Rule-based design of computational DNA devices

Carlo Spaccasassi[1], Matthew R. Lakin[2], Andrew Phillips[1]

[1]Microsoft Research, Cambridge, UK
[2]Department of Computer Science, University of New Mexico, Albuquerque, NM, USA

1 Introduction

As the state of the art in DNA nanotechnology continues to develop, highly sophisticated computational molecular devices are being designed and subsequently implemented in DNA. These devices employ a broad range of implementation strategies to perform computation, including DNA strand displacement, localisation to substrates, and the use of enzymes with polymerase, nickase and exonuclease functionality. However, existing computational design tools are unable to account for these different strategies in a unified manner.

This paper presents a programming language that allows a broad range of computational DNA systems to be expressed and analyzed. We define a semantic framework that allows DNA molecular motifs to be expressed as sub-graphs, and automatically identifies matching motifs in the full system, in order to apply a specified transformation expressed as a *rule*. The framework also supports the definition of *predicates*, which provide additional constraints in order for rules to apply. The framework is sufficiently expressive to encode the semantics of DNA strand displacement systems with complex topologies, together with computation performed by a broad range of enzymes.

Our language, called *Rules DSD*, is a logic programming language that extends Prolog with a novel equational theory to express DNA molecular motifs in a system. Molecular motifs are interpreted as sub-graphs occurring in a system of strand graphs[6]. Transformations of such motifs are safely handled in the Single Pushout approach, drawing from the theory of graph grammars[1]. The syntax and semantics of the language is presented in Section 2. Several encodings of molecular systems are provided in Section 3, including ribocomputing logical circuits[2].

2 Language definition

We extend the syntax of strand graphs[6] with *tags* and *logical variables*:

$dom ::= d \mid d^* \mid d\hat{} \mid d^*\hat{} \mid X$		Domains
$bond ::= i \mid X$		Bonds
$tag ::= n \mid c \mid f(tag_1, \ldots, tag_N)$		Tags: number n, string c, structure
$site ::= dom \mid dom!bond \mid \{dom : tag\} \mid \{dom!bond : tag\}$		Sites
$S ::= site_1 \ldots site_N$		Consecutive sites, $N \geq 1$
$P ::= <S_1> \mid \ldots \mid <S_N>$		Processes, $N \geq 0$

The basic abstraction of our language is the *domain*, which is a nucleotide sequence *dom* orthogonal to all other domains in a given system. We indicate domains with lower-case letters $(d, e, ...)$, complementary domains with a star (d^*) and toeholds with a caret $(d\hat{})$. A strand $< S >$ is modeled as a non-empty list S of domains ordered from the 5' end to the 3' end. A *process* P is a possibly empty set of strands. Tags model particular properties or states of a domain. Logical variables X are placeholders for concrete domains or bonds.

DNA molecular motifs are expressed in terms of *patterns* occurring in strand *contexts*:

$loc ::= l \mid X$		Locations
$patternSite ::= site \mid site@loc \mid X$		Indexed site
$S ::= patternSite_1 ... patternSite_N$		Indexed sites, $N \geq 1$
$\pi ::= <S> \mid <S \mid S> \mid S \mid S> \mid <S \mid nil$		Patterns
$ctx_N ::= \mathcal{C}_N \mid X$		N-holes context, $N \geq 1$
$\mathcal{C}_N ::= [\cdot]_i \mid P \mid <S\ \mathcal{C}_N \mid S\ \mathcal{C}_N \mid \mathcal{C}_N\ S> \mid \mathcal{C}_N \mid \mathcal{C}_N$		Context instance, $1 \leq i \leq N$

A *pattern* π is a motif that may occur in one or more strands. Pattern $< S >$ indicates a strand whose sequence of domains matches S exactly. The 3' and 5'-end patterns $S >$ and $< S$ indicate the respective ends of a strand. The segment pattern S matches a sub-sequence of domains anywhere in a strand. The nicking pattern $S > | < S$ indicates the two ends where the nick occurs. Pattern *nil* is the empty set of strands, and is used to model strand creation or deletion. Domains in a pattern are also assigned a unique location identifier *loc*.

A *context* $ctx_N[\pi_1]...[\pi_N]$ is a "process with N holes"[7], where each hole $[\cdot]_i$ is filled by a pattern π_i for $i \in N$. An example of context is $\mathcal{C}_2 =< d_1\ d_2\ [\cdot]_2\ | < d_4\ [\cdot]_1\ d_6 >$. A context $\mathcal{C}_N$ is well-formed whenever it contains exactly N holes, and each hole $[\cdot]_i$ occurs exactly once. We only consider well-formed contexts. The context body can be a logical variable X; the context $X[\pi_i]...[\pi_N]$ specifies that a process must contain patterns π_1 ... π_N to match.

