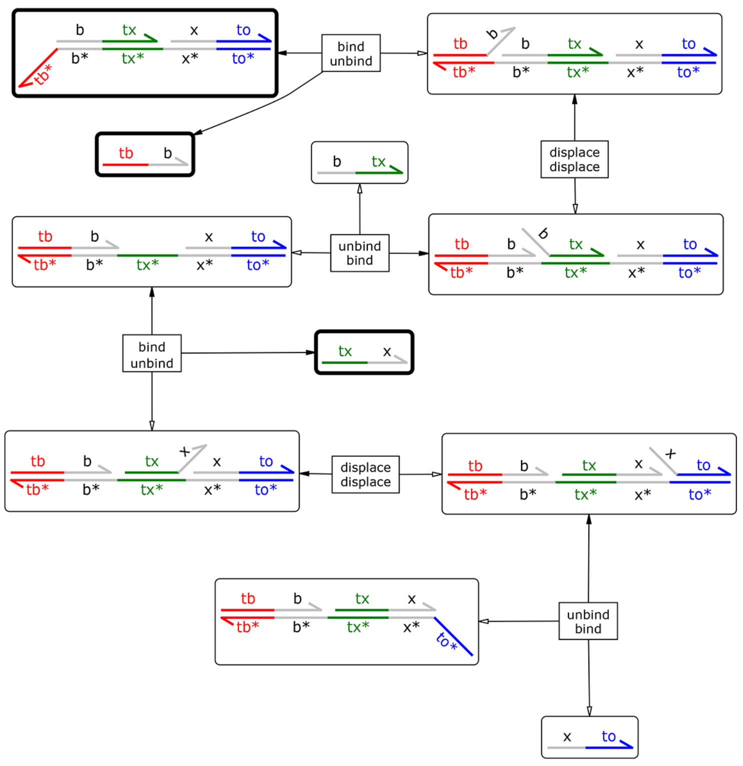


Figure 1. Logic program and automatically generated chemical reaction network for a DNA strand displacement example computing the Join of two signals. (A) Logic program encoding binding, unbinding, and displacement predicates, together with the initial conditions of the DNA strand displacement system. (B) Graphical representation of the corresponding chemical reaction network generated by the logic program. The graph consists of two types of nodes, representing *species* and *reactions*. Each species node contains a graphical representation of a DNA complex. Each reaction node displays the rate of the forward reaction on top and the rate of the reverse reaction, when present, on the bottom. Edges between a species node and a reaction node that have an open arrowhead denote the products of the reaction. Edges with either no arrowhead or a solid arrowhead denote the reactants, where solid arrowheads are used to denote a reversible reaction. Species present initially are highlighted in bold, with the remaining species generated automatically by the logic program.

of ref 27. For example, the generated CRN in Figure 2 corresponds to the Infinite semantics applied to the Join example. Intermediate semantic abstractions can be defined in a similar fashion. The full set of DNA strand displacement rules is provided in section S1.1, including rules for semantic abstractions. These rules are general enough to be applied to a broad range of systems, including most of the systems currently supported by the Visual DSD language defined in ref 27. Previously, the rules of this language were hard-coded in a corresponding implementation. Instead, our logic programming approach allows these rules to be defined by the user as high-level predicates in a logic program, and easily modified and extended. Furthermore, the principle of semantic abstractions is more general than strand displacement, and can be similarly applied to a broad range of alternative nucleic acid implementation strategies using our language.

Kinetic Rate Hypotheses. Our logic programming language can also encode hypotheses that determine how kinetic rates are assigned to reactions. Previous research[15] used Bayesian inference and model selection to compare different kinetic rate hypotheses, in order to determine which hypothesis was most likely, given the experimental data. Four new hypotheses were compared with a *Default* hypothesis, in which the binding rate of a toehold was determined purely by its DNA sequence.[27] However, for the new hypotheses the kinetic rates of the individual chemical reactions needed to be manually encoded. Here, we demonstrate how our logic programming language can encode kinetic rate hypotheses as high-level logic predicates, to automatically generate chemical reactions with consistent rates. We illustrate this using the kinetic rate hypothesis selected in ref 15 to be most likely given the data, in which parameters were assigned based on the toehold sequence and its surrounding context, referred to as the *Unique Context* hypothesis. In particular, the hypothesis made a distinction between *internal* toeholds, defined as those with a bound domain on either side, and *external* toeholds, defined as those with a bound domain on one side only.

For simplicity, the code in Figure 1 assumed that all bind, displace, and unbind reactions have the same rate. We first extended this to encode the *Default* kinetic rate hypothesis,[27] which allows a separate rate to be assigned to each toehold and provides support for default rates:

```
reaction([P1;P2], Rate, Q) :-
  find(D, "bind", Rate), bind(P1,P2,Q,D!i).
find(D,Type,Rate):- rate(D,Type,Rate).
find(D,Type,Rate):-
  default(D,Type,Rate), not rate(D,Type,_).
default(_,"bind",0.0003).
rate(tb^,"bind",0.0001).
rate(tx^,"bind",0.0002).
rate(to^,"bind",0.0003).
```

A

```
directive rules {
bind(P1,P2,Q,D!i) :-
  P1 = C1 [D], P2 = C2 [D'], compl(D, D'),
  Q = C1 [D!i] | C2 [D'!i], freshBond(D!i, P1|P2).

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

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

unbind(P,Q,D!i) :-
  P = C [D!i] [D'!i], toehold(D),
  Q = C [D] [D'], not adjacent(D!i,_,P).

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

slow([P1;P2], "bind",Q) :- bind(P1,P2,Q,_).
fast([P],"displace",Q) :- displace(P,Q,_,_).
fast([P],"displace",Q) :- displaceL(P,Q,_,_).
fast([P],"unbind",Q) :- unbind(P,Q,_).

merge(P,P,V) :- not fast([P],_,_).
merge(P,R,V) :-
  fast([P],_,Q), not member(Q,V), merge(Q,R,[Q#V]).

reaction([P1;P2],Rate,R) :- slow([P1;P2],Rate,Q), merge(Q,R,[(P1|P2);Q]).
reaction([P],Rate,R) :- slow([P],Rate,Q), merge(Q,R,[P;Q]).
}
directive parameters [ bind = 0.003;  displace = 1; unbind = 0.1 ]
( 10 [<tb^ b>]
| 10 [<tx^ x>]
| 100 [<to^*!1 x*!2 tx^*!3 b*!4 tb^*>
      | <x!2 to^!1> | <b!4 tx^!3>] )
```

B

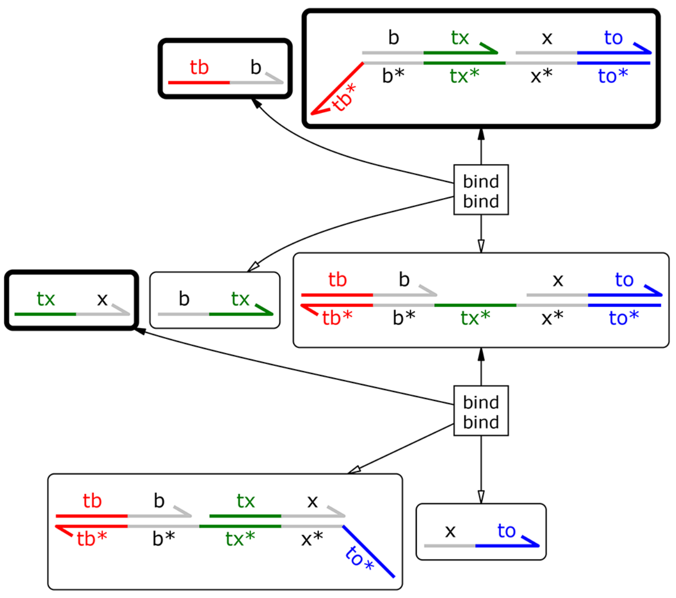


Figure 2. Logic program and automatically generated chemical reaction network for the Join example, with additional predicates for merging fast reactions. (A) The logic program of Figure 1 is extended with predicates for distinguishing between reactions that are considered either fast or slow, and for merging a maximal sequence of fast reactions. (B) Graphical representation of the corresponding chemical reaction network generated by the logic program. In this network, binding, displacement, and unbinding reactions are merged into a single step.

The *reaction* predicate looks up the rate of the reaction using the *find* predicate, which takes the domain D and the type of the reaction, in this case "bind", and returns the corresponding Rate. The programmer can specify a default bind rate for all domains using the *default* predicate, and specific bind rates associated with individual domains using the *rate* predicate. A similar approach can also be used to associate unbinding and displacement rates to specific domains.

To encode a more fine-grained kinetic hypothesis, in which the binding rate depends not only on the domain but also on its context, we can replace the existing *bind* predicate with context-specific predicates that check whether a domain is on the 3′ end, the 5′ end, or flanked on either side, and assign different rates accordingly (Figure 3). This simple example illustrates the overall approach; however, more complex encodings will be required in the general case. The key point is that the same predicates can be applied to a broad range of systems, in order to automatically assign kinetic parameters in a manner consistent with the encoded hypothesis. These parameters can then be fitted to data, for example using Bayesian parameter inference[15]. In addition to assigning parameter variables in a consistent manner, the same approach can be used to compute the numerical rate values themselves, either using simple heuristics such as the fraction of G-C nucleotide pairs, or by encoding predicates for computing free energies directly, since our language is sufficiently expressive to encode arbitrary computation.

Enzyme Interactions. Here we demonstrate the potential of our logic programming language to encode the behavior of a broad range of nucleic acid enzymes, by encoding the enzymes of the Polymerase-Exonuclease-Nickase (PEN) DNA toolbox[20−23,35,45] as logic predicates.

The DNA polymerase enzyme of the toolbox extends the 3′ end of a strand bound to a template, where the *polymerase* predicate is defined as follows:

```
polymerase(P,Q,A!i,B!j) :-
  P = C [A!i>]     [B'   A'!i], compl(B, B'),
  Q = C [A!i B!j>] [B'!j A'!i], freshBond(B!j, P).
```

The predicate states that if site A!i occurs at the 3′ end of a strand and is bound to a complementary sequence B′A′!i, then the 3′ end is extended by the polymerase enzyme, which creates a complementary site B!j bound to the template on a fresh bond j. We also allow the polymerase to displace domains already bound to the template, to encode the behavior of the strand displacing polymerase used in the toolbox:

```
polymerase(P,Q,A!i,B!j) :-
  P = C [A!i>]     [B'!j A'!i] [B!j],
  Q = C [A!i B!j>] [B'!j A'!i] [B].
```

A

```
directive rules {
... //Insert strand displacement predicates apart from bind
bindR(P1,P2,Q,D!i) :-
  P1 = C1 [D], P2 = C2 [<D' E'!j] [E!j], compl(D, D'),
  Q = C1 [D!i] | C2 [<D'!i E'!j] [E!j], freshBond(D!i, P1|P2).

bindL(P1,P2,Q,D!i) :-
  P1 = C1 [E'!j D'>] [E!j], P2 = C2 [D], compl(D, D'),
  Q = C1 [E'!j D'!i>] [E!j] | C2 [D!i], freshBond(D!i, P1|P2).

bindM(P1,P2,Q,D!i) :-
  P1 = C1 [F'!k D' E'!j] [E!j] [F!k], P2 = C2 [D], compl(D,D'),
  Q = C1 [F'!k D'!i E'!j] [E!j] [F!k] | C2 [D!i],
  freshBond(D!i, P1|P2).

slow([P1;P2],Rate,Q) :- find(D,"bindR",Rate),bindR(P1,P2,Q,_).
slow([P1;P2],Rate,Q) :- find(D,"bindL",Rate),bindL(P1,P2,Q,_).
slow([P1;P2],Rate,Q) :- find(D,"bindM",Rate),bindM(P1,P2,Q,_).

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

default(_,"bindR", 0.003). rate(to^,"bindR","ktoR").
default(_,"bindL", 0.003). rate(tb^,"bindL","ktbL").
default(_,"bindM", 0.006). rate(tx^,"bindM","ktxM").
}
directive parameters [ bind = 0.003;  displace = 1; unbind = 0.1 ]
( 10 [<tb^ b>]
| 10 [<tx^ x>]
| 100 [<to^*!1 x*!2 tx^*!3 b*!4 tb^*>
      | <x!2 to^!1> | <b!4 tx^!3>] )
```

B

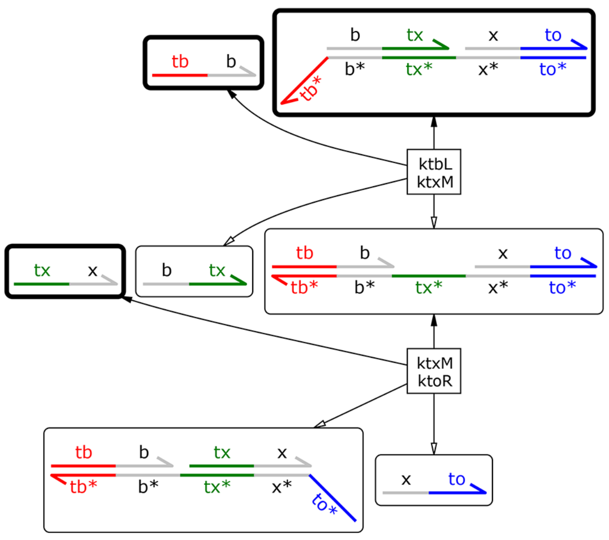


Figure 3. Logic program and automatically generated chemical reaction network for the Join example, with additional predicates for encoding kinetic rate hypotheses. (A) The logical program code is assumed to contain all of the predicates from Figure 2 apart from the *bind* predicate, which is replaced by three context-specific binding predicates that check whether a domain is on the 3′ end (*bindR*), the 5′ end (*bindL*) or flanked on either side (*bindM*), and assigns different rates accordingly. (B) Graphical representation of the corresponding chemical reaction network. A different kinetic parameter is used for a given domain in a given context.

Similar to the *displaces* predicate, we define a recursive *polymerases* predicate that allows a maximal sequence of domains to be extended by the polymerase in a single step, until the end of the template is reached:

```
polymerases(P,R,A!i,[B#L]) :-
 polymerase(P,Q,A!i,B!j),
 polymerases(Q,R,B!j,L).
polymerases(P,Q,A!i,[B]) :-
 polymerase(P,Q,A!i,B!j),
 not polymerase(Q,_,_,_).
```

The exonuclease enzyme of the toolbox degrades an unprotected single strand of DNA from the 5′ end, where the base case of the *exonuclease* predicate is defined as follows:

```
exonuclease(P,Q,A,[A]) :-
  P = C[<A>], unbound(A),
  Q = C[nil], not protected(A).
protected( _ {"O"}).
```

This states that if the process P contains a strand < A> consisting of a single unbound and unprotected domain A, then the strand is removed. Removal of a strand is encoded by replacing the strand with nil, which represents the empty process ⌀. Protection of a strand is encoded by the *protected* predicate, written protected(A). The predicate makes use of *tags t* (Table 1), which can be optionally associated with a domain *d*, written *d*{*t*}, where a tag can be an integer, a string, a name or a recursive structure $x(T_1,...,T_N)$. Here we assume that a given domain A is protected from exonuclease degradation if it has a phosphorothioate bond at its 5′ end, represented by tagging the domain with "O", written A{"O"}. The recursive case of the *exonuclease* predicate is defined as follows:

```
exonuclease(P,R,A@p,[A#L]) :-
  P = C[<A@p B], unbound(A),
  Q = C[<B@q], not protected(A),
  exonuclease(Q,R,B@q,L).
```

This states that if the process P contains the pattern <A@p B such that the domain A at the 5′ end is unbound and unprotected, then the domain is removed and the predicate is called recursively on the remaining 5′ end B@q of the strand. The predicate makes use of *locations l* (Table 1) that can be optionally associated with a domain *d*, written *d*@*l*, where a location can be a name, an integer or a logical variable. Locations are used to distinguish between unbound domains with the same name. In the case of bound domains we can simply use the bond as the identifier, since no two domains with the same name can have the same bond; however, for unbound domains an additional location identifier is needed. The location is passed in the recursive call to the *exonuclease* predicate, to allow a maximal sequence of domains to be degraded in a single step.

Finally, the nickase enzyme of the toolbox introduces a nick after a specific recognition site in a strand, where the *nickase* predicate is defined as follows:

```
nickase(P,Q,[A]) :-
  recognition([A]),
  P = C [A!i B!j]     [B'!j A'!i],
  Q = C [A!i> | <B!j] [B'!j A'!i].
```

A

```
directive rules {
... // Insert strand displacement predicates
polymerase(P,Q,A!i,B!j) :-
  P = C [A!i>] [B’ A’!i], compl(B, B’),
  Q = C [A!i B!j>] [B’!j A’!i], freshBond(B!j, P).

polymerase(P,Q,A!i,B!j) :-
  P = C [A!i>] [B’!j A’!i] [B!j],
  Q = C [A!i B!j>] [B’!j A’!i] [B].

polymerases(P,R,A!i,[B#L]) :-
  polymerase(P,Q,A!i,B!j), polymerases(Q,R,B!j,L).
polymerases(P,Q,A!i,[B]) :-
  polymerase(P,Q,A!i,B!j), not polymerase(Q,_,_,_).

exonuclease(P,R,A@p,[A#L]) :-
  P = C[<A@p B], unbound(A),
  Q = C[<B@q], not protected(A),
  exonuclease(Q,R,B@q,L).

exonuclease(P,Q,A@p,[A]) :-
  P = C[<A@p>], unbound(A),
  Q = C[nil], not protected(A).
protected( _ {"O"}).

nickase(P,Q,[A]) :- recognition([A]),
  P = C [A!i B!j]     [B’!j A’!i],
  Q = C [A!i> | <B!j] [B’!j A’!i].
recognition([a^]).

reaction([P], "polymerase", Q) :- polymerase(P,Q,_,_).
reaction([P], "exonuclease", Q) :- exonuclease(P,Q,_,_).
reaction([P], "nickase", Q) :- nickase(P,Q,_).
}

directive parameters [
  bind = 4.3e-4;  displace = 1;
  unbind = 0.01; polymerase = 0.3;
  nickase = 0.05; exonuclease = 0.005;
]
( 10  [<b^*{"O"} a^*>]
| 1  [<a^>] )
```

B

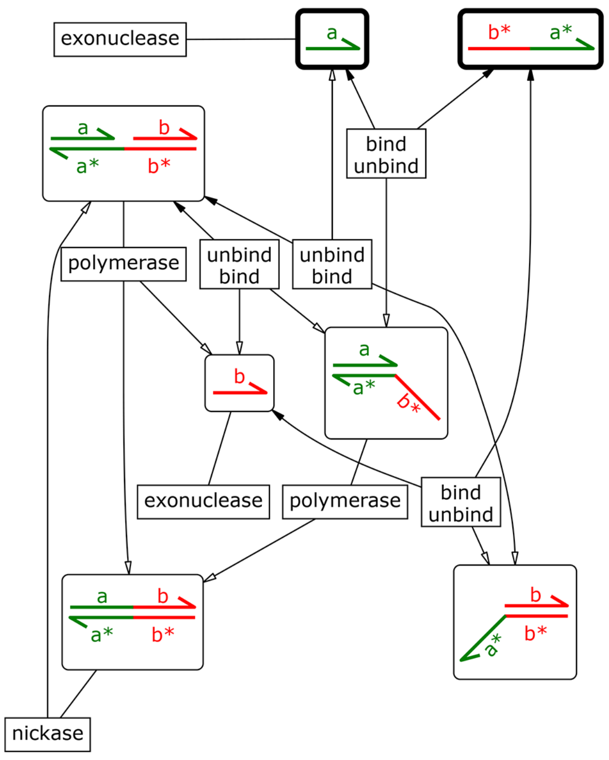


Figure 4. Logic program and corresponding chemical reaction network for enzyme computation. (A) The logic program contains predicates for strand displacing polymerase, exonuclease, and nickase. It is also assumed to contain all of the logic predicates of Figure 1. The initial conditions consist of a template strand with a phosphorothioate modification at the 5′ end to protect from degradation, together with a single activator strand. (B) Chemical reaction network generated from the logic program. The network demonstrates the main functionality of the three types of enzymes.

This states that if the process P contains a double stranded region with consecutive bound sites A!i and B!j, then a nick is introduced between the two sites, represented by the nicking pattern A!i>|<B!j. The nick is only performed if domain A is a recognition site for a nicking enzyme, represented by the predicate recognition ([A]). In the general case a recognition site can be a list of one or more consecutive domains.

Figure 4 shows how the predicates for these enzymes can automatically generate the behavior of a simple system, consisting of a protected template and an activator strand. The complete definitions of polymerase and nickase enzymes are summarized in section S1.2. In addition, section S1.2.1 presents example predicates encoding the behavior of ligation and restriction enzymes. These enzymes are not contained in the PEN DNA toolbox, but are used in many molecular biology processes such as Gibson Assembly.[46]

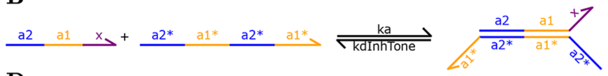
We used our enzyme predicates to model a synthetic biochemical oscillator called the *Oligator*,[20] which was previously implemented experimentally using the PEN DNA toolbox. We modeled the system at two levels of abstraction: the domain level and the nucleotide level. The system comprises three main types of interaction: activation, inhibition, and degradation. Activation follows from the amplification of a signal α in a positive feedback loop. The signal is produced by a polymerase-nickase reaction on a template strand T_1, which is composed of two adjacent complementary copies of α. This is similar to the template in Figure 4 but with sequence <a^*{"O"}a^*>, where α is represented by the domain a^. When α binds to T_1, polymerase extends α to $\alpha\alpha$, then nickase cuts the two signals, similar to the *bind*, *polymerase*, and *nickase* reactions in Figure 4. The operating temperature is chosen such that $\alpha\alpha$ binds stably to the template, but α unbinds spontaneously. Inhibition occurs by the chained production of two strands β and *Inh*, produced by templates T_2 and T_3, respectively, where *Inh* then inhibits T_1. This introduces the required delay to produce oscillations. Template T_2 produces β after binding a signal α, similar to the template <b^*{"O"}a^*> in Figure 4, while template T_3 produces *Inh* after binding β. The strand *Inh* binds more strongly to T_1 than α and is designed with a mismatching sequence at its 3′ end that

A

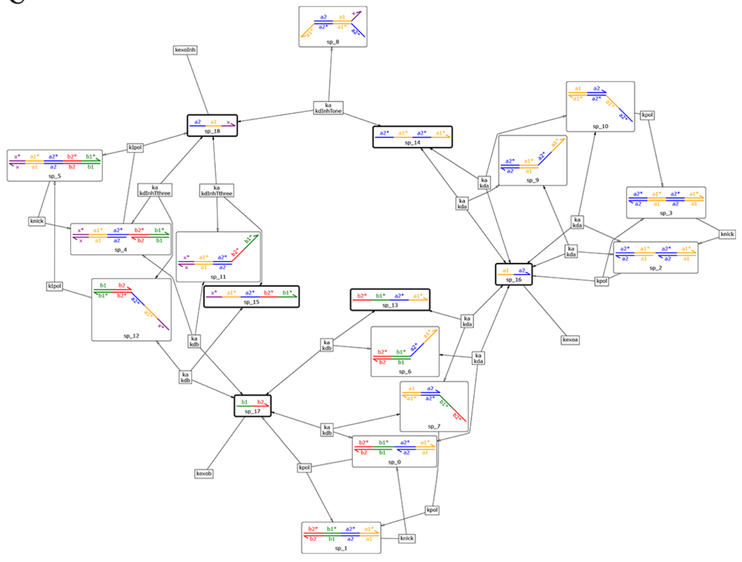
```
directive rules {
... // Insert strand displacement predicates
... // Insert enzyme predicates
nickase(P,Q,[A1;A2]) :- recognition([A1;A2]),
  P = C [A1!i1 A2!i2 B!j] [B'!j A2'!i2 A1'!i1],
  Q = C [A1!i1 A2!i2> | <B!j] [B'!j A2'!i2 A1'!i1].

//Require at least 2 consecutive nucleotides to bind
default([_;_#_],"bind","ka").
recognition([a1^; a2^]).
recognition([b1^; b2^]).
}
( 30 [<a2^*{"O"} a1^* a2^* a1^*>]      // T1
| 5  [<b2^*{"O"} b1^* a2^* a1^*>]      // T2
| 30 [<x^*{"O"} a1^* a2^* b2^* b1^*>] // T3
| 1  [<a1^ a2^>]                       // alpha
| 1  [<b1^ b2^>]                       // beta
| 1  [<a2^ a1^ x^>] )                  // Inh
```

B



C



D

```
directive rules {
... // Insert strand displacement and enzyme predicates
nickase(P,Q,[A1;A2;A3;A4;A5]) :- recognition([A1;A2;A3;A4;A5]),
  P = C [A1!i1 A2!i2 A3!i3 A4!i4 A5!i5 N1 N2 N3 N4 B!j] [B'!j N4' N3' N2' N1' A5'!i5 A4'!i4 A3'!i3 A2'!i2 A1'!i1],
  Q = C [A1!i1 A2!i2 A3!i3 A4!i4 A5!i5 N1 N2 N3 N4> | <B!j] [B'!j N4' N3' N2' N1' A5'!i5 A4'!i4 A3'!i3 A2'!i2 A1'!i1].

default([_; _; _; _; _; _; _; _; _; _ ; _ # L ],"bind",0.0003). //Require at least 11 consecutive nucleotides to bind
recognition([c^*; a^; c^*; a^*; c^]).
}
( 30 [<a^{ "O" }  a^ c^ a^ c^* a^ c^ a^* c^ c^* a^      a^ a^ c^ a^ c^* a^ c^ a^* c^ c^* a^>] // T1
| 5 [<c^*{ "O" } c^ a^ a^* c^* a^ c^ a^* c^ a^ a^*      a^ a^ c^ a^ c^* a^ c^ a^* c^ c^* a^>] // T2
| 30 [<a^*{ "O" } a^* a^ c^ a^* c^ c^* a^ a^ a^ c^ a^ c^* a^ c^ a^*     c^* c^ a^ a^* c^* a^ c^ a^* c^ a^ a^*>] //T3
| 1 [<a^* c^ c^* a^ c^* a^* c^ a^* c^* a^* a^* >] // alpha
| 1 [<a^ a^* c^* a^ c^* a^* c^ a^ a^* c^* c^ >]  // beta
| 1 [<a^ c^* a^* c^ a^* c^* a^* a^* a^* c^ c^* a^ c^* a^* a^ a^>] ) // Inh
```

E

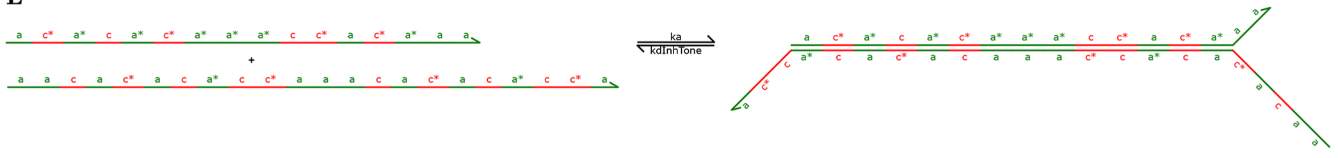


Figure 5. Logic programs for the synthetic Oligator,[20] encoded at the domain and nucleotide levels. (A) Domain-level logic program, where each signal is encoded as two consecutive domains. This is needed in order to implement the inhibition mechanism. For example, signal `alpha` is encoded as `a1^ a2^`. Separate recognition predicates are defined for `alpha` and `beta`. The full program code is provided in section S2.3. (B) Domain-level inhibition reaction, in which the inhibitor strand `Inh` binds to the `T1` template in order to inhibit amplification of `alpha`. (C) Automatically generated chemical reaction network, which is analogous to the manually encoded CRN from ref 20. (D) Nucleotide-level logic program, where the nucleotide `t` is encoded as `a*` and the nucleotide `g` is encoded as `c*`. Only a single recognition predicate is defined at the nucleotide level, which is applicable to both `alpha` and `beta`. The full program code is provided in section S2.4. Importantly, the code generates the same CRN as the domain-level logic program. (E) Nucleotide-level inhibition reaction.

prevents elongation by polymerase, thus inhibiting the production of more α. Finally, degradation is the continuous destruction of all species, except for the templates, by exonuclease. Phosphorothioate bonds at the 5′ end of all templates prevent exonuclease from degrading them. Note that only T_1, T_2, T_3, and α are necessary to start the oscillatory system.

Our domain-level encoding of the Oligator is shown in Figure 5A. Note that instead of modeling a signal α as a single domain `a^`, we model it as a sequence of two toeholds `<a1^ a2^>`. Similarly, we model signal β as `<b1^ b2^>` and signal *Inh* as `< a2^ a1^ x>`, where `x` is used as the mismatching sequence. This more fine-grained representation of signals is needed to encode the inhibition mechanism (Figure 5B). Templates are modeled as a sequence complementary to the input and output signals, and phosphorothioate bonds are modeled by the tag `"O"` at the 5′ end of each template. The recognition sites for nickase are α and β, as defined in the two `recognition` predicates. Note that in principle the signal α could bind `a2^*` on one T_1 template, and `a1^*` on a second T_1 template, recursively, resulting in the formation of template and signal polymers. However, in practice such polymers are unstable at the operating temperature of the experiments, because of the short length of the toehold. To avoid modeling such spurious polymers, binding rates are only allowed for sequences involving at least two domains, by writing `default([_;_#_],"bind","ka")`.

Our framework can also express the Oligator directly at the nucleotide level, using almost exactly the same set of logical predicates, where binding takes place on a maximal sequence of nucleotides. The DNA sequences of the Oligator signals and templates are the following (taken from Supporting Information of ref 20):

T_1	A*A*CA**GACTC**_GA-AACA_**GACTC**GA-3’P
T_2	G*C*AT**GACTC**AT-AACA**GACTC**GA-3’P
T_3	T*T*ACTCGAAACAGACT-GCAT**GACTC**AT-3’P
α	TC**GAGTC**TGTT
β	AT**GAGTC**ATGC
Inh	_AGTCTGTTTCGAGT_AA

All sequences are defined from $5'$ to $3'$, where the asterisk $(*)$ indicates a phosphorothioate modification, $3'$P indicates a $3'$ terminal phosphate modification used to block elongation, and bold letters correspond to the recognition sequence of the nicking enzyme Nt.BstNBI. Italicized letters on T_1 and *Inh* indicate their matching subsequences. The full logic program of the nucleotide Oligator is provided in section S2.4, and the initial conditions are shown in Figure 5D. We model nucleotides *A* and *C* with the toeholds `a^` and `c^`, respectively, where complementary nucleotides *T* and *G* are modeled as `a*` and `c*`, respectively. The structure of the encoding is similar to the one in Figure 4, except that the nickase recognition site has 5 bases instead of 2 domains. The `default` predicate ensures that binding can only occur for sequences at least 11 bases long, which is the length of the α and β species. Without this assumption, partial bindings of templates and signals can form polymers. Both the domain-level and nucleotide-level logic programs produce the same chemical reaction network automatically (Figure 5C), consistent with the manually encoded CRN of ref 20. One benefit of the nucleotide level model is that it allows a more precise encoding of the nicking site, where only a single recognition sequence is defined, which is present in both α and β strands. More generally, the nucleotide model can detect a broader class of errors than the domain model, such as the presence of nicking sites in unintended regions.

Localized Interactions. Recent work demonstrated how nucleic acid components that are localized to a surface can be used to perform computation.[17−19] Localized components can also perform nanoscale locomotion, where nucleic acid strands tethered to a surface form tracks that a nanomolecular device, such as a walker, can traverse.[4,18,47−51] Localization helps ensure that only spatially proximal strands can interact with each other. The interactions are determined by the geometry and biophysical constraints of the system, which can be modeled at varying levels of abstraction.[52] One approach is to rely on the programmer to specify which tethered components are close enough to interact.[30] In this approach, tethered strands are labeled with *tether*(a_1,...,a_N), where each a_i is a constant called a *location tag*, and strands that share at least one location tag are considered close enough to interact. We encode this approach in our logic programming language by associating location tags to a tethered domain. The predicate `tethered(d, [A1;...;AN])` associates tags `A1`,⋯,`AN` to the tethered domain `d`, where each tag is represented as a string. Using this approach, we only need to change the `bind` predicate in the existing strand displacement semantics to encode localized interactions:

```
bind(P1,P2,Q,D!i) :-
  proximal(P1, P2),
  P1 = C1 [D], P2 = C2 [D'], compl(D, D'),
  Q  = C1 [D!i]  |  C2 [D'!i],
  freshBond(D!i, P1|P2).

proximal(P1, P2) :-
  tethers(P1, Ts1), tethers(P2, Ts2),
  shared(Ts1, Ts2), pruning(P1, P2).
```

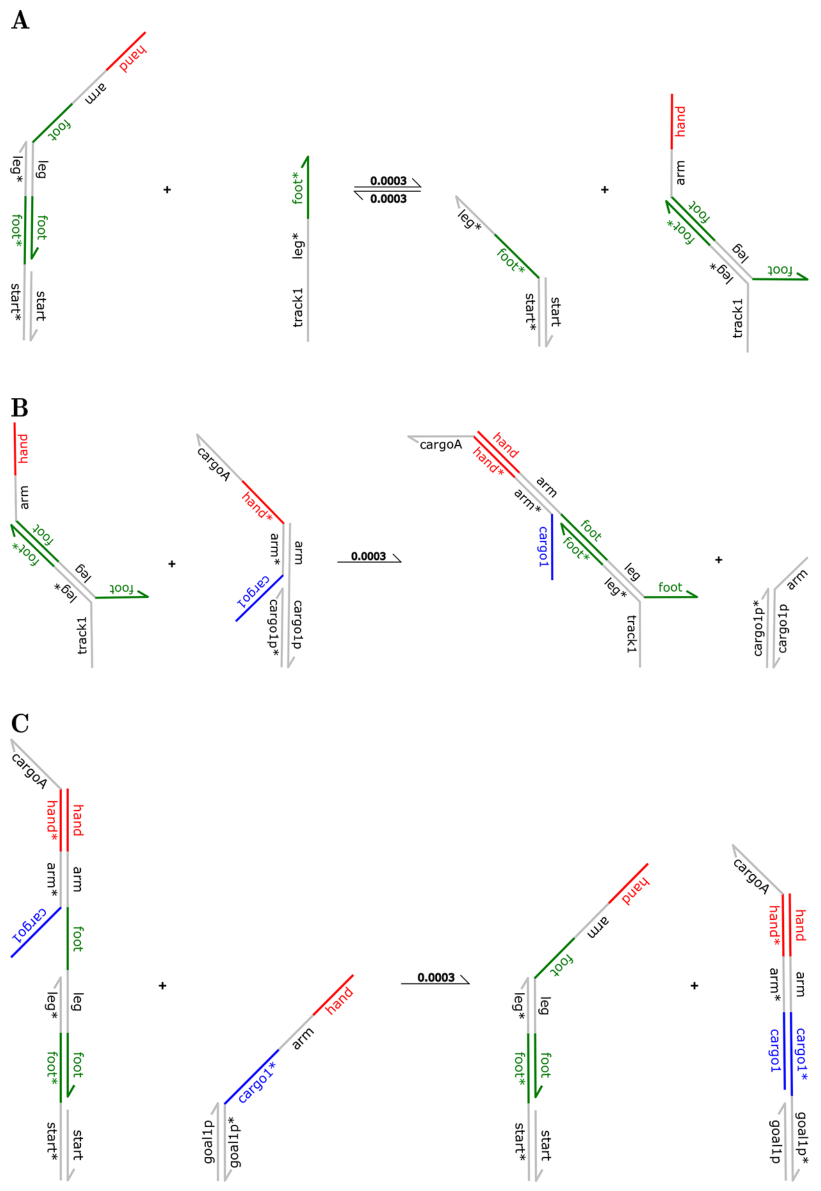
The `proximal` predicate checks whether species `P1` and `P2` are close enough to interact, where the `tethers` predicate identifies the tethers of the two species, and the `shared` predicate checks whether one of the species is freely diffusing with no tethers, or whether they are both tethered and share at least one location tag. The full definitions of the predicates are provided in section S1.3. In this encoding, the `pruning` predicate checks that species `P1` and `P2` have at most one tether each; however, more general encodings[52] can also be expressed.

To illustrate the approach, we present a logic program based on a previously implemented cargo-sorting DNA robot[18] (Figure 6). A DNA origami surface contains two tethered *cargo* strands, each with a cargo attached (`cargo1p` and `cargo2p`), where each cargo is tagged with a different fluorophore. There are two tethered *goal* strands, each complementary to one of the cargos. The goals are situated at opposite ends of a line of intermediate tethered *track* strands (`track1`, `track2`, and `track3`) that form a track. The cargos and the starting point of the robot are next to these track strands. The task of the DNA robot is to transport each cargo to its goal.

The robot takes random walks through the origami by alternatively binding two `foot` toeholds from one tethered strand to the next. It is also equipped with a `hand` toehold, whose complement is situated on the cargo. When this toehold binds to the cargo, a strand displacement reaction with the `arm` domain detaches the cargo. Once picked up, the DNA robot transports the cargo back to its corresponding tethered goal strand and releases it again by strand displacement.

Complex Nucleic Acid Topologies. Initial versions of the Visual DSD language focused on nucleic acid systems without any branching structures.[26,27] However, many interesting molecular devices require rich secondary structures such as hairpins, branches, loops, and pseudoknots in order to function. In subsequent work, a version of Visual DSD based on *strand graphs*[31] was proposed, in order to support such structures. The strand graph language provided a syntax for encoding arbitrary secondary structures at the domain level, together with a fixed set of behavioral rules that was hard-coded in the language implementation. Here we demonstrate that our logic programming language is sufficiently general to encode the same complex nucleic acid structures as strand graphs, together with the logic predicates needed to automatically generate their corresponding behavior. Importantly, our approach allows strand graph behaviors to be combined with enzyme interactions and other nucleic acid implementation strategies in a unified and extensible manner.

We first encoded the full semantics of strand graphs[31] in our logic programming language (section S1.4). This includes more general predicates that allow four-way branch migration and three-way branch migration across junctions. To demonstrate the combination of branching structures with additional



D

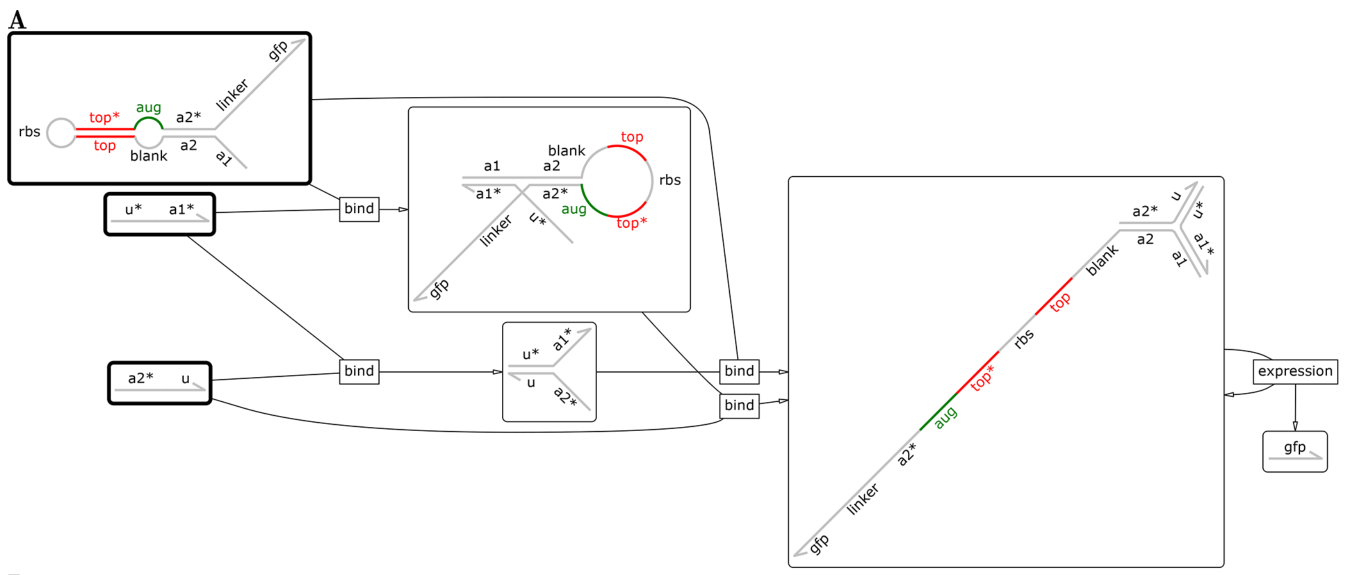
```
directive rules{
... //Insert strand displacement predicates
    //apart from bind
... //Insert localised bind rule

tethered(start,["dropA"; "toCargo"]).
tethered(track1,["toCargo"; "xing1"]).
tethered(track2,["xing1"; "xing2"; "pkup"]).
tethered(track3,["xing2"; "dropB"]).
tethered(goal1p,["dropA"]).
tethered(goal2p,["dropB"]).
tethered(cargo1p*,["pkup"; "xing1"]).
tethered(cargo2p*,["pkup"; "xing2"]).
}
( // robot
1 [ <hand^ arm foot^ leg!0 foot^!1>
  | <start*!3 foot^*!1 leg*!0>
  | <start!3> ]
// goals
| 1 [ <hand^ arm cargo1^* goal1p*!0>
    | <goal1p!0> ]
| 1 [ <hand^ arm cargo2^* goal2p*!0>
    | <goal2p!0> ]
// cargos
| 1 [ <cargo1^ arm*!0 hand^* cargoA>
    | <arm!0 cargo1p!1>
    | <cargo1p*!1> ]
| 1 [ <cargo2^ arm*!0 hand^* cargoB>
    | <arm!0 cargo2p!1>
    | <cargo2p*!1> ]
//tracks
| 1 [ <track1 leg* foot^*> ]
| 1 [ <track2 foot^* leg*> ]
| 1 [ <track3 leg* foot^*> ] )
```

Figure 6. Logic program of a cargo-sorting DNA robot,[18] and a subset of the corresponding chemical reactions. (A) The DNA robot performs a random walk over origami tracks by reversible strand displacement. (B) The robot picks up the cargo irreversibly. (C) The robot drops the cargo at its goal track irreversibly. (D) Logic program, which uses the strand displacement predicates of section S1.1, replacing the `bind` predicate with its localized counterpart.

implementation strategies, we focused on *ribocomputing devices*,[5] which involve multiple hairpins and branched junctions, together with additional translational machinery. A ribocomputing device is a collection of transcribed RNA strands that encode a logic circuit, where RNA molecules are provided as inputs and protein translation generates the corresponding output. The ribosome binding site (RBS) that triggers translation is enclosed in a hairpin structure, which inhibits translation by preventing access to the RBS, as long as the hairpin remains closed. When present, input RNA molecules activate a strand displacement reaction that opens the hairpin, leading to translation of the protein output, which is typically a reporter protein such as GFP or lacZ. Ribocomputing logic gates have been reported[5] that implement AND, OR, and NOT logic. In this scheme, AND logic is implemented via partially complementary input sequences that must come together to form a multiarm junction, which performs a strand displacement reaction to open the hairpin occluding the RBS. In the case of OR logic, the RNA containing the reporter gene contains multiple hairpins, each of which occludes a different RBS. Opening one of these hairpins via strand displacement suffices to initiate translation of the reporter protein. NOT logic is achieved by expressing a trigger RNA that binds and opens the hairpin, but which contains toeholds that enable the complementary input RNA to strip it away via strand displacement, which allows the hairpin to close and sequester the RBS. AND gates with up to four inputs and OR gates with up to six inputs have been reported.[5] Furthermore, multilevel ribocomputing logic circuits were created with up to 12 inputs,[5] demonstrating the potential complexity of ribocomputing circuits. These circuits also have practical applications, where simple ribocomputing switches have already been used for tasks in biosensing for Ebola[8] and Zika[6] virus diagnostics.

A



B

```
directive rules {
... // Insert strand graph predicates here
reaction([P], "expression", Q) :-
    P = C [rbs@X D aug^] [nil],  not hidden(rbs@X, P),
    Q = C [rbs D aug^]    [<gfp>].
}
directive parameters [ expression = 1;  bind = 0.0003 ]
directive simulation { final=5000; plots=[<gfp>] }
( 1 [<u* a1*>]
| 1 [<a2* u  >]
| 1 [<a1 a2!0 blank top^!1 rbs top^*!1 aug^ a2*!0 linker gfp>] )
```

Figure 7. Chemical reaction network and corresponding logic program code of the ribocomputing AND gate.[5] (A) Chemical reaction network. (B) Corresponding logic program code, which uses the strand graph predicates from section S1.4.

The following logic program encodes the structure of the ribocomputing AND gate (Figure 7):

```
  ( [<u* a1*>]      // input A1
  | [<a2* u>]       // input A2
  | [<a1 a2!0 blank top^!1 rbs top^*!1
      aug^ a2*!0 linker gfp>] )  // AND gate
```

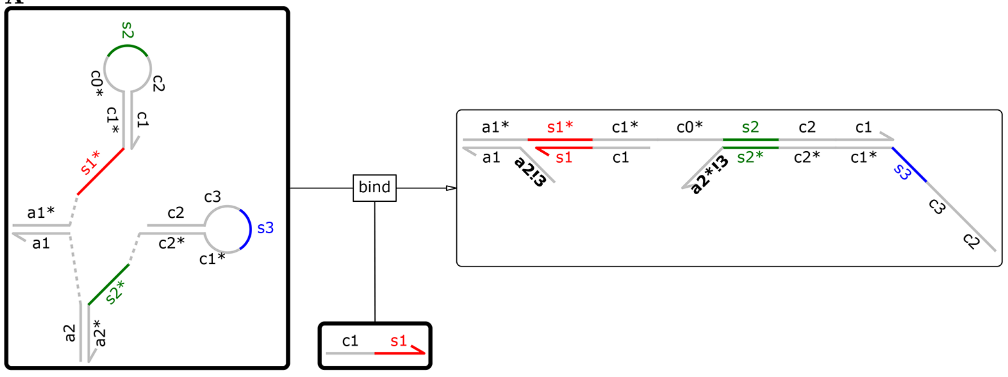
The first two strands represent input RNA strands A1 and A2, which can bind with each other on the common domain u. The AND gate hides the ribosome binding site rbs in a hairpin structure formed by the toehold top^ followed by the domain a2. The hairpin contains the start codon aug^, with an additional blank domain as padding. Initially, the AND gate only exposes a1, which binds input A1. This is still not sufficient to open the hairpin, since a2 is still hybridized. Input A2 binds to the intermediate domain u and forms a branch with an open domain a2, which in turn displaces the domain on the hairpin to open it. Because it is a toehold, the top^ domain is weak enough to unbind spontaneously, leaving rbs exposed and allowing green fluorescent protein to be produced by translation. Here we do not model ribosome binding and translation explicitly, but rather we encode both with the following rule, expressed in our logic programming language:

```
reaction([P], "expression", Q) :-
  P = C [rbs@X _ aug^] [nil],  not hidden(rbs@X, P),
  Q = C [rbs _ aug^]   [<gfp>].
```

The hidden predicate holds if the input domain, rbs in this case, occurs in a closed loop (see section S1.4 for the full definition). Protein translation is modeled by substituting the empty pattern nil in P with a new strand <gfp> in Q. Figure 7 shows the resulting compiled reaction network of the ribocomputing AND gate. This example demonstrates how our logic programming approach can encode, in a unified manner, the translation interactions required by ribocomputing systems, together with the RNA hybridization interactions required to activate the switches that enable translation to proceed.

The syntax of our language also supports more intricate systems that contain *pseudoknotted* structures. Figure 8 shows an example of such a system, based on a localized DNA hybridization chain reaction (HCR)[53] system consisting of a track to which six consecutive hairpins are bound. Here we present a simplified version of this system with only two hairpins, for illustration. Each hairpin has a toehold s^$_i$ on its stem and a toehold s^$_{i+1}$ inside its hairpin loop, where $i \in [1..2]$ for our simplified example. Toeholds s^$_{i+1}$ are sequestered provided hairpin i is closed. Strand <c1 s1^> triggers a chain reaction: it binds to the stem of the first hairpin on toehold s1^ and then displaces domain c1, thus opening the hairpin. Toehold s2^ is now free and can bind to a subsequent hairpin stem, open it by strand displacement and continue the chain reaction. Following the chain reaction, the hairpin structures are bound to each other and to the track, resulting in the formation of pseudoknots. The logic program that captures the HCR example is shown in Figure 8b. Interestingly, there is no need for custom predicates to handle pseudoknots. The hidden predicate automatically captures the fact that toeholds s$_i$ are sequestered in the hairpins but not in the stems. Note that

A



B

```
directive rules {
... // Insert strand graph predicates here
fast(P, Q) :- binds(P,Q, _, L).
}
(1000 [<c1 s1^>]
|100  [ <a2!3 a1!0> | <a1*!0 s1^* c1*!1 c0* s2^ c2 c1!1> | <c2!4 c3 s3^ c1* c2*!4 s2^* a2*!3>])
```

Figure 8. Chemical reaction network and corresponding logic program code for a localized DNA Hybridization Chain Reaction system.[53] (A) Chemical Reaction Network. (B) Corresponding logic program, which uses the strand graph predicates from section S1.4.

pseudoknots also have the effect of sequestering the domains they contain.

Visualizing pseudoknotted structures is a challenging problem,[54] which in its most general form involves laying DNA strands on topological figures such as toruses so that no edges cross. In contrast, our approach simplifies the visualization of a pseudoknotted structure by removing the bonds responsible for the pseudoknots, and marking the affected domains with bond identifiers, based on the syntax of processes (Table 1). Bonds are eliminated recursively until no pseudoknot is present and the resulting graph is planar. Affected domains are drawn in bold font and display an exclamation mark together with a bond identifier, such as a2!3 and a2*!3 in Figure 8a. This indicates that there is a bond between domains with the same bond identifier. Future improvements can be made to represent the bonds as lines instead of using identifiers. In addition, to avoid overlapping strands in the visualization, we artificially separate contiguous strands so that they can be placed more conveniently on a plane, and connect the separated pieces by dashed gray lines. Such lines have no biological meaning, but are merely a visual clue signifying that domains connected by a dashed line are contiguous.

■ DISCUSSION

In this paper we presented a logic programming language that can encode the behavior of complex nucleic acid systems using logical predicates. We showed that, thanks to the manipulation of motifs provided by patterns and contexts, our language can express a wide range of biomolecular information processing systems. We also demonstrated how previous incarnations of the Visual DSD language[27,29−31] can be encoded in this language in a unified manner. This includes elementary strand displacement rules together with a hierarchy of behavioral abstractions,[27] custom reactions,[29] localized interactions,[30] and complex topologies[31] including pseudoknots. Furthermore, our approach supports new extensions including enzyme interactions[20] and the encoding of kinetic rate hypotheses,[15] neither of which were previously supported in Visual DSD. More importantly, our approach is extensible in that new nucleic acid implementation strategies can be encoded simply by defining new logic predicates.

We envisage a number of usage scenarios for our logic programming language. In the most common scenario, we anticipate that the basic rules for nucleic acid strand displacement will be included by default, and the user will add new enzyme rules or kinetic hypotheses depending on the particular implementation strategy. More broadly, we anticipate two different classes of users: those who write predicates that define particular nucleic acid implementation strategies, and those who select from a set of existing predicates to model their systems of interest. This will allow scientists not familiar with logic programming to still take advantage of the enhanced customization that our approach enables.

We briefly compare our logic programming approach with Kappa,[55,56] which is a rule-based modeling language that captures the interactions between agents via named sites. A detailed comparison between Kappa and strand graphs has already been provided in ref 31, which also holds for our logic programming language, since it is expressive enough to encode strand graphs. While Kappa rules define patterns that can be matched to a system, our approach further allows the definition of arbitrary logical predicates to be associated with a rule, allowing increased generality. For example, our approach allows a single binding rule to be applied for all domains that are complementary, whereas Kappa would require a separate binding rule to be written for each specific domain. Furthermore, our approach allows complex topologies to be expressed using predicates, such as whether a domain is hidden inside a hairpin, or is part of a junction of arbitrary size. A similar analysis holds for other rule-based languages such as BioNetGen.[57] An interesting area of future work would be to encode Kappa in our logical framework, in order to extend the Kappa language with arbitrary logical predicates.

The contribution of this paper is to generalize our previous work on modeling DNA strand displacement systems[26,30,31] by encoding these systems in a general-purpose logic-programming language. In a similar vein, we note previous work on biological modeling using the general-purpose Python programming language in the PySB system,[58] which allows rule-based models, equivalent to Kappa models, to be encapsulated in modules and integrated with Python code. This approach allows model composition to be achieved programmatically within Python in a highly flexible manner. However, it does not directly encode support for logic programming predicates and their resolution.

In the specific context of DNA strand displacement reactions, we note that other groups have developed alternative reaction enumerators, such as Peppercorn[32] and the DyNAMiC Workbench.[33] These enumerators make slightly different choices from Visual DSD, for instance in their treatment of fast reactions and resting complexes. Our logic programming language presented here is sufficiently expressive to encode these alternative reaction enumeration schemes and to compare them in a unified framework.

Previous published work provided support for user-defined chemical reactions to be specified in Visual DSD,[29] in addition to the automatically generated strand displacement reactions. However, this approach required each individual reaction to be written explicitly, and did not support high-level logical predicates. For example, in order to encode a nickase enzyme the programmer needed to manually identify each species that should be capable of nicking, determine the resulting products, and write out all of the corresponding chemical reactions. Furthermore, the manual process of identifying these additional reactions is time-consuming and potentially error-prone. In contrast, our logic programming approach allows a single nickase rule to be defined for the appropriate nucleic acid motif, and then applies this rule to all matching species, generating the corresponding reactions automatically.

A prototype implementation of our logic programming language has been integrated in Visual DSD and is freely accessible at http://classicdsd.azurewebsites.net, together with a collection of built-in *Logic Programming* examples. Note that the performance of the web-based tool is limited, but is sufficient to reproduce the results of the paper. All of the figures in this paper were generated automatically from the software and saved as SVG files, which in some cases were further modified to improve their layout. This integration means that all of the existing simulation and analysis methods present in Visual DSD are supported by our logic programming language. We have also implemented a backward compatibility layer to support programs written in the syntax of the previous version of Visual DSD, referred to as *Classic DSD*. Programs that do not contain the keywords *directive rules* will be executed as Classic DSD programs. Conversely, adding these keywords to any Classic DSD program allows the user to define a custom semantics by providing their own semantic rules. Note that when a custom semantics is defined, any rate constants associated with specific domains in the program are ignored for consistency, since the user-defined semantic rules determine both the reactions that are generated and their corresponding rates. For example, we define a custom `find` predicate to compute reactions rates.

In terms of compilation time, for a selection of examples that are also supported by previous versions of Visual DSD,[27] our logic programming language has an average penalty of approximately 25%, including for the Join and Catalytic examples. This is not unexpected given the generality of the language. Although breath-first search is associated with poor performance for general purpose programming, we have not experienced prohibitive slowdowns in the examples we have encountered thus far. Unsurprisingly, the nucleotide Oligator presents the longest compilation time (~20 s). Future work will involve further optimizations to our approach.

As with previous versions of Visual DSD, the semantics of our logic programming language is defined as a compilation to chemical reaction networks. As a result, for systems with large numbers of molecular species and interactions it may be infeasible to generate the corresponding reaction network in its entirety. The networks generated from the systems in this paper are on the order of tens of reactions, and many of the systems that have been implemented in practice generate networks of a tractable size. However, more detailed computational models could potentially generate substantially more reactions. To address this issue, it is possible to activate a Just-In-Time (JIT) compilation mode by adding the keywords *directive jit*. This mode allows the reaction network of a system to be generated dynamically, by stochastically generating a single simulation trajectory of the system, following the approach outlined in ref 44. Future work could investigate the development of analysis methods that reason directly on a system and its corresponding rules, without the need to generate all possible reactions.

Future work will also involve additional extensions to better support nucleotide-level models, together with syntactic support for non-nucleotide species such as GFP, which are currently written as strands such as `<gfp>`. It would also be interesting to better support the distinction between DNA and RNA strands, for example by allowing tags to be associated with strands in addition to domains. Future work could also investigate a more formal definition of the language semantics, together with formal proofs that precisely capture its expressive power with respect to existing languages. A formal encoding of Kappa into our language would also be interesting to explore. As a practical matter, a useful future direction would be the inclusion of a formal *module system* whereby particular sets of predicates can be defined as self-contained modules and easily loaded into a model with a single command. This would enable the creation of a *standard library* of commonly used predicate sets for well-known systems such as DNA strand displacement or the PEN-DNA toolbox, and would also enhance communication of models between researchers by providing a common interchange format for their definitions. Other notable examples such as the metastable DNA fuel[59] can be encoded in our framework, but in this case the semantics must take into consideration biophysical constraints, that in this particular case keep the metastable fuel in a kinetic trap. We speculate that a semantics that takes molecular geometry into account, such as the one in ref 52 can be encoded in our language and might be suitable in this case, but such a discussion is out of the scope of the present paper.

METHODS

SLDNF Resolution. Given a goal A, SLDNF resolution instantiates the root of a search tree with A. It then tries to match A with the head of a Horn clause C_i in the logic program using unification. For any match of C_i with substitution θ_i found by unification, a new child to the root node is created, where A is substituted with the body of C_i after applying θ_i. Resolution proceeds recursively on the children, until a leaf node with no goals to expand is discovered and a solution θ is found, where θ is the composition of all substitutions θ_i computed by unification from the root to the leaf. If no clause matches a goal then resolution fails, that node is discarded and another node is chosen. In the general case, clause definitions in a logic program can be recursive and may cause the SLDNF resolution to build an infinite tree. An example of a recursive logic program is the factorial function, which can be encoded as follows:

```
factorial(0, 1).
factorial(N, M) :-
  N > 0,
  N′ is N - 1,
  factorial(N′, M′),
  M is M′ * N.
```

The first line of the program encodes the fact that factorial of 0 is 1, and is the base case of the program. The subsequent clause computes the factorial of a positive number N recursively in terms of $N - 1$. The SLDNF resolution tree therefore has $N + 1$ nodes from root to leaf. Resolution fails for negative numbers, since the predicate `N > 0` (which is parsed as a predicate greater `(N, 0)`) does not hold in such cases. Without this predicate, SLDNF resolution would create an infinite resolution tree. SLDNF can also prove negative predicates, using what is known as *negation-as-failure*. A negative predicate holds when its SLDNF tree is *finite* and all of its nodes result in failed goals. If the SLDNF tree is infinite, SLDNF will not terminate even if all of its goals fail.

Language Semantics. We briefly summarize the semantics of our logic programming language. Unification of domains and bonds proceeds by syntactic decomposition. Patterns $\pi_1...\pi_N$ can be substituted by patterns $\pi'_1...\pi'_N$ of a similar kind in a context. For example, a sequence S_1S_2 can be replaced by S_1> | <S_2 to model nicking by nickase, and a strand <S> can be replaced by $\varnothing$ to model degradation by exonuclease. 3′ and ′5 ends are only replaced by the same kind of pattern.

Context substitution follows the Double Pushout (DPO) approach from the theory of graph grammars.[38] In DPO, graph transformations are sound as long as no dangling edge is removed, and no node is added and removed at the same time. The first condition translates to checking that a bond is always removed from both of the domains it connects, and that patterns are substituted with similar patterns. The second condition is always satisfied by well-formedness: by definition of our contexts no two holes can overlap, therefore no part of a system can be added and removed at the same time. Unification fails at run-time whenever a predicate violates these conditions.

There are many ways in our syntax to express what is nominally the same process. For example, process <a!0 b!1> | < b*!1 a*!0> is essentially the same process as <b*!3a*!2> | < a!2b!3>, except for the fact that bond 0 is called 2, bond 1 is called 3, and the order of the strands is inverted. In order to ignore such syntactical differences and focus on the semantics, our theory also allows equations $P \doteq Q$ to hold when the processes P and Q are *equivalent*, written $P \equiv Q$. We allow standard process calculus equivalences, such as commutativity ($P \mid Q \equiv Q \mid P$), distributivity ($P \mid (Q \mid R) \equiv (P \mid Q) \mid R$) and absorption ($P \mid 0 \equiv P$). Processes are also considered equivalent up to renaming of bonds.

The surrounding context in which a logical variable occurs is often sufficient to identify its kind. For example, X in $X\,[\pi_1]$ is a context variable; D in $D!i$ is a domain variable. If this is not possible, we assume that the variable has the most general kind possible in the syntax. For example, we say that D in $D@l$ is a site variable rather than a domain variable.

Contexts are the core mechanism to programmatically identify and manipulate DNA motifs. Motifs are identified when a clause defines equality constraints of the form $P \doteq X[\pi_1]\cdots[\pi_n]$. Our unification theory solves such equations by finding all well-formed contexts C_N and variable substitutions θ for the logical variables in $\pi_1...\pi_N$ such that $P = C_N[\theta(\pi_1)]\cdots[\theta(\pi_n)]$. In our theory $\doteq$ is not a congruence: we do not allow equations such as $P \doteq X[\pi_1]\ ...[\pi_N] \mid Q$ for efficiency reasons, much like how unification for arithmetic operations is handled in Prolog.

Process Canonical Form. Process equivalence implies that the same process can be written in many different forms. To verify whether two processes P and Q are equivalent, we define the *canonical form* of a process as a function $canon(-)$ such that $canon(P) = canon(Q)$ if and only if $P \equiv Q$. The practical advantage of such a function is that an implementation of our logic programming language can store all processes in canonical form and just test them for equality during reaction enumeration. The algorithm that calculates the canonical form of a process is a variation of the coinductive bisimulation algorithms for concurrent process algebras such as CCS,[39] together with sorting ideas inspired by McKay's algorithm for graph isomorphism.[60] We call our algorithm the *bisimulation sort*.

At the beginning of bisimulation sort, strands are sorted *lexicographically*, that is, strands are compared point-wise based on their sites, and are sorted by domain names, site kind (e.g., whether they have a bond or not) and overall length. Because of α-equivalence, bond names are not used at this stage. Lexicographically equivalent strands are grouped together into an *equivalence class*. The output of this initial setup is thus an ordered set of lexicographically sorted strands. The end goal of bisimulation sort is to sort each strand inside these equivalence classes in a canonical form as well, so that the whole process is then canonical.

Although some sites are indistinguishable from others based on their names or kind alone, they might be distinguished by their context. For example, a process might contain two lexicographically equivalent strands (i.e., same domain names, same site kinds and same length), but one strand might have a site d!0 bound to the single domain strand <d*!0> , while the other strand might have a domain d!1 at the same position as the former strand, but bound to a different, bigger complex, such as <d*!1 e*!2 > | <e!2>.

As a further example, <a!0 b!1> and <a!2 b!3> are lexicographically equivalent in <a!0 b!1> | <a!2 b!3> | <b*!3 a*!2 b*!1 a*!0>, but <a!0 b!1> is bound at the 3′ end of <b*!3 a*!2 b*!1 a*!0>, while <a!2 b!3> is bound at the 5′ end. This is another kind of distinction that defines a unique canonical form.

Finally, sometimes strands cannot be differentiated by the context. For example in <a!0 a*!1> | <a!1 a*!0> , the two strands are not only lexicographically equivalent, but they are also perfectly complementary and symmetrical: any permutation of bonds produces the same system. In such cases the order of strands and bond names are uninfluential. Our bisimulation sort algorithm is a variant of the Paige-Tarjan block refinement algorithm [ref 61, chapter 6] to find strands equivalent up to bond names.

Bisimulation sort computes the canonical form coinductively, by continuing to split blocks until no more splitters are available. If a block contains two or more strands, these strands are isomorphic. After this sorting phase is finished, all bonds are renamed in increasing order of appearance in the sorted strands.

Compilation Performance. We report the average compilation time for the logic programming examples presented in the text. We measured the time it takes to parse, compute, and output the chemical reaction network of each example, averaged over 10 iterations, running on an Intel Xeon E5-1620 processor with four cores @ 3.60 GHz. As a comparison, we also compared the compilation time of the Join and Catalytic examples in the previous version of Visual DSD, both in Infinite and Detailed mode (Table 4). In comparison with the previous version of Visual DSD, our implementation shows a compilation time penalty of approximately 25%. This penalty is not unexpected, given the generality of the language. There is also room for improvement, given that we prioritized soundness over performance during the compiler implementation. In an earlier implementation of our language, performance analysis using a profiler highlighted two bottlenecks in the system. After resolving these, we gained a 10-fold improvement in performance. We conjecture that further improvements are possible.

Table 4. Compilation Times

example	logic DSD (s)	visual DSD (s)
ribocomputing OR	5.8653	
ribocomputing AND	2.2722	
localized HCR	4.784	
cargo-sorting robot (classic DSD semantics, no merging)	9.7368	
cargo-sorting robot (strand graphs semantics, merging)	6.6322	
Oligator	3.1413	
Oligator, nucleotides	22.2991	
join, infinite	1.9866	1.4772
join, detailed	1.7183	1.5080
catalytic (infinite)	1.9654	1.3705
catalytic (detailed)	1.7545	1.3741

■ ASSOCIATED CONTENT

Ⓢ Supporting Information

The Supporting Information is available free of charge on the ACS Publications website at DOI: 10.1021/acssynbio.8b00229.

Strand displacement, enzyme, localized, and strand graph rules; join, catalytic, oligator, nucleotide oligator, cargo-sorting DNA robot, and enzymatic walker examples (PDF)

■ AUTHOR INFORMATION

Corresponding Author

*Email: andrew.phillips@microsoft.com.

ORCID

Andrew Phillips: 0000-0001-9725-1073

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

Filippo Polo implemented a library to visualize arbitrary DNA complexes, and connected the logic programming language to the existing Visual DSD user interface. Boyan Yordanov, Filippo Polo, and Rasmus Petersen implemented a library to visualize nonpseudoknotted DNA complexes. This material is based upon work supported by the National Science Foundation under Grant Nos. 1518861, 1525553, and 1814906 to M.R.L.

■ REFERENCES

(1) Surana, S., Bhat, J. M., Koushika, S. P., and Krishnan, Y. (2011) An autonomous DNA nanomachine maps spatiotemporal pH changes in a multicellular living organism. *Nat. Commun. 2*, 340.

(2) Schnitzbauer, J., Strauss, M. T., Schlichthaerle, T., Schueder, F., and Jungmann, R. (2017) Super-resolution microscopy with DNA-PAINT. *Nat. Protoc. 12* (6), 1198−1228.

(3) Meng, W., Muscat, R. A., McKee, M. L., Milnes, P. J., El-Sagheer, A. H., Bath, J., Davis, B. G., Brown, T., O'Reilly, R. K., and Turberfield, A. J. (2016) An autonomous molecular assembler for programmable chemical synthesis. *Nat. Chem. 8*, 542−548.

(4) Gu, H., Chao, J., Xiao, S.-J., and Seeman, N. C. (2010) A proximity-based programmable DNA nanoscale assembly line. *Nature 465*, 202−205.

(5) Green, A. A, Kim, J., Ma, D., Silver, P. A, Collins, J. J, and Yin, P. (2017) Complex cellular logic computation using ribocomputing devices. *Nature 548* (7665), 117.

(6) Pardee, K., Green, A. A., Takahashi, M. K., Braff, D., Lambert, G., Wook Lee, J., Ferrante, T., Ma, D., Donghia, Ni., Fan, M., Daringer, N. M., Bosch, I., Dudley, D. M., O'Connor, D. H., Gehrke, L., and Collins, J. J. (2016) Rapid, low-cost detection of Zika virus using programmable biomolecular components. *Cell 165*, 1255−1266.

(7) Green, A. A., Silver, P. A., Collins, J. J., and Yin, P. (2014) Toehold switches: De-novo-designed regulators of gene expression. *Cell 159* (4), 925−939.

(8) Pardee, K., Green, A. A., Ferrante, T., Cameron, D. E., DaleyKeyser, A., Yin, P., and Collins, J. J. (2014) Paper-based synthetic gene networks. *Cell 159* (4), 940−954.

(9) Groves, B., Chen, Y.-J., Zurla, C., Pochekailov, S., Kirschman, J. L., Santangelo, P. J., and Seelig, G. (2016) Computing in mammalian cells with nucleic acid strand exchange. *Nat. Nanotechnol. 11*, 287−294.

(10) Chen, Y.-J., Groves, B., Muscat, R. A., and Seelig, G. (2015) DNA nanotechnology from the test tube to the cell. *Nat. Nanotechnol. 10*, 748−760.

(11) Zhang, D. Y., and Seelig, G. (2011) Dynamic DNA nanotechnology using strand-displacement reactions. *Nat. Chem. 3*, 103−113.

(12) Soloveichik, D., Seelig, G., and Winfree, E. (2010) DNA as a universal substrate for chemical kinetics. *Proc. Natl. Acad. Sci. U. S. A. 107* (12), 5393−5398.

(13) Qian, L., and Winfree, E. (2011) Scaling up digital circuit computation with DNA strand displacement cascades. *Science 332*, 1196−1201.

(14) Qian, L., Winfree, Er., and Bruck, J. (2011) Neural network computation with DNA strand displacement cascades. *Nature 475*, 368−372.

(15) Chen, Y.-J., Dalchau, N., Srinivas, N., Phillips, A., Cardelli, L., Soloveichik, D., and Seelig, G. (2013) Programmable chemical controllers made from DNA. *Nat. Nanotechnol. 8*, 755−762.

(16) Srinivas, N., Parkin, J., Seelig, G., Winfree, E., and Soloveichik, D. (2017) Enzyme-free nucleic acid dynamical systems. *Science 358* (6369), 2052.

(17) Chatterjee, G., Dalchau, N., Muscat, R. A., Phillips, A., and Seelig, G. (2017) A spatially localized architecture for fast and modular DNA computing. *Nat. Nanotechnol. 12*, 920−927.

(18) Thubagere, A. J., Li, W., Johnson, R. F., Chen, Z., Doroudi, S., Lee, Y. L., Izatt, G., Wittman, S., Srinivas, N., Woods, D., Winfree, E., and Qian, L. (2017) A cargo-sorting DNA robot. *Science 357* (6356), e6558.

(19) Bui, H., Shah, S., Mokhtar, R., Song, T., Garg, S., and Reif, J. (2018) Localized DNA hybridization chain reactions on DNA origami. *ACS Nano 12* (2), 1146−1155.

(20) Montagne, K., Plasson, R., Sakai, Y., Fujii, T., and Rondelez, Y. (2011) Programming an *in vitro* DNA oscillator using a molecular networking strategy. *Mol. Syst. Biol. 7* (466), 466.

(21) Fujii, T., and Rondelez, Y. (2013) Predator-prey molecular ecosystems. *ACS Nano 7* (1), 27−34.

(22) Baccouche, A., Montagne, K., Padirac, A., Fujii, T., and Rondelez, Y. (2014) Dynamic DNA-toolbox reaction circuits: A walkthrough. *Methods 67* (2), 234−249.

(23) Montagne, K., Gines, G., Fujii, T., and Rondelez, Y. (2016) Boosting functionality of synthetic DNA circuits with tailored deactivation. *Nat. Commun. 7*, 13474.

(24) Kim, J., and Winfree, E. (2011) Synthetic *in vitro* transcriptional oscillators. *Mol. Syst. Biol. 7*, 465.
(25) Weitz, M., Kim, J., Kapsner, K., Winfree, E., Franco, E., and Simmel, F. C. (2014) Diversity in the dynamical behaviour of a compartmentalized programmable biochemical oscillator. *Nat. Chem. 6*, 295−302.
(26) Phillips, A., and Cardelli, L. (2009) A programming language for composable DNA circuits. *J. R. Soc., Interface 6*, S419−S436.
(27) Lakin, M. R., Youssef, S., Cardelli, L., and Phillips, A. (2012) Abstractions for DNA circuit design. *J. R. Soc., Interface 9* (68), 470−486.
(28) Lakin, M. R., Youssef, S., Polo, F., Emmott, S., and Phillips, A. (2011) Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics 27* (22), 3211−3213.
(29) Yordanov, B., Kim, J., Petersen, R. L., Shudy, A., Kulkarni, V. V., and Phillips, A. (2014) Computational design of nucleic acid feedback control circuits. *ACS Synth. Biol. 3* (8), 600−616.
(30) Lakin, Matthew R., Petersen, R., Gray, Kathryn E., and Phillips, A. (2014) Abstract modelling of tethered DNA circuits. In *Proceedings of the 20th International Conference on DNA Computing and Molecular Programming (DNA20)*, Lecture Notes in Computer Science (Murata, S., and Kobayashi, S., Eds.) Vol. *8727*, pp 132−147, Springer International Publishing.
(31) Petersen, R. L., Lakin, M. R., and Phillips, A. (2016) A strand graph semantics for DNA-based computation. *Theor. Comput. Sci. 632*, 43−73.
(32) Grun, C., Sarma, K., Wolfe, B., Shin, S. W., and Winfree, E. (2014) A domain-level DNA strand displacement reaction enumerator allowing arbitrary non-pseudoknotted secondary structures. Verification of Engineered Molecular Devices and Programs (VEMDP), July 17, Vienna, Austria.
(33) Grun, C., Werfel, J., YuZhang, D., and Yin, P. (2015) DyNAMiC Workbench: an integrated development environment for dynamic DNA nanotechnology. *J. R. Soc., Interface 12*, 20150580.
(34) Aubert, N., Mosca, C., Fujii, T., Hagiya, M., and Rondelez, Y. (2014) Computer-assisted design for scaling up systems based on DNA reaction networks. *J. R. Soc., Interface 11* (93), 20131167.
(35) van Roekel, H W. H., Meijer, L H. H., Masroor, S., Félix Garza, Z C., Estévez-Torres, A, Rondelez, Y., Zagaris, A., Peletier, Mark A., Hilbers, Peter A. J., and de Greef, Tom F. A. (2015) Automated design of programmable enzyme-driven DNA circuits. *ACS Synth. Biol. 4* (6), 735−745.
(36) Nilsson, U., and Maluszynski, J. (1995) *Logic, Programming, and PROLOG*, 2nd ed., John Wiley & Sons, Inc., New York, NY, USA.
(37) Doets, K. (1994) *From Logic to Logic Programming*, MIT Press: Cambridge, MA, USA.
(38) Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., and Loewe, M. (1996) Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. Technical report, University of Pisa.
(39) Rozenberg, G., Ed. (1997) Foundations. *Handbook of Graph Grammars and Computing by Graph Transformations*, Vol. *1*, World Scientific.
(40) Yurke, B., Turberfield, A. J., Mills, A. P., Jr, Simmel, F. C., and Neumann, J. L. (2000) A DNA-fuelled molecular machine made of DNA. *Nature 406*, 605−608.
(41) Cook, M., Soloveichik, D., Winfree, E., and Bruck, J. (2009) *Programmability of Chemical Reaction Networks*, Springer, Berlin, Heidelberg, pp 543−584.
(42) Qian, L., Soloveichik, D., and Winfree, E. (2011) Efficient Turing-Universal Computation with DNA Polymers. In *Lecture Notes in Computer Science*, (Sakakibara, Y., and Mi, Y., Eds.) Vol. *6518*, pp 123−140, Springer-Verlag.
(43) Sangiorgi, D., and Walker, D. (2001) *PI-Calculus: A Theory of Mobile Processes*, Cambridge University Press, New York, NY, USA.
(44) Lakin, M. R., Paulevé, L., and Phillips, A. (2012) Stochastic simulation of multiple process calculi for biology. *Theor. Comput. Sci. 431*, 181−206.
(45) Gines, G., Zadorin, A. S., Galas, J.-C., Fujii, T., Estevez-Torres, A., and Rondelez, Y. (2017) Microscopic agents programmed by DNA circuits. *Nat. Nanotechnol. 12*, 351−359.
(46) Gibson, D. G., Young, L., Chuang, R.-Y., Venter, J. C., Hutchison, C. A., III, and Smith, H. O. (2009) Enzymatic assembly of DNA molecules up to several hundred kilobases. *Nat. Methods 6*, 343−345.
(47) Tian, Y., He, Y., Chen, Y., Yin, P., and Mao, C. (2005) A DNAzyme that walks processively and autonomously along a one-dimensional track. *Angew. Chem., Int. Ed. 44*, 4355−4358.
(48) Green, S. J., Bath, J., and Turberfield, A. J. (2008) Coordinated chemomechanical cycles: A mechanism for autonomous molecular motion. *Phys. Rev. Lett. 101*, 238101.
(49) Bath, S. J., Green, K. E., Allen, K. e., and Turberfield, A. J. (2009) Mechanism for a directional, processive, and reversible DNA motor. *Small 5* (13), 1513−1516.
(50) Lund, K., Manzo, A. J., Dabby, N., Michelotti, N., Johnson-Buck, A., Nangreave, J., Taylor, S., Pei, R., Stojanovic, Milan N., Walter, NilsG̃., Winfree, E., and Yan, H. (2010) Molecular robots guided by prescriptive landscapes. *Nature 465*, 206−2010.
(51) Wickham, S. F. J., Bath, J., Katsuda, Y., Endo, M., Hidaka, K., Sugiyama, H., and Turberfield, A. J. (2012) A DNA-based molecular motor that can navigate a network of tracks. *Nat. Nanotechnol. 7*, 169−173.
(52) Lakin, M. R., and Phillips, A. (2017) Automated, constraint-based analysis of tethered DNA nanostructures. In *DNA Computing and Molecular Programming*, (Brijder, R., and Qian, L., Eds.) pp 1−16, Springer International Publishing.
(53) Bui, H., Miao, V., Garg, S., Mokhtar, R., Song, T., and Reif, J. (2017) Design and analysis of localized DNA hybridization chain reactions. *Small 13* (12), 1602983.
(54) Shabash, B., and Wiese, K. C. (2017) RNA visualization: Relevance and the current state-of-the-art focusing on pseudoknots. *IEEE/ACM Trans. Comput. Biol. Bioinf. 14* (3), 696−712.
(55) Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J., Caires, Luís, and Vasconcelos, Vasco T. (2007) Rule-based modelling of cellular signalling. *CONCUR 2007−Concurrency Theory*, 17−41.
(56) Danos, V., Feret, J., Fontana, W., Harmer, R., Hayman, J., Krivine, J., Thompson-Walsh, C., and Winskel, G. (2012) Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*, Leibniz International Proceedings in Informatics (LIPIcs), (D'Souza, D., Kavitha, T., and Radhakrishnan, J., Eds.) Vol. *18*, pp 276−288, Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.
(57) Harris, L. A., Hogg, J. S., Tapia, J.-J., Sekar, J. A. P., Gupta, S., Korsunsky, I., Arora, A., Barua, D., Sheehan, Robert P., and Faeder, James R. (2016) Bionetgen 2.2: advances in rule-based modeling. *Bioinformatics 32* (21), 3366−3368.
(58) Lopez, C. F., Muhlich, J. L., Bachman, J. A., and Sorger, P. K. (2013) Programming biological models in Python using PySB. *Mol. Syst. Biol. 9* (1), 646.
(59) Seelig, G., Yurke, B., and Winfree, E. (2006) Catalyzed relaxation of a metastable DNA fuel. *J. Am. Chem. Soc. 128* (128), 12211−12220. PMID: 16967972.
(60) McKay, B. D., and Piperno, A. (2014) Practical graph isomorphism, II. *Journal of Symbolic Computation 60*, 94−112.
(61) Bergstra, J. A. (2001) *Handbook of Process Algebra*, Elsevier Science Inc., New York, NY, USA.