Apart from contexts, Rules DSD logic programs follow the standard syntax and semantics of Prolog [5]:

$t ::= X \mid n \mid c \mid \pi \mid ctx_N[\pi_1]\dots[\pi_N]$	Terms: numbers n, strings c, patterns, contexts,
$\quad \mid f(t_1,\dots,t_n) \mid [t_1;\ \dots;\ t_n]$	functors, lists
$A ::= p(t_1,\dots,t_n)$	Atomic predicate
$L ::= A \mid \texttt{not}\ A$	Literal
$C ::= A :- L_1,\dots,L_n$	Definite clause

Contexts are the core mechanism to programmatically identify and manipulate DNA motifs. Motifs are identified when a clause defines equality constraints of the form $P \doteq X[\pi_1]\dots[\pi_n]$. Our unification theory solves such equations by finding all well-formed contexts $\mathcal{C}_N$ and variable substitutions θ for the logical variables in $\pi_1 \dots \pi_N$ such that $P = \mathcal{C}_N[\theta(\pi_1)]\dots[\theta(\pi_n)]$. Patterns π_1 ... π_N can be substituted by some other patterns π'_1 ... π'_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, or a strand $< S >$ by *nil* to model degradation by exonuclease. 3' and '5 ends are only replaced by the same kind of pattern.

Context substitution follows the Single Pushout (SPO) approach from the theory of graph grammars[1]. In SPO, 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 the domains it connects, and that patterns are substituted with similar patterns. The second condition is always satisfied by well-formedness: in such contexts no two holes overlap, therefore no part of a system can be added and removed at the same time. Unification fails at run-time whenever a predicate breaks these conditions.

Reaction enumeration follows the approach delineated in [3]. Reactions are specified by the special clause $reaction([P_1;...;P_N], R, Q)$, which specifies N-molecular reactions from P_1 ... P_N reactants to a product Q with rate R. Each species P_i is guaranteed to be a connected component of strands; Q is automatically split into species.

3 Rule modeling

3.1 Elementary DNA strand displacement rules

DNA strand displacement rules (omitting the symmetrical rule for displace) can be expressed as follows:

```
reaction([P1;P2], "bind",Q) :- P1 = C1 [D], P2 = C2 [D'], compl(D, D'),
                                 Q = C1 [D!i] | C2 [D'!i], freshBond(D!i, P1|P2).
reaction([P],"displace",Q) :- P = C [E!j D] [D!i] [D'!i E'!j],
                              Q = C [E!j D!i] [D] [D'!i E'!j].
reaction([P],"unbind",Q) :- P = C [D!i] [D'!i], toehold(D),
                            Q = C [D] [D'], not adj(D!i,_,P).
adj(D!i,E!j,P) :- P = C [D!i E!j] [E'!j D'!i].
adj(D!i,E!j,P) :- P = C [E!j D!i] [D'!i E'!j].
```

The first reaction describes the binding of two complexes. The rule looks for unbound complementary domains `D` and `D'` in the input species `P1` and `P2`, where `compl(D, D')` is an inbuilt predicate that tests complementarity. The resulting species Q adds a new bond `i` to both domains and composes the two contexts `C1[D!i]` and `C2[D'!i]`. Rate parameters such as `"unbind"` are mapped to concrete rates elsewhere in the program. The displace rule models the displacement of a bound domain `D!i` by an unbound domain `D` when the strands are connected on the same backbone `E' D'`. The last rule models the spontaneous unbinding of toeholds, when not anchored.

3.2 Enzymatic reactions

The following rules encode the enzymes from the PEN toolbox[4], which includes polymerase, nickase and exonuclease. The user-defined predicate `recognition` indicates a recognition site of nickase. The exonuclease rule makes use of tags to avoid degrading phosphorothioated domains.

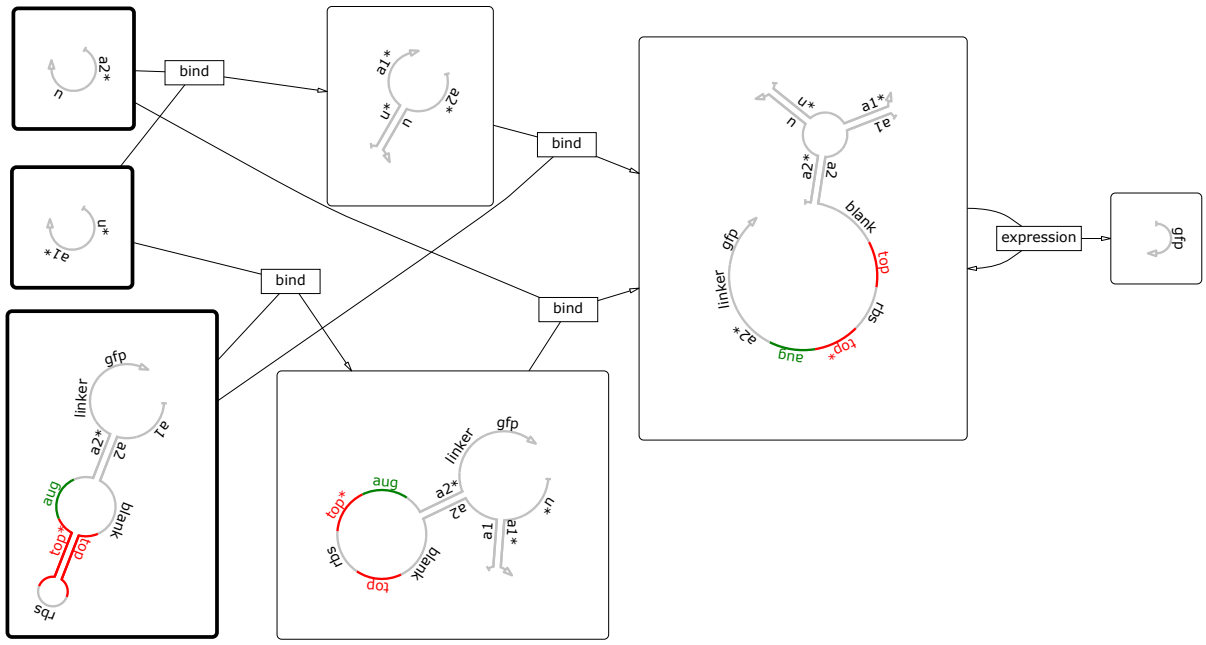

Fig. 1: Reaction network of a ribocomputing AND gate with input triggers $< u* \ a1* >$ and $< a2^* \ u >$. After the triggers bind to the AND gate, the ribosome binding site *rbs* is exposed and translation of $< gfp >$ is activated in a catalytic loop.

```
reaction([P], "polymerase", Q) :- P = C(d!j>,     E*   d*!j),
                                  Q = C(d!j E!k>, E*!k d*!j).
reaction([P], "nickase", Q) :- P = C(D!j E!k,     E*!k D*!j), recognition([D]),
                               Q = C(D!j> | <E!k, E*!k D*!j).
reaction([P], "exonuclease", Q) :-  P = C(<d>), Q = C(nil).
reaction([P], "exonuclease", Q) :-  P = C[<A B], unbound(A), Q = C[<B], not protected(A).
protected({ _ : "phosphorothioated"}).
```

3.3 Ribocomputing AND gate

As a final example we encoded the ribocomputing AND gate [2]. Ribocomputing devices are structures that inhibit the expression of an output gene by hiding its ribosome binding site in a hairpin. The hairpin opens only when a particular input logic is available. Figure 1 shows the resulting reaction network.

```
reaction([P], "expression", Q) :-
  P = C [rbs T augˆ B] [nil], unbound(T), unbound(B),  Q = C [rbs T augˆ B] [<gfp>].
...
[<u* a1*>]      // input A1
| [<a2* u>]     // input A2
| [<a1 a2!0 blank topˆ!1 rbs topˆ*!1 augˆ a2*!0 linker gfp>] // AND gate
```

References

1. Ehrig, H., et al.: Algebraic approaches to graph transformation - part II: single pushout approach and comparison with double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, pp. 247–312. World Scientific (1997)
2. Green, A., et al.: Complex cellular logic computation using ribocomputing devices **548** (07 2017)
3. Lakin, M.R., Paulevé, L., Phillips, A.: Stochastic simulation of multiple process calculi for biology. Theor. Comput. Sci. **431**, 181–206 (may 2012). https://doi.org/10.1016/j.tcs.2011.12.057
4. Montagne, K., Plasson, R., Sakai, Y., Fujii, T., Rondelez, Y.: Programming an in vitro DNA oscillator using a molecular networking strategy. Mol. Syst. Biol. **7**(466), 466 (feb 2011). https://doi.org/10.1038/msb.2010.120
5. Nilsson, U., Maluszynski, J.: Logic, Programming, and Prolog. John Wiley & Sons, Inc., New York, NY, USA, 2nd edn.
6. Petersen, R.L., Lakin, M.R., Phillips, A.: A strand graph semantics for DNA-based computation. Theor. Comput. Sci. **632**, 43–73 (2016). https://doi.org/10.1016/j.tcs.2015.07.041
7. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA