# 26th International Conference on DNA Computing and Molecular Programming

**DNA 26, September 14–17, 2020, Oxford, UK (Virtual Conference)**

Edited by

Cody Geary

Matthew J. Patitz


LIPIcs – Vol. 174 – DNA 26 www.dagstuhl.de/lipics

*Editors*

**Cody Geary**
Interdisciplinary Nanoscience Centre, University of Aarhus, Denmark
codyge@gmail.com

**Matthew J. Patitz**
Department of Computer Science and Computer Engineering,
University of Arkansas, Fayetteville, AR, USA
patitz@uark.edu

*ACM Classification 2012*
Theory of computation → Models of computation; Applied computing → Molecular structural biology; Applied computing → Biological networks; Information systems → Information storage systems

**ISBN 978-3-95977-163-4**

*Published online and open access by*
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-163-4.

*Publication date*
September, 2020

*Bibliographic information published by the Deutsche Nationalbibliothek*
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.



Digital Object Identifier: 10.4230/LIPIcs.DNA.2020.0

## LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.



**ISSN 1868-8969**

**https://www.dagstuhl.de/lipics**

# Contents



## Regular Papers



26th International Conference on DNA Computing and Molecular Programming (DNA 26).
Editors: Cody Geary and Matthew J. Patitz
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# Preface

This volume contains the papers presented at DNA 26: the 26th International Conference on DNA Computing and Molecular Programming. The conference was originally scheduled to be held at the University of Oxford, but due to the COVID-19 pandemic it was changed to an online format. The virtual conference was held during September 14-17, 2020, and was organized under the auspices of the International Society for Nanoscale Science, Computation, and Engineering (ISNSCE). The DNA conference series aims to draw together researchers from the fields of mathematics, computer science, physics, chemistry, biology, and nanotechnology to address the analysis, design, and synthesis of information-based molecular systems.

Papers and presentations were sought in all areas that relate to biomolecular computing, including, but not restricted to: algorithms and models for computation on biomolecular systems; computational processes in vitro and in vivo; molecular switches, gates, devices, and circuits; molecular folding and self-assembly of nanostructures; analysis and theoretical models of laboratory techniques; molecular motors and molecular robotics; information storage; studies of fault-tolerance and error correction; software tools for analysis, simulation, and design; synthetic biology and in vitro evolution; and applications in engineering, physics, chemistry, biology, and medicine.

Authors who wished to orally present their work were asked to select one of two submission tracks: Track A (full paper) or Track B (one-page abstract with supplementary document). Track B is primarily for authors submitting experimental results who plan to submit to a journal rather than publish in the conference proceedings. We received 52 submissions for oral presentations: 25 submissions to Track A and 27 submissions to Track B. Each submission was reviewed by at least three reviewers, with several reviewed by four reviewers. The Program Committee accepted 11 papers for Track A (44%) and 11 papers for Track B (41%). Additionally, the Program Committee reviewed and accepted 37 submissions to Track C (poster) and selected 6 for short oral presentations. This volume contains the papers accepted for Track A.

We express our sincere appreciation to our invited speakers, Tom de Greef, Marta Kwiatkowska, Jérôme Leroux, Ard Louis, Damien Woods, and Niles Pierce. We especially thank all of the authors who contributed papers to these proceedings, and who presented papers and posters during the conference. Last but not least, the editors thank the members of the Program Committee and the additional invited reviewers for their hard work in reviewing the papers and providing constructive comments to the authors.

September 2020

Cody Geary
Matt Patitz

26th International Conference on DNA Computing and Molecular Programming (DNA 26).
Editors: Cody Geary and Matthew J. Patitz
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# Organization

## Steering Committee

| | |
|---|---|
| Luca Cardelli | Oxford University, UK |
| Anne Condon (Chair) | University of British Columbia, Canada |
| Masami Hagiya | University of Tokyo, Japan |
| Natasha Jonoska | University of Southern Florida, USA |
| Lila Kari | University of Waterloo, Canada |
| Chengde Mao | Purdue University, USA |
| Satoshi Murata | Tohoku University, Japan |
| John H. Reif | Duke University, USA |
| Grzegorz Rozenberg | University of Leiden, The Netherlands |
| Rebecca Schulman | Johns Hopkins University, USA |
| Nadrian C. Seeman | New York University, USA |
| Friedrich Simmel | Technical University Munich, Germany |
| David Soloveichik | University of Texas at Austin, USA |
| Andrew J. Turberfield | Oxford University, UK |
| Erik Winfree | California Institute of Technology, USA |
| Damien Woods | Maynooth University, Ireland |
| Hao Yan | Arizona State University, USA |

26th International Conference on DNA Computing and Molecular Programming (DNA 26).
Editors: Cody Geary and Matthew J. Patitz

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## Program Committee

| | |
|---|---|
| Matt Patitz (Co-chair) | University of Arkansas, USA |
| Cody Geary (Co-chair) | Aarhus University, Denmark |
| Ebbe Andersen | Aarhus University, Denmark |
| Luca Cardelli | University of Oxford, UK |
| Yuan-Jyue Chen | Microsoft Research, USA |
| Anne Condon | University of British Columbia, Canada |
| David Doty | University of California, Davis, USA |
| Elisa Franco | University of California, Los Angeles, USA |
| Anthony Genot | CNRS, France |
| Manoj Gopalkrishnan | Indian Institute of Technology, Bombay, India |
| Elton Graugnard | Boise State University, USA |
| Masami Hagiya | University of Tokyo, Japan |
| Rizal Hariadi | Arizona State University, USA |
| Natasha Jonoska | University of South Florida, USA |
| Lila Kari | University of Waterloo, Canada |
| Matthew Lakin | University of New Mexico, USA |
| Chenxiang Lin | Yale University, USA |
| Yan Liu | Arizona State University, USA |
| Olgica Milenkovic | University of Illinois, USA |
| Satoshi Murata | Tohoku University, Japan |
| Pekka Orponen | Aalto University, Finland |
| Tom Ouldridge | Imperial College London, UK |
| Lulu Qian | California Institute of Technology, USA |
| John Reif | Duke University, USA |
| Dominic Scalise | Johns Hopkins University, USA |
| Nicolas Schabanel | CNRS and École normale supérieure de Lyon, France |
| Joseph Schaeffer | Autodesk Research, USA |
| Robbie Schweller | University of Texas Rio Grande Valley, USA |
| William Shih | Harvard University, USA |
| David Soloveichik | University of Texas, USA |
| Darko Stefanovic | University of New Mexico, USA |
| Jamie Stewart | California Institute of Technology, USA |
| Petr Sulc | Arizona State University, USA |
| Chris Thachuk | California Institute of Technology, USA |
| Greg Tikhomirov | California Institute of Technology, USA |
| Andrew Turberfield | Oxford University, UK |
| Bryan Wei | Tsinghua University, China |
| Shelley Wickham | University of Sydney, Australia |
| Erik Winfree | California Institute of Technology, USA |
| Damien Woods | Maynooth University, Ireland |
| Fei Zhang | Rutgers University, USA |

## Additional Reviewers

Abdulmelik Mohammed
Christian Cuba Samaniego
Daniel Fu
Daniel Hader
David Arredondo
David Haley
Eric Severson
Eugen Czeizler
Ho-Lin Chen
Joanna Ellis-Monaghan
Lance Williams
Margherita Maria Ferrari
Miklos Z. Racz
Scott Summers
Shalin Shah
Shinnosuke Seki
Tianqi Song
Wen Wang

## Organizing Committee for DNA 26

| | |
|---|---|
| Andrew Phillips (Co-chair) | Microsoft Research, Cambridge, UK |
| Andrew Turberfield (Co-chair) | University of Oxford, UK |
| Claire Garland | Institute of Physics, UK |

## Sponsors

International Society for Nanoscale Science, Computation, and Engineering
Biological Physics Group, Institute of Physics
Department of Physics, University of Oxford
Microsoft Research

# The Topology of Scaffold Routings on Non-Spherical Mesh Wireframes

**Abdulmelik Mohammed**
Department of Mathematics and Statistics, University of South Florida, Tampa, FL, USA
abdulmelik@usf.edu

**Nataša Jonoska**
Department of Mathematics and Statistics, University of South Florida, Tampa, FL, USA
jonoska@usf.edu

**Masahico Saito**
Department of Mathematics and Statistics, University of South Florida, Tampa, FL, USA
saito@usf.edu

## Abstract

The routing of a DNA-origami scaffold strand is often modelled as an Eulerian circuit of an Eulerian graph in combinatorial models of DNA origami design. The knot type of the scaffold strand dictates the feasibility of an Eulerian circuit to be used as the scaffold route in the design. Motivated by the topology of scaffold routings in 3D DNA origami, we investigate the knottedness of Eulerian circuits on surface-embedded graphs. We show that certain graph embeddings, checkerboard colorable, always admit unknotted Eulerian circuits. On the other hand, we prove that if a graph admits an embedding in a torus that is not checkerboard colorable, then it can be re-embedded so that all its non-intersecting Eulerian circuits are knotted. For surfaces of genus greater than one, we present an infinite family of checkerboard-colorable graph embeddings where there exist knotted Eulerian circuits.

**2012 ACM Subject Classification** Mathematics of computing → Discrete mathematics

**Keywords and phrases** DNA origami, Scaffold routing, Graphs, Surfaces, Knots, Eulerian circuits

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.1

**Funding** This research was (partially) supported by the grants NSF DMS-1800443/1764366 and the Southeast Center for Mathematics and Biology, an NSF-Simons Research Center for Mathematics of Complex Biological Systems, under National Science Foundation Grant No. DMS-1764406 and Simons Foundation Grant No. 594594. Travel support is provided to Abdulmelik Mohammed through an AMS-Simons Travel Grant (2020).

## 1 Introduction

The conception of stable branched DNA molecules was one of the central ideas in the birth of DNA nanotechnology [28, 29]. Branched nucleic acids exhibit a mathematical structure naturally modelled by graphs, where graph vertices (roughly points) correspond to the branch locations while graph edges (roughly line segments connecting points) model linear double-helical domains. Graph-theoretic models for the construction of three-dimensional DNA nanostructures have been proposed as early as 1997 [15, 16]. The first experiments demonstrating the self-assembly of non-regular graphs using DNA junctions as vertices and duplexes as edge connectors were presented in 2003 [27]. DNA self assembly has also been used to solve small instances of graph-theoretic problems such as the Directed Hamilton Path problem [2] and the vertex 3-colorability problem [33].

Graphs of convex polyhedra [8, 11, 13, 14, 30] have been synthesized using a variety of DNA vertex and edge motifs. Graph theory took an explicit and integral role in the automated design of non-convex polyhedra when graphs embedded in topological spheres were

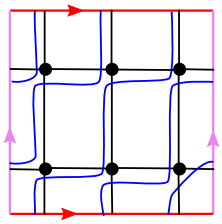



**Figure 1** A knotted Eulerian circuit (A-trail) on a torus.

exploited to model a large class of wireframe DNA origami [5, 22]. Thereafter, graph-theoretic modelling has been widely adopted for the design and synthesis of 2D [4, 18, 19] and 3D wireframe DNA origami [17, 32].

In DNA origami [26], a long, typically circular, scaffold strand is folded into a target conformation using hundreds of short helper strands. One of the key and challenging steps in designing complex 3D DNA origami is the routing of the circular scaffold strand so that it covers half the mass of each of the constituent helical domains. In graph based design of DNA origami [5, 4, 32], the scaffold routing typically corresponds to an Eulerian circuit of a graph which has been obtained from the target wireframe after some processing. Briefly, an Eulerian circuit is a closed path in a graph which traces each edge exactly once. Eulerian circuits capture the essential idea that the scaffold constitutes exactly one of the strands in each double helical domain. A general scheme for stapling Eulerian scaffold routings has been proposed in [22].

A fundamental consideration when employing circular strands in the design of nanostructures is ensuring that the topology of the strand routing in the design corresponds to the topology of the physical strand. For instance, the scaffold strand currently used in DNA origami assembly is unknotted. In most DNA origami constructs, the scaffold does not intersect itself when it traces the structure. For this reason, a class of non-intersecting Eulerian circuits called A-trails was adopted for unknotted scaffold routing of Eulerian graphs embedded in a sphere [5]. However, it has been pointed out that A-trails can be knotted for graphs embedded in tori [9]. An example of a knotted A-trail on a torus is shown in Figure 1. The A-trail is illustrated with the blue curve. As usual, the torus is obtained by gluing the horizontal boundaries in red together to form a cylinder and then gluing the violet boundaries to close the cylinder to a torus. Compare with Figure 3 to see that the A-trail corresponds to a trefoil knot. Unknotted scaffold routings may be achieved with non-intersecting Eulerian circuits (a generalization of A-trails, see definitions in Section 2) for graphs that are embedded in surfaces. In this paper, we further investigate the knottedness of non-intersecting Eulerian circuits. These Eulerian circuits can represent knotted or unknotted scaffold routings. Here we specify properties of graph embeddings in surfaces when knotted or unknotted scaffold routings arise from non-intersecting Eulerian circuits.

An approximation algorithm for finding unknotted scaffold routings on triangular embeddings in positive genus surfaces has been proposed earlier [23]. For certain Eulerian graphs, the algorithm can trace some edges twice even if the embedded graph contains an unknotted non-intersecting Eulerian circuit. It has been proved that for checkerboard-colorable graph embeddings (see definition in Section 2) in a torus, A-trails, if any exist, are unknotted [24]. In this paper, we present a number of additional results connecting checkerboard-colorable graph embeddings and the knottedness of non-intersecting Eulerian circuits. We generalize the result of [24] by proving that *all* non-intersecting Eulerian circuits of checkerboard-colorable torus graphs are unknotted. We show that *at least one* unknotted non-intersecting

(a) (b) (c)

**Figure 2** Closed orientable surfaces of genus 1, 2 and 3 in (a), (b) and (c), respectively.

Eulerian circuits exists for all checkerboard-colorable embeddings in orientable closed surfaces, including surfaces of genus greater than one. We show that, however, checkerboard-colorable graph embeddings in surfaces of genus greater than one can contain knotted Eulerian circuits. For tori, we characterize graphs which admit embeddings where *all* non-intersecting Eulerian circuits are knotted; such embeddings would require a knotted scaffold for routing as a non-intersecting Eulerian circuit.

## 2 Preliminaries

Graphs embedded in non-spherical surfaces significantly expand the class of wireframe DNA origami that can be designed based on topological techniques. For instance, reinforced cubes [32] and certain cubic lattices can be modelled as graphs on non-spherical surfaces. In this section, we present the basic topological concepts needed to introduce non-intersecting Eulerian circuits on surface-embedded graphs, our model for topological study of scaffold routings. We refer the reader to Armstrong's book [3] for an accessible account on surfaces, the monograph by Fleischner [10] for a detailed exposition on Eulerian graphs and the first two chapters of Rolfsen's classic [25] for an illustrative introduction to knot theory.

### 2.1 Surfaces

Surfaces are mathematical models of spaces which, when sufficiently zoomed in, look like a flat plane. Surfaces are commonly used in computer graphics as boundary models of well-defined 3D shapes. The simplest example of a surface is the unit sphere $S^2 = \{(x, y, z) \in \mathbb{R}^3 | x^2 + y^2 + z^2 = 1\}$. Topologically, a sphere is any space homeomorphic to the unit sphere. For instance, the underlying spaces of all the meshes constructed in [5] are topological spheres.

The simplest surface topologically distinct from a sphere is a torus. It is commonly recognized in its standard embedding like the crust of a doughnut (cf. Figure 2a). A torus can be fairly complicated as a geometric figure. The surface of a regular coffee mug is, for instance, topologically a torus. Let $S^1$ denote the unit circle in the plane. Formally, a *torus* $T$ is a surface homeomorphic to the product space $S^1 \times S^1$. Viewing $S^1$ as the unit circle in the complex plane, points in a torus can be given coordinates $(e^{i\theta}, e^{i\phi})$, for $0 \leq \theta, \phi < 2\pi$. In the standard embedding of the torus (Figure 2a), $\theta$ can be understood as the counter-clockwise rotation with respect to the axis of rotational symmetry, while $\phi$ denotes the right-handed rotation with respect to the core circle of the embedding. A torus is commonly represented by its fundamental polygon, a square whose parallel edges are identified and glued to form the torus (compare Figure 3c and 3b). On the square, $\theta$ can be understood to go from 0 to $2\pi$ along the horizontal edge in the positive $x$ direction, while $\phi$ does so along the vertical edge in the positive $y$ direction.

More complicated surfaces are constructed by joining tori together as follows. The *connected sum* of two surfaces $F_1$ and $F_2$ is obtained by removing topological open disks $D_i$ from $F_i$, for $i \in \{1, 2\}$, and gluing the resulting surfaces $F_i \setminus D_i$ along their boundaries. For instance, the connected sum of two tori is the 2-torus shown in Figure 2b; the blue

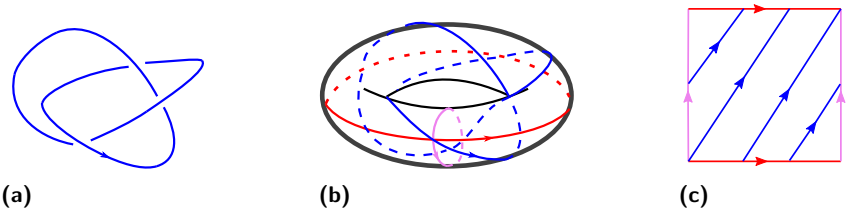



**Figure 3** A trefoil knot (a) in a torus (b) and in the fundamental square of the torus (c).

curve indicates the location where the two tori are summed. The classification theorem of (compact, connected, orientable, and without boundary) surfaces states that any surface is either a sphere, a torus, or the connected sum of $n$ tori, for $n \geq 2$. Here, $n$ denotes the *genus* of the surface. The sphere is considered to have genus 0 while the torus has genus 1. As a sample of the classification theorem, three surfaces of genus 1, 2 and 3 are shown in Figure 2a, 2b and 2c, respectively.

A *loop* in a surface $F$ is a continuous map $\beta\colon S^1 \to F$, where $S^1$ is oriented in this setting, for instance, in the counter-clockwise direction. A loop $\beta$ is *simple* if $\beta(s_1) \neq \beta(s_2)$, for all pair of distinct points $s_1, s_2$ in $S^1$. A simple loop $\beta$ is said to be *separating* if $F \setminus \mathrm{Im}(\beta)$ consists of two disjoint connected components; otherwise it is *non-separating*. The blue curve in Figure 2b is a separating loop. Two basic examples of non-separating simple loops are the longitude and meridian of the torus, drawn in red and violet in Figure 3b, respectively. The *longitude* of the torus is the loop $\beta_L\colon S^1 \to S^1 \times S^1$ with $\beta_L(e^{i\theta}) = (e^{i\theta}, 1)$, while the *meridian* is the loop $\beta_M\colon S^1 \to S^1 \times S^1$ with $\beta_L(e^{i\phi}) = (1, e^{i\phi})$.

A *knot* is an embedding of the unit circle in $\mathbb{R}^3$. A trefoil knot, which is obtained by joining the two ends of the everyday overhand knot, is illustrated in Figure 3a. Two knots are *equivalent* if there is an orientation preserving self-homeomorphism of $\mathbb{R}^3$ taking the first knot to the second. Intuitively, this represents the fact that two knots are equivalent if and only if the first knot can be continuously deformed to the second one without crossing itself during the deformation. A knot is *trivial* or an *unknot* if it is equivalent to the unit circle in the plane. Otherwise it is *non-trivial*. A knot is trivial if and only if it bounds a disk (tamely) embedded in $\mathbb{R}^3$ (see Theorem 10.6, p. 224 in [3]).

A *torus knot* is a non-trivial knot that lies in the standard torus. As the sketch in Figure 3b demonstrates, the trefoil knot is a torus knot; Figure 3c depicts the knot in the fundamental square of the torus. Loops on the torus belong to homotopy classes that can be identified by a pair of integers $(a, b)$, where $a$ denotes the number of times the loop goes around in the positive longitude direction and $b$ denotes the number of times it goes around the positive meridian direction. A class $(a, b)$ is represented by a simple loop if and only if both $a$ and $b$ are zero, or $gcd(a, b) = 1$ [25, p. 19]. A simple loop on a torus is a trivial knot if $|a| \leq 1$ or $|b| \leq 1$; otherwise, it is a non-trivial knot. Thus, torus knots can be identified with a pair of coprime integers $(a, b)$ with absolute values greater than one. The trefoil knot shown in Figure 3a is a torus knot of type $(2, 3)$.

A *longitudinal (Dehn) twist* of a torus is a self-homeomorphism $h_L\colon T \to T$ with $h((e^{i\theta}, e^{i\phi})) = (e^{i(\theta+\phi)}, e^{i\phi})$. A *meridional (Dehn) twist* is a self-homeomorphism $h_M\colon T \to T$ with $h((e^{i\theta}, e^{i\phi})) = (e^{i\phi}, e^{i(\phi+\theta)})$. It is to be understood that $h_L$ and $h_M$ constitute positive twists while their inverses form negative twists. Intuitively, a longitudinal (resp. meridional) twist is obtained by cutting the torus along the longitude (resp. meridian), twisting the resulting cylinder by 360° and gluing the cylinder ends together to form a torus. On the fundamental square of the torus, a longitudinal twist can be visualized as a horizontal shear,

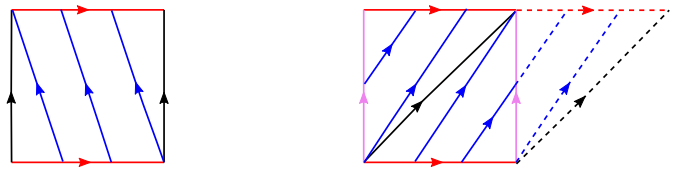



**Figure 4** A longitudinal twist of a torus sending a $(-1, 3)$ loop in a torus (a) to the $(2, 3)$ torus knot (b).

as illustrated in Figure 4; the upper triangle protruding from the square is to be understood as coming back on the left to join with the lower triangle. A meridional twist can analogously be visualized as a vertical shear of the square. A positive longitudinal twist maps a knot of class $(a, b)$ to a knot of class $(a + b, b)$ while a positive meridional twist maps a knot of class $(a, b)$ to a knot of class $(a, a + b)$ [25, p. 24]. Negative twists map from class $(a, b)$ to classes $(a - b, b)$ and $(a, -a + b)$, respectively. A positive longitudinal twist taking a $(-3, 1)$ unknot to the $(2, 3)$ trefoil knot is shown in Figure 4; Figure 4a shows the unknot, while the trefoil knot that is produced by the twist is shown in Figure 4b.

## 2.2 Graphs

Graphs are natural models to represent the branching of nucleic acids and have been successfully used to design DNA origami polyhedral wireframes [5, 32]. While a surface models the set of all points in the boundary of a polyhedron, the wireframe composed of the corners and edges of a polyhedron constitute the graph that is embedded in the surface. Here, we briefly recall some basic notions related to graphs. We refer the reader to [12] for a thorough but accessible introduction to graphs on surfaces.

All graphs under consideration in this paper are finite and undirected but, for brevity of construction, can contain multiedges and loops. It is assumed that all graphs contain at least one edge. Each edge in a graph is understood to be composed of two half edges which are incident to the two endpoints of the edge; in the case of a loop edge, the two half edges meet the same vertex. The degree of a vertex $v$ is the number of half edges incident to it and is denoted by $d(v)$. A vertex is said to be *even* if it has an even degree.

For graphs that are embedded in surfaces, it is convenient to think of graphs as topological spaces which are endowed a 1-dimensional cell structure, where the 0-cells correspond to vertices and the 1-cells correspond to edges. An *embedding* $g\colon G \to F$ of a graph $G$ in a surface $F$ is a topological embedding of $G$ into $F$; that is, the image $g(G)$ is homeomorphic to the topological space $G$. In other terms, an embedding of a graph is a drawing of the graph on the surface where no edges cross. The space $F \setminus g(G)$ consists of disjoint connected subspaces called *faces*. An embedding of a graph in a surface is said to be *checkerboard colorable* if the faces can be assigned two colors (e.g. black and white) such that, for every edge, the two faces on the two sides of the edge are assigned distinct colors; if there is an edge where one face is present on both sides of the edge, the embedding is not checkerboard colorable. See Figure 8a for a checkerboard-colorable embedding of $K_7$, the complete graph on seven vertices.

An embedding $g\colon G \to F$ is said to be *cellular* if each face is homeomorphic to the open unit disk. A cellular embedding of a simple graph is said to be *triangular* if each face is bounded by three distinct edges. An embedding $g\colon G \to F$ determines a counter-clockwise cyclic order $\rho_v$ of the half edges incident at a vertex $v$, for each vertex $v$ in the graph. The

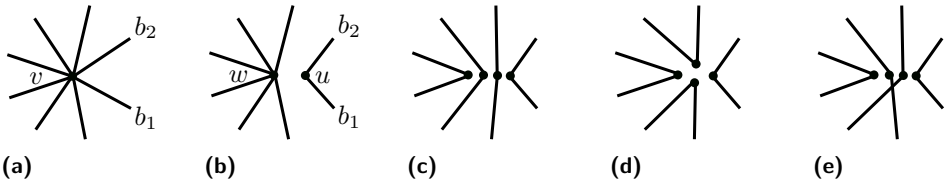



**Figure 5** Smoothing of an even vertex. (a) Neighboring half edges in a vertex, (b) smoothing one transition composed of the neighboring half edges, (c) a smoothing of the vertex induced by a non-intersecting Eulerian circuit, (d) a smoothing induced by an A-trail, (e) a splitting away of transitions where two transitions intersect.

order $\rho_v$ is called a *rotation* at $v$. The collection $\rho = \{\rho_v : v \in G\}$ of rotations at vertices is called a *rotation system*. In a rotation system, each vertex can be treated as rigid (see [7] for the notion of rigid vertices). Conversely, if each vertex is rigid, it gives rise to a cellular embedding $g : G \to F$ for some (closed orientable) surface $F$.

In wireframe DNA origami [5, 32], the fact that the scaffold comprises one strand of each double-helical domain is conveniently captured by an Eulerian circuit of an underlying graph. A *circuit* in a graph is a closed walk $(v_0, e_0, v_1, \ldots, v_{l-1}, e_{l-1}, v_0)$ with no repeated edges, where $l \geq 1$ is the *length* of the circuit and each $e_i$, for $0 \leq i \leq l-1$, is an edge in the graph with endpoints $v_i$, $v_{i+1 \bmod l}$. An *Eulerian circuit* is a circuit which visits every edge of the graph. A graph is said to be *Eulerian* if it contains an Eulerian circuit. A connected graph is Eulerian if and only if every vertex is of even degree. Closely related to circuits are cycles and transitions. A *cycle* is a circuit with no repeated vertices. For a surface-embedded graph, a cycle corresponds to a simple loop and the separating/non-separating qualification equally apply to cycles. A *transition* is an unordered pair of half edges incident to a common vertex. A circuit $C = (v_0, e_0, v_1, \ldots, v_{l-1}, e_{l-1}, v_0)$ can also be seen as a collection of transitions $\{b_i, b'_{i+1 \bmod l}\}$, where $b_i$ is the half edge of $e_i$ incident to $v_{i+1 \bmod l}$, and $b'_i$ is the half edge of $e_i$ incident with $v_i$, for all $i \in \{0, \ldots, l-l\}$. In this sense, we can say that $\{b_i, b'_{i+1 \bmod l}\}$ is contained in $C$.

Let $g : G \to F$ be an embedding of a graph in a surface. Let $v$ be a vertex of $G$ with $d(v) \geq 4$ and let the rotation $\rho_v$ determined by $g$ be $(b_0, \ldots, b_{d(v)-1})$. Let $0 \leq i, j, k, l \leq d(v) - 1$ with $i < j, k < l, i < k$. A pair of disjoint transitions $\{b_i, b_j\}$, $\{b_k, b_l\}$ *intersect* if $i < k < j < l$ (cf. Figure 5e). An Eulerian circuit of an Eulerian graph $G$ is said to be *non-intersecting* with respect to $g : G \to F$ if it contains no intersecting transitions with respect to $g$ (cf. the collection of transitions of the vertex $v$ in Figure 5a suggested by Figure 5c). It has been shown that a non-intersecting Eulerian circuit can be found in polynomial time for any Eulerian graph embedded in a sphere [1, 31], or in any other surface [10, 23]. However, the computational complexity changes when considering a subclass of non-intersecting Eulerian circuits called A-trails. Two half edges $b_1$, $b_2$ incident to a vertex $v$ are said to be *neighbors* if $\rho_v(b_1) = b_2$ or $\rho_v(b_2) = b_1$ (see Figure 5a for an example). An *A-trail* is a non-intersecting Eulerian circuit where every transition is composed of neighboring half edges (cf. Figure 5d). Deciding whether a surface-embedded graph has an A-trail is known to be NP-COMPLETE, even when restricted to embeddings in a sphere [6].

Let $g : G \to F$ be a graph embedded in a surface. Let $v$ be a vertex of $G$, $d(v) \geq 4$, with rotation $\rho_v$ determined by $g$. Let $t = \{b_1, b_2\}$ be a transition composed of neighboring half edges incident to $v$. A *smoothing of a transition* $t$ is the graph embedded in $F$ obtained from $(G, g)$ by deleting $v$ and adding two new vertices $u$ and $w$ such that $b_1$ and $b_2$ become incident with $u$ and the rest of the half edges become incident with $w$. The graph obtained

after smoothing is embedded exactly according to $g$ except in a local disk neighborhood of $v$ where $u$ and $w$ are embedded in a manner illustrated by the example in Figure 5b. Note that the two half edges flanking $b_1$ and $b_2$ become neighbors in the new embedding. The notion of smoothing defined here is a special case of the notion of "splitting away a pair of edges" [10, p. III.16] catered to non-intersecting Eulerian circuits. Now suppose $v$ is even and its incident half edges are partitioned into disjoint mutually non-intersecting transitions. The transitions can be ordered as $\sigma = (t_1, \ldots, t_{d(v)/2})$ such that $t_1$ is composed of neighboring half edges, and each $t_{i+1}$ is composed of neighboring half edges after $t_i$ has been smoothed. A *smoothing of* $v$ is the embedded graph $\tilde{g}_v \colon \tilde{G}_v \to F$ obtained after such a sequence $\sigma$ of smoothings of transitions. Two possible smoothings of the vertex $v$ in Figure 5a are shown in Figures 5c and 5d. We note that smoothings of a vertex are in bijection with crossingless chord diagrams. The number of possible smoothings of a vertex $v$ is the Catalan number $C_k = \frac{1}{k+1}\binom{2k}{k}$, where $k = \frac{d(v)}{2}$. A *smoothing of a non-intersecting Eulerian circuit* $\gamma$ is the embedded cycle graph $\tilde{\gamma}$ obtained after smoothing all the vertices according to the transitions in $\gamma$. The smoothed Eulerian circuit $\tilde{\gamma}$ is unique up to isotopy. Figures 5c and 5d illustrate smoothings of a vertex based on a non-intersecting Eulerian circuit and an A-trail, respectively.

Having established the concepts, the general scheme of discussion is as follows: we are given an Eulerian graph $G$ embedded in a surface $F$ and a non-intersecting Eulerian circuit $\gamma$; then $F$ is embedded in $\mathbb{R}^3$. In notation, this is described as: $\gamma \to G \overset{g}{\hookrightarrow} F \overset{f}{\hookrightarrow} \mathbb{R}^3$.

We then ask whether $f(\tilde{\gamma})$ is an unknot or a non-trivial knot. We present results where $f(\tilde{\gamma})$ is an unknot in Section 3 and results where $f(\tilde{\gamma})$ is a non-trivial knot in Section 4. When $f(\tilde{\gamma})$ is an unknot, the regular unknotted scaffold can be routed according to $\gamma$; otherwise either a knotted scaffold must be used, or a different unknotted non-intersecting Eulerian circuit must be chosen. If all $f(\tilde{\gamma})$ are non-trivial knots, a knotted scaffold is necessary for routing the embedded graph using a non-intersecting Eulerian circuit.

## 3 Unknotted Scaffold Routings

When the available scaffold is unknotted, as typically is the case, we aim to find unknotted non-intersecting Eulerian circuits. In this section, we show that checkerboard colorability of an embedding is a sufficient condition for an embedded graph to contain an unknotted non-intersecting Eulerian circuit, thus allowing design using the typical unknotted scaffold strand.

It is well-known that a graph embedded in a sphere is Eulerian if and only if the embedding is checkerboard colorable [10, Theorem III.68]. Although an Eulerian graph embedded in a positive genus surface may not be checkerboard colorable, we show that checkerboard colorability affects the topology of Eulerian circuits on surface-embedded graphs. It has been shown that [24, Theorem 3.6] all A-trails (if any exist) on checkerboard-colorable torus graphs are unknotted, for any embedding $f \colon T \to \mathbb{R}^3$. We first generalize this result to all non-intersecting Eulerian circuits using a more topological proof. We then show a general result for all surfaces: every checkerboard-colorable surface-embedded graph admits an unknotted non-intersecting Eulerian circuit.

Non-intersecting Eulerian circuits are unknotted on a sphere due to the Jordan-Schönflies theorem [25, p. 9], which states that every simple loop in a sphere is separating and bounds a disk. On the other hand, simple loops in a torus can either be separating or non-separating. A separating loop in a torus bounds a disk on one side and thus one strategy to find an unknotted non-intersecting Eulerian circuit on a torus graph is to search for a separating non-intersecting Eulerian circuit. In Lemma 2, we show that the checkerboard colorability


**Figure 6** A checkerboard coloring viewed locally at a vertex (a) and how it induces a checkerboard coloring when the vertex is smoothed (b).

of a graph embedding is a sufficient criteria for its non-intersecting Eulerian circuits to be separating. To prove Lemma 2, we first prove in Lemma 1 that checkerboard colorability is preserved under smoothing and unsmoothing of vertices.

▶ **Lemma 1.** *Let $g\colon G \to F$ be an embedding of an Eulerian graph $G$ in a surface $F$ and let $\tilde{g}_v\colon \tilde{G}_v \to F$ be an embedding obtained by smoothing a vertex $v$ of $G$. Then, $g$ is checkerboard colorable if and only if $\tilde{g}_v$ is checkerboard colorable.*

**Proof.** The proof idea is sufficiently illustrated by the example in Figure 6, where a checkerboard coloring of $g$ (Figure 6a) is extended to a checkerboard coloring of $\tilde{g}_v$ (Figure 6b). In words, since any smoothing of $v$ is by definition obtained as a sequence of smoothings of transitions (composed of neighboring edges), it is sufficient to prove the claim for a smoothing of a transition. In a checkerboard coloring, if the faces that merge when smoothing a transition are distinct, they are colored alike before they merge. In this manner, a checkerboard coloring of $g$ extends to a checkerboard coloring of $\tilde{g}_v$ when the faces are merged. When unsmoothing a transition, if a face is split into two faces, the new faces inherit the color of the parent for a checkerboard coloring of the new embedding. In this way, a checkerboard coloring of $\tilde{g}_v$ naturally induces a checkerboard coloring of $g$. ◀

We can now prove Lemma 2 that relates checkerboard colorability of graph embeddings and the separating property of non-intersecting Eulerian circuits.

▶ **Lemma 2.** *Let $g\colon G \to F$ be an embedding of an Eulerian graph $G$ in a surface $F$. The following claims hold for every smoothed non-intersecting Eulerian circuit $\tilde{\gamma}$ of $(G, g)$:*

**(i)** *If $g$ is checkerboard colorable, then $\tilde{\gamma}$ is separating;*

**(ii)** *If $g$ is not checkerboard colorable, then $\tilde{\gamma}$ is non-separating.*

**Proof.** **(i)** Let $\gamma$ be an arbitrary non-intersecting Eulerian circuit of $(G, g)$. If $g$ is checkerboard colorable, then $\tilde{\gamma}$ is checkerboard colorable by Lemma 1. In a checkerboard coloring of $\tilde{\gamma}$ the two sides of $\tilde{\gamma}$ must be colored differently; thus the two sides must be in distinct faces and $\tilde{\gamma}$ must be separating.

**(ii)** By the contrapositive, suppose there exists a separating smoothed non-intersecting Eulerian circuit $\tilde{\gamma}$. Since $\tilde{\gamma}$ is separating, the two separate regions can be colored distinctly to obtain a checkerboard coloring of $\tilde{\gamma}$. By Lemma 1, unsmoothing $\tilde{\gamma}$ to $g$ gives rise to a checkerboard coloring of $g$. ◀

Lemma 2 equips us to generalize Theorem 3.6 of [24] to non-intersecting Eulerian circuits on checkerboard-colorable torus graphs, as stated in Theorem 3.

▶ **Theorem 3.** *If $g\colon G \to T$ is a checkerboard-colorable cellular embedding of an Eulerian graph in a torus, then $f(\tilde{\gamma})$ is an unknot for any non-intersecting Eulerian circuit $\gamma$ of $(G, g)$ and any embedding $f\colon T \to \mathbb{R}^3$.*

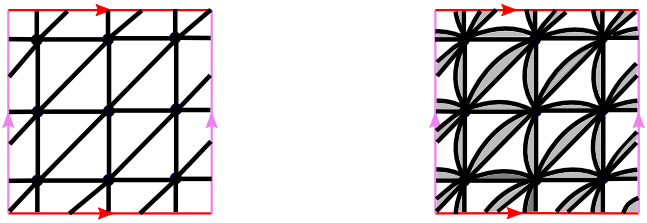



**Figure 7** A checkerboard-colorable graph embedding (b) obtained by doubling the edges of a graph which has a triangular embedding in a torus (a).

**Proof.** By Lemma 2, any smoothed non-intersected Eulerian circuit $\tilde{\gamma}$ of $(G, g)$ is separating. A separating loop in a torus bounds a disk and thus $\tilde{\gamma}$ bounds a disk. Under any homeomorphism of $T$, $\tilde{\gamma}$ still bounds a disk and thus $f(\tilde{\gamma})$ is an unknot for any embedding $f\colon T \to \mathbb{R}^3$ of the torus in $\mathbb{R}^3$. ◀

For checkerboard-colorable embeddings on a torus, by Theorem 3, *any* non-intersecting Eulerian circuit can be used as a route for an unknotted scaffold strand. Theorem 3 suggests the existence of graphs where the unknottedness of non-intersecting Eulerian circuits can be guaranteed purely from the adjacency structure of the abstract graph, i.e., independent of the graph embedding in the torus and of the torus' embedding in $\mathbb{R}^3$. An infinite family of graphs with this property is presented in Proposition 4. For such families of graphs, the possibility of routing using unknotted scaffold strand is completely determined from the abstract graph.

▶ **Proposition 4.** *There exist an infinite family $\mathcal{G}$ of Eulerian graphs such that for all $G \in \mathcal{G}$, and all $g\colon G \to T$, and all $f\colon T \to \mathbb{R}^3$, and all non-intersecting Eulerian circuit $\gamma$ of $(G, g)$, $f(\tilde{\gamma})$ is an unknot.*

**Proof.** Let $\mathcal{G}$ be the family of graphs obtained by doubling the edges of graphs with triangular embedding in a torus. Let $G$ be a graph in $\mathcal{G}$. One example is shown in Figure 7b. Consider any pair $e_1, e_2$ of double edges with endpoints $u$ and $v$. With slight abuse of notation, let $\rho_u$ (resp. $\rho_v$) denote the cyclic counter-clockwise order of the edges, instead of half edges, incident with $u$ (resp. $v$). In any embedding $g$ of $G$ in a torus, either $\rho_u(e_1) = e_2$ or $\rho_u(e_2) = e_1$. If $\rho_u(e_1) = e_2$ then $\rho_v(e_2) = e_1$; otherwise $\rho_v(e_1) = e_2$. Thus, double edges such as $e_1, e_2$ bound faces in $g$. These faces can be shaded black, while the other faces are left white, to get a checkerboard coloring of $g$ (cf. Figure 7b). The claim then follows from Theorem 3. ◀

Theorem 3 crucially depends on the surface being a torus, as a separating loop in a surface of genus greater than one need not bound a disk. For instance, the blue loop in the double torus in Figure 2b is separating but bounds punctured tori on both sides. In Section 4 (Theorem 8), we employ this property to construct families of checkerboard-colorable embeddings in $F_n$ ($n \geq 2$) with knotted non-intersecting Eulerian circuits. Although checkerboard colorability is not sufficient to guarantee that *all* non-intersecting Eulerian circuits are unknotted for embeddings in surfaces of genus at least two, it is sufficient to ensure that there is *at least one* unknotted non-intersecting Eulerian circuit, as shown in Theorem 5. Thus, checkerboard-colorable graph embeddings can generally be routed using an unknotted scaffold.

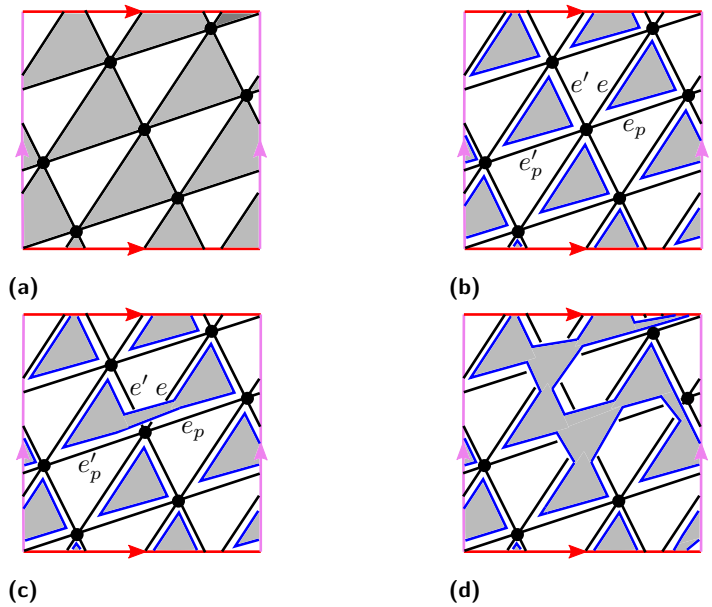



**Figure 8** An unknotted non-intersecting Eulerian circuit of $K_7$ in a torus. (a) a checkerboard-colorable embedding of $K_7$ in a torus, (b) circuits bounding the black faces, (c) merging circuits, (d) the unknotted non-intersecting Eulerian circuit.

▶ **Theorem 5.** *If $g\colon G \to F$ is a checkerboard-colorable cellular embedding of an Eulerian graph $G$ in a surface $F$, then there exists a non-intersecting Eulerian circuit $\gamma$ of $G$ such that $f(\tilde{\gamma})$ is unknotted for any embedding $f\colon F \to \mathbb{R}^3$.*

**Proof.** Let $g\colon G \to F$ be a checkerboard-colorable embedding of an Eulerian graph in a surface $F$. An example is given by the embedding of $K_7$ in the torus shown in Figure 8a. Let the faces of $g$ be colored with black and white. By the definition of checkerboard coloring, each edge is incident to exactly one black face and one white face. Thus, the collection of all the boundary circuits of the black faces form a non-intersecting circuit partition of $G$. Because the embedding is cellular, the circuits bound disjoint closed disks after a small isotopy. This is illustrated in Figure 8b for the embedding of $K_7$ in a torus.

To convert the non-intersecting circuit partition into a non-intersecting Eulerian circuit $\gamma$, we perform a re-splicing of disjoint circuits one by one at each vertex (see Lemma 7 of [23] for details). We go through the edges incident to the vertex in the cyclic order they appear in the embedding, and if two neighboring edges $e$ and $e'$ are not in the same circuit, we re-splice the two circuits so that $e$ and $e'$ are paired to each other and $e$'s previous pair $e_p$ is paired with $e'$'s previous pair $e'_p$ (cf. Figure 8c). This re-pairing merges the two circuits and reduces the number of circuits in the circuit partition, while keeping the circuit partition non-intersecting. A repeated application of this operation for every vertex in the graph yields a non-intersecting Eulerian circuit $\gamma$.

Now consider any embedding $f\colon F \to \mathbb{R}^3$. To prove $f(\tilde{\gamma})$ is an unknot, we show by induction that $\tilde{\gamma}$ bounds a disk. In particular, we prove that, after each merge of circuits through a re-pairing of edges, each circuit in the circuit partition, up to isotopy, bounds a closed disk. The base case is handled by the circuit partition formed from the black faces. Suppose by induction hypothesis that all the circuits before the pairing of $e$ and $e'$ bound a disk. The re-pairing joins the two disjoint disks by a band, which results in a new disk that the new circuit bounds (cf. Figure 8c). For the embedding of $K_7$ in a torus, the non-intersecting Eulerian circuit, and the disk that it bounds can be seen in Figure 8d. ◀

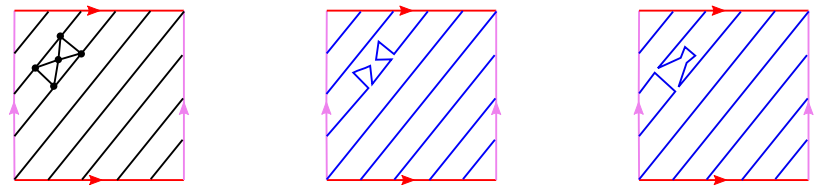



**Figure 9** Knotted Eulerian circuits on an embedding of $K_5$ in the torus. (a) an embedding of $K_5$ in a torus, (b) a non-intersecting Eulerian circuit which is a $(4, 5)$ torus knot, (c) a non-intersecting Eulerian circuit which is a $(2, 3)$ torus knot.

## 4 Knotted Scaffold Routings

In Section 3, we saw that checkerboard-colorable embeddings are closely related to the existence of unknotted scaffold routings. In this section, we study the relationship between non-checkerboard colorable embeddings and the existence of knotted scaffold routings.

A non-intersecting Eulerian circuit $\gamma$ on a surface-embedded graph can be knotted due to the embedding $g$ of the graph in the surface or due to the embedding $f$ of the surface in $\mathbb{R}^3$. Moreover, $f(\tilde{\gamma})$ can be either an unknot or a non-trivial knot for a fixed embedding $g$, depending on $f$. For instance, consider the Eulerian graph $B$ formed by the crossing of the meridian and longitude of the torus. That is, $B$ is the bouquet of two circles with one vertex and two loop edges and its embedding $g$ is the natural one where the vertex is placed at the crossing point of the meridian and longitude (recall Figure 3b). Note that $B$ has two Eulerian circuits which have identical structure; let $\gamma$ be one of these circuits. In a standard embedding of the torus (Figure 3b), $f(\tilde{\gamma})$ is an unknot. However, if the torus is embedded in $\mathbb{R}^3$ as a tubular neighborhood of a non-trivial knot $K$ such that the longitude is equivalent to $K$, then $f(\tilde{\gamma})$ is also equivalent to $K$ and thus non-trivial. The construction generalizes to graph embeddings that are not checkerboard colorable, in the sense described in Theorem 6.

▶ **Theorem 6.** *Suppose $g : G \to F$ is an embedding of an Eulerian graph $G$ in a surface $F$ and suppose that $g$ is not checkerboard colorable. Then, for any non-intersecting Eulerian circuit $\gamma$, there exists an embedding $f : F \to \mathbb{R}^3$ such that $f(\tilde{\gamma})$ is a non-trivial knot.*

**Proof.** Let $g : G \to F$ be an embedding that is not checkerboard colorable, and $\gamma$ be a non-intersecting Eulerian circuit. By Lemma 2, $\tilde{\gamma}$ is a non-separating loop in $F$. Hence, after applying a homeomorphism of $F$, $\tilde{\gamma}$ can be considered to be positioned as a *longitudinal* loop in $F$ (a curve that goes around a hole, just like a longitude of a torus). Then we can choose an embedding $f : F \to \mathbb{R}^3$ such that this longitudinal loop $\tilde{\gamma}$ is knotted.

The observation above, taking $\tilde{\gamma}$ as longitudinal as a consequence of $g$ being not checkerboard colorable, can be deduced using the first homology groups in homology theory; here the technical details are omitted. ◀

We now focus on the case where the embedding of the surface is standard. It has been shown that the bouquet of two circles can be embedded in a standard torus so that all the non-intersecting Eulerian circuits are knotted [24, Figure 11]. Figure 9a shows an embedding of the toroidal graph $K_5$ where all its non-intersecting Eulerian circuits are knotted. A non-intersecting Eulerian circuit of this embedding of $K_5$ is either a $(4, 5)$ torus knot (e.g. Figure 9b) or a $(2, 3)$ torus knot (e.g. Figure 9c). Theorem 7 characterizes Eulerian graphs which admit toroidal embeddings where all the non-intersecting Eulerian circuits are knotted.

Theorem 7 shows the existence of embeddings of Eulerian graphs where a routing as a non-intersecting Eulerian circuit would necessitate the use of knotted scaffold strands. It also supports the suggestion in [9] that knotted scaffolds could expand the possible set of DNA origami meshes that can be constructed.

▶ **Theorem 7.** *An Eulerian graph admits a cellular embedding in a standardly embedded torus where all smoothed non-intersecting Eulerian circuits are knotted if and only if it admits a cellular embedding in a torus that is not checkerboard colorable.*

**Proof.** ( $\implies$ ) By the contrapositive, if all the embeddings of a graph in a torus are checkerboard colorable, then by Theorem 3, each of these embeddings will contain an unknotted non-intersecting Eulerian circuit.

( $\impliedby$ ) Let $g\colon G \to T$ be a cellular embedding of an Eulerian graph in a torus such that the embedding is not checkerboard colorable. The main idea of the proof is to use self-homeomorphisms of the torus to twist $g$ so that each of the non-intersecting circuits becomes a non-trivial knot when the torus is embedded in a standard fashion in $\mathbb{R}^3$. This is possible because the number of non-intersecting Eulerian circuits is finite and each smoothed non-intersecting Eulerian circuit is non-separating (Lemma 2). A concrete combination of twists is presented next.

Since every (smoothed) non-intersecting Eulerian circuit of $(G, g)$ is non-separating, each oriented non-intersecting Eulerian circuit can be represented by a pair $(a, b)$ of integers with $(a, b) \neq (0, 0)$ and $gcd(a, b) = 1$. Let the $i$th oriented non-intersecting Eulerian circuit (in some order) be represented with $(a_i, b_i)$. Let $k, l, m$ be natural numbers representing the twists that are to be determined. Applying $k$ longitudinal twists to $T$ converts the embedding $g$ to an embedding $g_1$ so that the Eulerian circuits become simple loops of type $(a_i + kb_i, b_i)$. Next, applying $l$ meridional twists converts $g_1$ to an embedding $g_2$ so that the circuits become simple loops of type $(a_i + kb_i, la_i + (lk + 1)b_i)$. Finally, applying $m$ longitudinal twists converts $g_2$ to an embedding $g_3$ so that the circuits are simple loops of type $((1 + lm)a_i + (k + mlk + m)b_i), la_i + (k + 1)b_i))$. We thus only need to choose $k, l, m$ so that $|(1 + lm)a_i + (k + mlk + m)b_i)| > 1$ and $|la_i + (k + 1)b_i)| > 1$, for all $i$; that is, $k, l, m$ are to be chosen so that all the circuits become non-trivial knots. For this purpose, we can choose $l = 2, m = 1$ and $k = \max_{i:b_i \neq 0}\{\frac{2|a_i|}{|b_i|} + 1\}$ if there exists a $b_i \neq 0$, or $k = 1$ if $b_i = 0$ for all $i$. Since $(a_i, b_i) \neq (0, 0)$, we need to consider three cases:

- **(i)** $a_i = 0$ and $b_i \neq 0$. Then, $|(1 + lm)a_i + (k + mlk + m)b_i)| = |(3k + 1)b_i| = (3k + 1)|b_i| \geq (6\frac{|a_i|}{|b_i|} + 4)|b_i| = 4|b_i| \geq 4$. Additionally, $|la_i + (k + 1)b_i)| = |(k + 1)b_i| = (k + 1)|b_i| \geq (\frac{2|a_i|}{|b_i|} + 2)|b_i| \geq 2$.
- **(ii)** $a_i \neq 0$ and $b_i = 0$. Then $|(1 + lm)a_i + (k + mlk + m)b_i)| = 3|a_i| \geq 3$. Moreover, $|la_i + (k + 1)b_i)| = 2|a_i| \geq 2$.
- **(iii)** $a_i \neq 0$ and $b_i \neq 0$. Then $|(1 + lm)a_i + (k + mlk + m)b_i)| = |3a_i + (3k + 1)b_i| \geq |(3k + 1)b_i| - |3a_i| = (3k + 1)|b_i| - |3a_i| \geq (6\frac{|a_i|}{|b_i|} + 4)|b_i| - 3|a_i| = 3|a_i| + 4|b_i| \geq 7$. And $|la_i + (k + 1)b_i)| = |2a_i + (k + 1)b_i| \geq |(k + 1)b_i| - |2a_i| = (k + 1)|b_i| - |2a_i| \geq (\frac{2|a_i|}{|b_i|} + 2)|b_i| - 2|a_i| = 2|b_i| \geq 2$. ◀

Note that the twists in the proof of Theorem 7 need not change the rotation system determined by the embedding. This highlights the geometric nature of the problem, in the sense that the existence of knotted non-intersecting Eulerian circuits cannot generally be completely determined from the combinatorial structure of the embedding. In fact, the original embedding $g$ may have no knotted non-intersecting Eulerian circuits at all, as is the case for instance, with the standard embedding of the bouquet of two circles in the standard torus. Nevertheless, Theorem 7 provides a mechanism to check whether an

Eulerian graph admits an embedding in a torus where all the non-intersecting Eulerian circuits are knotted, as one can algorithmically determine whether a graph admits a cellular embedding in a torus that is not checkerboard colorable. Indeed, this can be done by going through the finite number of possible rotation systems of the graph, obtaining the cellular embeddings corresponding to the rotation systems via standard face-tracing algorithms in topological graph theory [12, p. 115], checking that the embedding is in a torus from the generalized Euler's polyhedron formula [12, p. 27, p. 122], and then checking for checkerboard colorability. Determining checkerboard colorability of a cellular embedding is equivalent to deciding whether the geometric dual is bipartite, which can be done through a standard breadth-first-search algorithm.

For surfaces of genus greater than one, even checkerboard-colorable embeddings can have knotted non-intersecting Eulerian circuits, as demonstrated by the infinite family in Theorem 8. Note that in Theorem 8, the claim is not that *all* non-intersecting Eulerian circuits are knotted but that there is *at least one* that is knotted. The problem of characterizing graphs which admit cellular embeddings in a standardly embedded surface $F_n$, $n \geq 2$, so that all non-intersecting Eulerian circuits are knotted is left for future work. Theorem 8 suggests that, unlike the case of the torus, checkerboard-colorable embeddings of graphs in surfaces of genus larger than one can possibly be routed and constructed using knotted scaffold strands.

▶ **Theorem 8.** *Let $F_n$ be an orientable closed surface of genus $n$ that is standardly embedded in $\mathbb{R}^3$.*

- **(i)** *For all $n \geq 2$, there exist infinitely many Eulerian graphs that have checkerboard-colorable cellular embeddings in $F_n$ with knotted non-intersecting Eulerian circuits.*
- **(ii)** *For any non-trivial knot $K$, there exists an Eulerian graph $G$ cellularly embedded with a checkerboard coloring in $F_n$ for some $n \geq 1$ having $K$ as a non-intersecting Eulerian circuit of $G$.*

**Proof.** First consider the case $n = 2$ for (i). Let $S$ be an orientable surface with a connected boundary obtained from a disk by attaching two twisted unknotted bands. An example is depicted in Figure 10. The twists must be full (versus half) twists to obtain an orientable surface. The boundary $\partial S$ of $S$ is a non-trivial knot $K$.

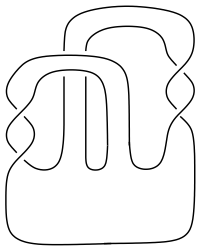


**Figure 10** Two twisted bands attached to a disk.

Let $F = F_n$ $(n = 2)$ be the surface obtained by thickening $S$. Figure 11 depicts this process. In Figure 11a a portion of a band is depicted. The top image of Figure 11a is a cross sectional view of a part of a band depicted at the bottom. In Figure 11b a thickened band is depicted with its cross section shown at the top. The boundary after thickening is a tube. By applying this process to $S$, we obtain a standard surface $F$ as depicted in Figure 12. The knot $K$ can be regarded as staying on $F$ as in Figure 12, indicated by a red curve. Note that $K$ divides $F$ into two parts (in Figure 11b the two parts are the front and back faces).

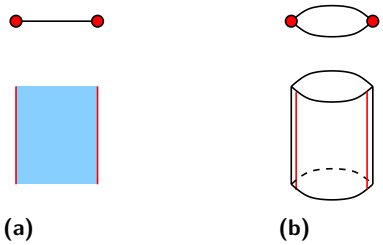



**Figure 11** Thickening a band (a) to a tube (b).

Next we construct a graph $G$ cellularly embedded in $F$ by *finger moves* as depicted in Figure 13. In Figure 13a, a dotted arc connects two parts of $K$. Push one end of $K$ along the arc, and at the other end make it intersect in two double points as indicated in Figure 13b. After one finger move we obtain a 4-regular graph with two vertices. In Figure 14, it is shown that a finger move preserves the checkerboard colorability as in Figure 14b, and there is a choice of a non-intersecting Eulerian circuit that is the original knot $K$ as illustrated in Figure 14c by a blue curve. By repeating finger moves across non-cellular faces, we obtain a cellularly embedded graph $G$ with $K$ as a non-intersecting Eulerian circuit.

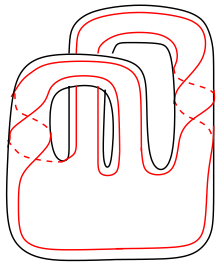


**Figure 12** The boundary surface after thickening contains the original knot.

This construction can be performed for any even $n \in \mathbb{N}$. For an odd $n$, we add a trivial handle to $F_{n-1}$ as indicated in Figure 15a. At this point $G$ becomes non-cellular. To obtain a new cellularly embedded graph, we perform two finger moves as depicted in Figure 15b. The new graph retains the checkerboard colorability and the property of having $K$ as a non-intersecting Eulerian circuit, as desired. The construction allows for infinitely many such graphs, for example by performing additional finger moves, or by choosing different arcs for finger moves. This completes the proof of (i).


**Figure 13** A finger move (b) along a dotted arc (a).

(ii) It is known that any knot $K$ can be realized as the boundary of an orientable surface $S$, such that a thickened $S$ is a standard handlebody. Hence a similar argument applies. ◀

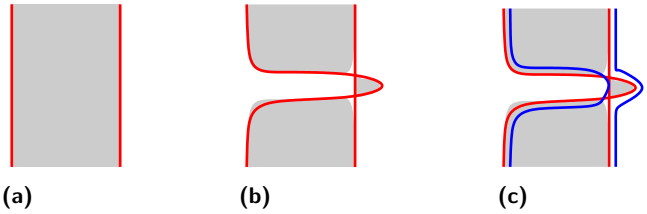



**Figure 14** A checkerboard coloring before (a) and after a finger move (b). A choice of non-intersecting Eulerian circuit after a finger move (c).

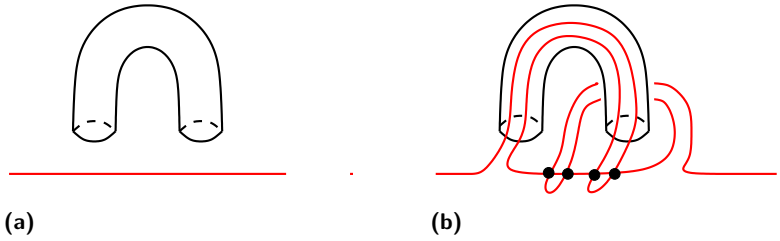



**Figure 15** A handle added to make the genus odd (a) and finger moves to make the embedding cellular (b).

## 5 Conclusion

Eulerian circuits are emerging as broadly applicable model of strand routings in biomolecular technology [4, 5, 20, 21, 32]. For circular strands, the knot type of the strand routing in the design must conform to the knot type of the strand in solution. Herein, we studied the knottedness of strand routings modelled by non-intersecting Eulerian circuits of Eulerian graphs embedded in surfaces.

We showed a strong connection between checkerboard-colorable graph embeddings in surfaces and the knottedness of non-intersecting Eulerian circuits. We extended the result of [24] by showing that all non-intersecting Eulerian circuits are unknotted for checkerboard-colorable torus graphs (Theorem 3). Thus, checkerboard-colorable torus graphs can be routed (as non-intersecting Eulerian circuits) using unknotted scaffolds but they cannot be routed using knotted ones. For checkerboard-colorable embeddings in surfaces of genus greater than one, we showed that there is at least one unknotted non-intersecting Eulerian circuit (Theorem 5). Thus, all checkerboard-colorable graph embeddings can be routed using unknotted scaffold strands. We proved that checkerboard-colorable embedded graphs in surfaces of genus greater than one can have knotted Eulerian circuits (Theorem 8) and hence knotted scaffolds can potentially be used to construct checkerboard colorable graph embeddings in non-toroidal (and non-spherical) surfaces. For torus graphs, we characterized Eulerian graphs which admit an embedding in a standard torus where all non-intersecting Eulerian circuits are knotted. These are precisely the Eulerian graphs which admit embeddings in a torus that are not checkerboard colorable (Theorem 7). This shows the existence of Eulerian graphs embedded in surfaces that require knotted scaffolds for construction. The results presented can suggest, for instance, reconditioning of graphs to meet checkerboard colorability so that unknotted scaffold routings can potentially be found. In general, knot theory of non-intersecting Eulerian circuits is also of theoretical interest, as suggested in [24].

We note that, although the problem was motivated by DNA-origami scaffold routings, the results presented could be applied for any routing of a circular strand that can be modelled as a non-intersecting circuit in a surface-embedded graph. This is because a circuit in a

graph can be considered as an Eulerian circuit of a subgraph. The study of surface-embedded graphs significantly expands the systematic ways of designing nanostructures, and the study of the topology of circuits on such graphs can be a useful guide in the design of topologically complex 3D nanostructures.

## References


1 Jaromir Abrham and Anton Kotzig. Construction of planar Eulerian multigraphs. In *Proc. Tenth Southeastern Conf. Comb., Graph Theory, and Computing*, pages 123–130, 1979.

2 Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994. doi:10.1126/SCIENCE.7973651.

3 Mark A. Armstrong. *Basic Topology*. Springer New York, 1983.

4 Erik Benson, Abdulmelik Mohammed, Alessandro Bosco, Ana I. Teixeira, Pekka Orponen, and Björn Högberg. Computer-aided production of scaffolded DNA nanostructures from flat sheet meshes. *Angewandte Chemie International Edition*, 55(31):8869–8872, 2016. doi:10.1002/anie.201602446.

5 Erik Benson, Abdulmelik Mohammed, Johan Gardell, Sergej Masich, Eugen Czeizler, Pekka Orponen, and Björn Högberg. DNA rendering of polyhedral meshes at the nanoscale. *Nature*, 523(7561):441–444, 2015. doi:10.1038/nature14586.

6 Samuel W. Bent and Udi Manber. On non-intersecting Eulerian circuits. *Discrete Applied Mathematics*, 18(1):87–94, 1987. doi:10.1016/0166-218X(87)90045-X.

7 Dorothy Buck, Egor Dolzhenko, Nataša Jonoska, Masahico Saito, and Karin Valencia. Genus ranges of 4-regular rigid vertex graphs. *Electronic Journal of Combinatorics*, 22(3):P3.43, 2015.

8 Junghuei Chen and Nadrian C. Seeman. Synthesis from DNA of a molecule with the connectivity of a cube. *Nature*, 350(6319):631–633, 1991. doi:10.1038/350631a0.

9 Joanna A. Ellis-Monaghan, Greta Pangborn, Nadrian C. Seeman, Sam Blakeley, Conor Disher, Mary Falcigno, Brianna Healy, Ada Morse, Bharti Singh, and Melissa Westland. Design tools for reporter strands and DNA origami scaffold strands. *Theoretical Computer Science*, 671:69–78, 2017. doi:10.1016/j.tcs.2016.10.007.

10 Herbert Fleischner. *Eulerian Graphs and Related Topics. Part 1, Volume 1*, volume 45 of *Annals of Discrete Mathematics*. North-Holland Publishing Co., Amsterdam, 1990.

11 R. P. Goodman, I. A. T. Schaap, C. F. Tardin, C. M. Erben, R. M. Berry, C. F. Schmidt, and A. J. Turberfield. Rapid chiral assembly of rigid DNA building blocks for molecular nanofabrication. *Science*, 310(5754):1661–1665, 2005. doi:10.1126/science.1120367.

12 Jonathan L. Gross and Thomas W. Tucker. *Topological Graph Theory*. Dover Publications, INC, 2001. Dover reprint, original published in 1987.

13 Yu He, Tao Ye, Min Su, Chuan Zhang, Alexander E. Ribbe, Wen Jiang, and Chengde Mao. Hierarchical self-assembly of DNA into symmetric supramolecular polyhedra. *Nature*, 452(7184):198–201, 2008. doi:10.1038/nature06597.

14 Ryosuke Iinuma, Yonggang Ke, Ralf Jungmann, Thomas Schlichthaerle, Johannes B. Woehrstein, and Peng Yin. Polyhedra self-assembled from DNA tripods and characterized with 3D DNA-PAINT. *Science*, 344(6179):65–69, 2014. doi:10.1126/science.1250944.

15 Nataša Jonoska, Stephen A. Karl, and Masahico Saito. Creating 3-dimensional graph structures with DNA. In Harvey Rubin and David H. Wood, editors, *DNA Based Computers III*, volume 48 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 123–136. AMS and DIMACS, 1999.

16 Nataša Jonoska and Masahico Saito. Boundary components of thickened graphs. In Nataša Jonoska and Nadrian C. Seeman, editors, *7th International Workshop on DNA-Based Computers*, volume 2340 of *Lecture Notes in Computer Science*, pages 70–81. Springer, 2001. doi:10.1007/3-540-48017-X_7.

**17** Hyungmin Jun, Tyson R. Shepherd, Kaiming Zhang, William P. Bricker, Shanshan Li, Wah Chiu, and Mark Bathe. Automated sequence design of 3D polyhedral wireframe DNA origami with honeycomb edges. *ACS Nano*, 13(2):2083–2093, 2019. doi:10.1021/acsnano.8b08671.

**18** Hyungmin Jun, Xiao Wang, William P. Bricker, and Mark Bathe. Automated sequence design of 2D wireframe DNA origami with honeycomb edges. *Nature Communications*, 10(5419):1–9, 2019. doi:10.1038/s41467-019-13457-y.

**19** Hyungmin Jun, Fei Zhang, Tyson Shepherd, Sakul Ratanalert, Xiaodong Qi, Hao Yan, and Mark Bathe. Autonomously designed free-form 2D DNA origami. *Science Advances*, 5(1), 2019. doi:10.1126/sciadv.aav0655.

**20** Vid Kočar, John S. Schreck, Slavko Čeru, Helena Gradišar, Nino Bašić, Tomaž Pisanski, Jonathan P. K. Doye, and Roman Jerala. Design principles for rapid folding of knotted DNA nanostructures. *Nature Communications*, 7:10803, 2016. doi:10.1038/ncomms10803.

**21** Ajasja Ljubetič, Fabio Lapenta, Helena Gradišar, Igor Drobnak, Jana Aupič, Žiga Strmšek, Duško Lainšček, Iva Hafner-Bratkovič, Andreja Majerle, Nuša Krivec, Mojca Benčina, Tomaž Pisanski, Tanja Ćirković Veličković, Adam Round, José María Carazo, Roberto Melero, and Roman Jerala. Design of coiled-coil protein-origami cages that self-assemble in vitro and in vivo. *Nature Biotechnology*, 35(11):1094–1101, 2017. doi:10.1038/nbt.3994.

**22** Abdulmelik Mohammed. *Algorithmic Design of Biomolecular Nanostructures*. PhD thesis, Aalto University, 2018.

**23** Abdulmelik Mohammed and Mustafa Hajij. Unknotted strand routings of triangulated meshes. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, volume 10467 of *Lecture Notes in Computer Science*, pages 46–63. Springer, 2017.

**24** Ada Morse, William Adkisson, Jessica Greene, David Perry, Brenna Smith, Jo Ellis-Monaghan, and Greta Pangborn. DNA origami and unknotted A-trails in torus graphs. *arXiv preprint arXiv:1703.03799*, 2017. arXiv:/arxiv.org/pdf/1703.03799.pdf.

**25** Dale Rolfsen. *Knots and Links*. AMS Chelsea Publishing, 2003. Reprint, original print in 1976.

**26** Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006. doi:10.1038/nature04586.

**27** Phiset Sa-Ardyen, Nataša Jonoska, and Nadrian C. Seeman. Self-assembling DNA graphs. *Natural Computing*, 2:427–438, 2003. doi:10.1023/B:NACO.0000006771.95566.34.

**28** Nadrian C. Seeman. Nucleic-acid junctions and lattices. *Journal of Theoretical Biology*, 99(2):237–247, 1982. doi:10.1016/0022-5193(82)90002-9.

**29** Nadrian C. Seeman and Neville R. Kallenbach. Design of immobile nucleic acid junctions. *Biophysical Journal*, 44(2):201–209, 1983. doi:10.1016/S0006-3495(83)84292-1.

**30** William M. Shih, Joel D. Quispe, and Gerald F. Joyce. A 1.7-kilobase single-stranded DNA that folds into a nanoscale octahedron. *Nature*, 427(6975):618–621, 2004. doi:10.1038/nature02307.

**31** Mu-Tsun Tsai and Douglas B. West. A new proof of 3-colorability of Eulerian triangulations. *Ars Mathematica Contemporanea*, 4(1):73–77, 2011.

**32** Rémi Veneziano, Sakul Ratanalert, Kaiming Zhang, Fei Zhang, Hao Yan, Wah Chiu, and Mark Bathe. Designer nanoscale DNA assemblies programmed from the top down. *Science*, 352(6293):1534, 2016. doi:10.1126/science.aaf4388.

**33** Gang Wu, Nataša Jonoska, and Nadrian C. Seeman. Construction of a DNA nano-object directly demonstrates computation. *Biosystems*, 98(2):80–84, 2009. doi:10.1016/j.biosystems.2009.07.004.

# Simplifying Chemical Reaction Network Implementations with Two-Stranded DNA Building Blocks

**Robert F. Johnson**
California Institute of Technology, Pasadena, CA, USA
rfjohnso@dna.caltech.edu

**Lulu Qian**
California Institute of Technology, Pasadena, CA, USA

## Abstract

In molecular programming, the Chemical Reaction Network model is often used to describe real or hypothetical systems. Often, an interesting computational task can be done with a known hypothetical Chemical Reaction Network, but often such networks have no known physical implementation. One of the important breakthroughs in the field was that any Chemical Reaction Network can be physically implemented, approximately, using DNA strand displacement mechanisms. This allows us to treat the Chemical Reaction Network model as a programming language and the implementation schemes as its compiler. This also suggests that it would be useful to optimize the result of such a compilation, and in general to find effective ways to design better DNA strand displacement systems.

We discuss DNA strand displacement systems in terms of "motifs", short sequences of elementary DNA strand displacement reactions. We argue that describing such motifs in terms of their inputs and outputs, then building larger systems out of the abstracted motifs, can be an efficient way of designing DNA strand displacement systems. We discuss four previously studied motifs in this abstracted way, and present a new motif based on cooperative 4-way strand exchange. We then show how Chemical Reaction Network implementations can be built out of abstracted motifs, discussing existing implementations as well as presenting two new implementations based on 4-way strand exchange, one of which uses the new cooperative motif. The new implementations both have two desirable properties not found in existing implementations, namely both use only at most 2-stranded DNA complexes for signal and fuel complexes and both are physically reversible. There are reasons to believe that those properties may make them more robust and energy-efficient, but at the expense of using more fuel complexes than existing implementation schemes.

**2012 ACM Subject Classification** Computer systems organization → Molecular computing

**Keywords and phrases** Molecular programming, DNA computing, Chemical Reaction Networks, DNA strand displacement

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.2

**Funding** *Robert F. Johnson*: NSF Graduate Research Fellowship.
*Lulu Qian*: NSF grant CCF-1908643.

**Acknowledgements** We would like to thank Chris Thachuk and Erik Winfree for helpful discussions on new DNA strand displacement motifs and optimization thereof.

## 1 Introduction

What does it mean to optimize a molecular system? One particular field in molecular programming is currently faced with that question. The Chemical Reaction Network (CRN) model is often used to describe systems of interacting molecules. The model can either describe real systems, to analyze their behavior and computational function, or describe hypothetical systems, with known computational function but perhaps no known physical

example. It was therefore a significant breakthrough when Soloveichik et al. showed that any CRN, real or hypothetical, can be approximately implemented by a system of DNA strand displacement (DSD) mechanisms [34]. This allows the Chemical Reaction Network model to be used as a programming language, where programs can be written in the abstract and compiled into physical molecules. Other CRN-to-DSD implementation schemes promptly followed [27, 4], each with their own strengths and weaknesses. Some have been implemented experimentally, with variable – but mostly good – degrees of success and robustness [7, 36]. Given a programming language and a concept of compiling it, one would naturally want to optimize the result of that compilation and ask, can we do better than the best implementation schemes so far?

So what does it mean to optimize a DSD system? We focus on DNA-only (or "enzyme-free") systems using standard toehold-mediated 3-way [45, 48] and 4-way [25, 10] strand displacement mechanisms. First, such DSD CRN implementations so far require "fuel species" (or "fuels"), DNA complexes that have to be synthesized by whatever method and added to the DSD system at the start. Fuel complexes that mediate a reaction by interacting with signal strands are often referred to as "gates", though this is not usually formally defined. When testing DSD circuits in the lab, fuels are chemically synthesized, annealed, and manually added to the test tube; in the hypothetical future where DSD is used in autonomous molecular devices, those devices would need some as-yet-undecided mechanism to synthesize or input fuels. Any property of the fuel species, such as length of strands, number of strands, or number of fuels, that makes them more costly to synthesize, or more difficult to synthesize without undesired byproducts, is thus a target for optimization. Second, no physical DSD system ever does exactly what the formal DSD model says it should. Some of this is due to improbable, but not impossible, "leak reactions" not included in the formal model, while some is due to the aforementioned undesired byproducts or other imperfect synthesis of the fuels [36].

In terms of robust DSD systems and their fuels, we can take a lesson from experiments with seesaw gates [28, 40]. For a two-reactant two-product reaction, the Soloveichik et al. translation scheme uses 3-stranded fuels [34], the Cardelli scheme 4-stranded fuels [4], and the Qian et al. scheme [27] (in the corrected version) a 5-stranded or a 7-stranded fuel. The seesaw gates compute logic gates which are less complex than chemical reactions, but they do so with only single strands and 2-stranded complexes [28]. Possibly because of this, they have been used to build larger circuits and to be robust to experimental imperfections, such as unpurified strands [40].

For this purpose, we have been investigating implementing CRNs using only 2-stranded fuels. Simple DSD systems, such as detecting a desired sequence [5] or AND gates [16], are often 2-stranded, in addition to the seesaw gates mentioned above. There is even a class of hairpin-based systems that construct larger structures from single-stranded initial complexes [44], including the Hybridization Chain Reaction often used in imaging [11], and a design for hairpin-based logic circuits [12]. However, none of these are a full Chemical Reaction Network implementation, or even an equivalently powerful dynamical system – while logic gates are universal for computing functions, CRNs have a dynamical behavior that logic gates in general do not.

We focus in this work on DSD systems using only 2-stranded fuels and where all mechanisms are physically reversible. We focus on 2-stranded fuels for the robustness concerns above, as well as the theoretical question of whether 2-stranded complexes are sufficient for complex behavior (as discussed further in [18]). We focus on physical reversiblility because it reduces the quantity of fuel consumed by reversible reactions. Many interesting computations and

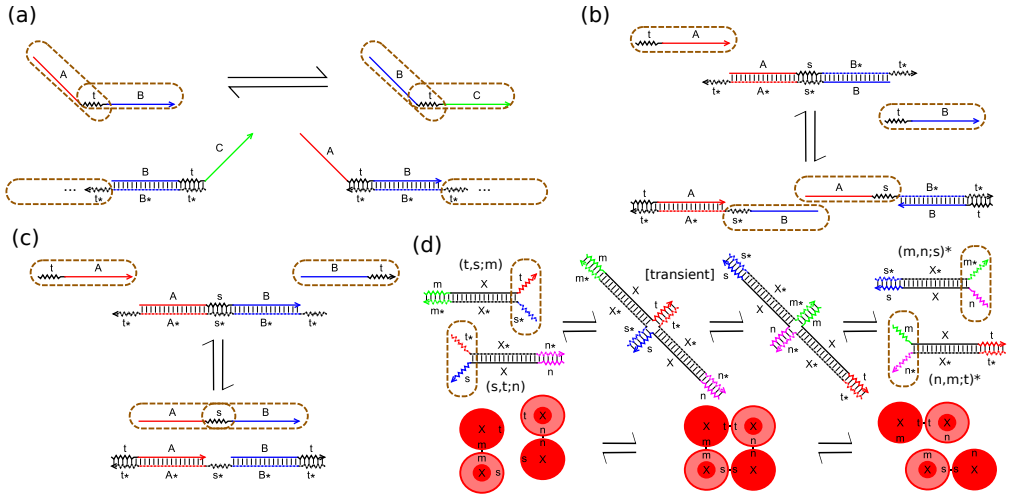



**Figure 1** Four previously studied reversible 2-stranded DSD motifs, shown through common examples. (a) Toehold exchange; (b) Symmetric cooperative hybridization; (c) Asymmetric cooperative hybridization; (d) 4-way strand exchange, with a diagram and names used in the abstracted notation we will introduce.

dynamical behaviors require reversible reactions. For example, logically reversible operations allow computation with arbitrarily low energy if they are implemented with physically reversible reactions [2, 3], such as DSD implementations of stack machines [27], Gray code counters [9], and space-bounded computations [37]. DNA buffers [29] use reversible reactions to maintain stable [30] and dynamical [31] spatial patterns. DNA circuits can be reset to process new input signals when reversible reactions are used for restoring fuel molecules in response to reset signals [14, 13, 12]. (Existing implementations often are or can be made physically reversible; Qian et al. [27] demonstrate it explicitly, while simple methods to make other existing schemes [34, 4] physically reversible is an exercise for the interested reader.)

In this work, we discuss ways of implementing CRNs using only 2-stranded fuels and where all mechanisms are physically reversible. We discuss four known 2-stranded DSD motifs that can serve as building blocks for such implementations, and we present a new cooperative 4-way strand exchange motif that starts with 2-stranded complexes. We discuss two ways of implementing general CRNs with these motifs, and tradeoffs between the two schemes. Finally, we show how, using CRN bisimulation, these schemes can be proven correct assuming the assumptions of the formal DSD model reflect real DSD systems.

We believe that having abstract descriptions of simple motifs will help the design of complex DSD systems. Whatever complex behavior is desired, it may be easier to implement by combining the simple logical operations of known motifs. To demonstrate this, we first discuss the 5 motifs and their behavior on an abstract level, then show how various CRN implementations can be constructed and comprehended by combining those abstract behaviors.

## 2 Two-stranded motifs

We identify five “motifs”, or simple condensed reactions, out of which we build two-stranded CRN implementations. Four of these motifs have been previously studied, while one is new. We discuss the properties of each motif in itself, while in Section 3 we will discuss how

those properties interact when building larger circuits. For building two-stranded CRNs, key questions about a given motif are what logical operation it represents, whether its outputs have the form of its inputs and/or the inputs of the other motifs, and whether its outputs and reverse gates are 2-stranded.

### Toehold Exchange

A reversible 3-way strand displacement exchanges which of two strands is bound to a gate (Figure 1 (a)). The input strand is an unbound toehold-long domain combination, while the input gate has that long domain bound with that toehold open. The reaction has two high-level effects. First, the output strand has the same long domain (B, in the figure) in a different toehold context, and may have different long domains (A versus C) on the other side of its newly open toehold. Second, the gate now has a different toehold open, which may allow interaction with adjacent domains. See for example the first CRN implementations [34], seesaw gates [28], and various others [47].

### Cooperative Hybridization (symmetric)

Two 3-way strand displacement reactions occur simultaneously on either side of a gate complex, meeting in the middle and allowing the two halves to dissociate only if both inputs are present (Figure 1 (b)). The input strands are unbound toehold-long domain combinations, while the output signals have the same long domains adjacent to different open toeholds. See for example Cherry et al.'s winner-take-all circuits [8]. This mechanism, like the two other cooperative motifs, is "cooperative" in the sense that it requires two inputs to simultaneously, "cooperatively", displace parts of the gate complexes for a productive reaction to happen.

### Cooperative Hybridization (asymmetric)

Two 3-way strand displacement reactions occur simultaneously on either side of a gate complex, meeting in the middle and releasing an output strand only if both inputs are present (Figure 1 (c)). The input strands are unbound toehold-long domain combinations, while the output strand has those two long domains in combination with a different toehold; but with only one toehold, barring complex mechanisms either one but only one of them can react. However, even if both inputs are single strands the reverse gate is a 3-stranded complex, so this motif is not "reversible with 2-stranded fuels". Introduced and tested by Zhang [46].

### 4-way Strand Exchange

Two 2-stranded complexes bind by two toeholds and exchange strands via 4-way branch migration (Figure 1 (d)). The inputs are 2-stranded complexes sharing a common long domain, with complementary pairs of open toeholds and (if the reaction is reversible) a closed toehold on each. The outputs are 2-stranded complexes in the same form, with the formerly open toeholds now paired up and closed and the formerly closed toeholds now split and open. Experimentally tested by Dabby [10]. Various mechanisms, simple and complex, based on 4-way strand exchange have been used experimentally in a number of devices [41, 24, 5, 16].

### 4-way Cooperative Hybridization

Two 4-way branch migrations happen on either side of a gate, meeting in the middle and separating into two intermediate complexes (Figure 2). Observe that the "top" toeholds ($t$ and $t$) on the initial $X$ and $Y$ complexes end up on one of the two products, while the

"bottom" toeholds ($s^*$ and $v^*$) end up on another. That is, each of the two products carries only half of the information of the original reactants, and products of different instances of this reaction can interact in the reverse reaction. If for example the $(t, t)$ top half of this reaction interacted with a $(v^*, s^*)$ bottom half from a different instance, while the $(s^*, v^*)$ bottom half interacted with an $(a, a)$ top half, the result would be $X$ and $Y$ complexes with the same form as the original reactants but different toehold combinations. The effect of such a quadruplet of reactions is strand exchange between one pair of complexes coupled to strand exchange between the other, simultaneously changing the open toehold combinations on distinct long domains. This is important because affecting distinct long domains in a coupled manner was the one thing that, under a set of additional restrictions that this mechanism satisfies, our previous work [18] showed that uncooperative 4-way strand exchange could not do.

While the other four mechanisms discussed have been experimentally demonstrated to work, cooperative 4-way branch migration has not yet been tested. In particular, the final dissociation step requires 3 toeholds separated by two 4-way junctions to dissociate. We think this is plausible, based on Dabby's observation that 2 toeholds separated by one 4-way junction can dissociate [10]; or, if this is not the case, that there is some $0 < Length(l) \leq 6$ for which that dissociation is possible and reversible. It is possible that $Length(l) = 0$ (i.e. no third toehold) will give the desired behavior, but from Dabby's results, "closed" (both toehold lengths at least 2) 4-way branch migration seems to proceed much faster than "open" 4-way branch migration. Thus we suspect that $Length(l) \neq 0$, and in particular $Length(l) \geq 2$, will give the desired fast and reversible reaction kinetics.

### An abstraction for 4-way-based mechanisms

Common to both uncooperative and cooperative 4-way strand exchange is a basic signal complex: two strands, one long domain bound to its complement flanked by one bound pair of complementary toeholds and one open pair of non-complementary toeholds, as seen repeatedly in Figures 1 (d) and 2. As both types of 4-way strand exchange transform complexes of this form into complexes of the same form with different domain combinations, we find an abstract description of this type of molecule useful. For example, we write the molecule with long domain $X$, open 3' (end of the DNA) toehold $t$, open 5' toehold $s^*$, and bound toehold $m$ as $X(t, s; m)$. Note that the semicolon distinguishes open toeholds $t, s^*$ available for interaction from the closed $(m, m^*)$ toehold pair that cannot interact with other complexes, but can be opened for interaction by a reaction. When the long domain is unimportant or universal, such as a system composed entirely of uncooperative 4-way strand exchange, we omit it and write simply $(t, s; m)$. For experimental reasons we prefer to have strands made up of only non-$*$ or only $*$ domains, and design non-$*$ and $*$ domains to have distinct sequence properties (for example, using a three-letter code [28]). Then $X(t, s; m)$ unambiguously describes the top reactant of Figure 1 (d), with $s$ understood to mean an open $s^*$ toehold. With that assumption, the top product in Figure 1 (d) would be $X(m, n; s)^*$, with the first toehold listed still being on the 3' end of its strand, but now understood to mean an open $m^*$ toehold. Without that assumption, we might use a more general notation where those molecules are $X(t, s^*; m)$ and $X^*(m^*, n; s^*)$ respectively. The circle abstraction shown in said figures is also useful to illustrate strand exchange reactions. Each circle represents a strand with one long domain and two toeholds, where half-faded circles represent strands made of $*$ domains. Thin connections (both figures) represent strands bonded directly, requiring matching domains; thick connections labelled with a toehold domain (horizontal in Figure 2) represent strands connected by gate strands from a cooperative 4-way strand exchange reaction, which can be between any domains so long as the appropriate gate exists.

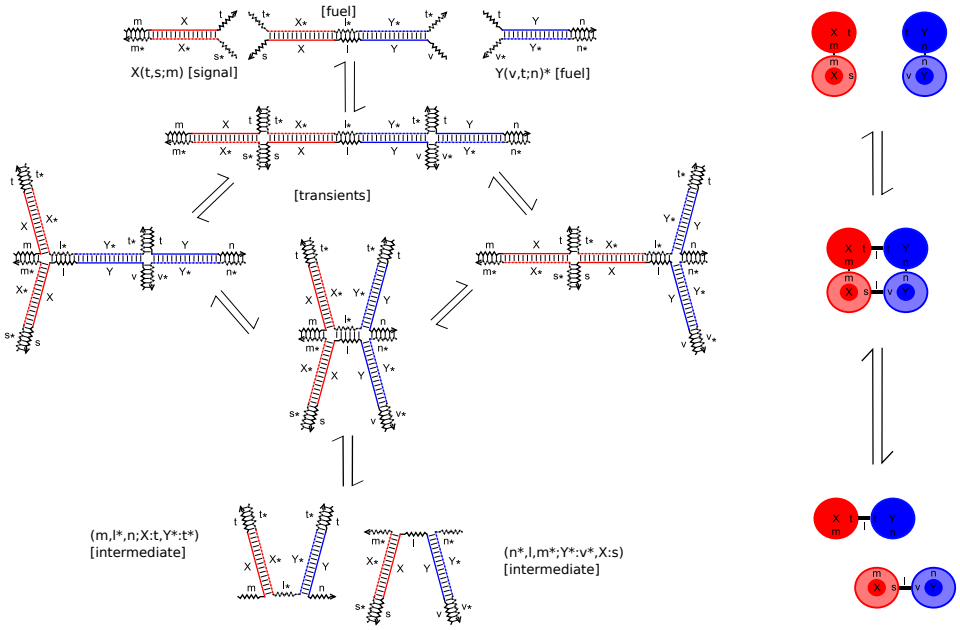



**Figure 2** A cooperative 4-way branch migration mechanism. Initial $X$ and $Y$ complexes combine with a gate that matches their open toehold combinations, producing two 3-stranded complexes each with one of the strands of $X$ and one of the strands of $Y$. These complexes can recombine with each other or with the corresponding products of a similar reaction, which in the latter case will produce $X$ and $Y$ complexes with different toehold combinations. On the right, this reaction is shown in abstracted form. The cooperative 4-way CRN is based on groups of four of these reactions, two in the reverse of the direction shown, where in the reverse reactions each product of one forward reaction interacts with the corresponding product of the other forward reaction. Complexes are labeled with names in the abstract notation if applicable, and their role in the cooperative 4-way CRN implementation scheme. "Signal" and "fuel" complexes have 2 strands as desired; stable "intermediate" complexes can have any number of strands; and "transient" complexes will quickly decay to one side or the other of the reaction. The marking of $X(t, s, m)$ as signal and $Y(v, t, n)^*$ as fuel is based on the CRN implementation scheme presented in Section 3, but in general the two can be any combination of signal and fuel, or could be intermediates of a more complex pathway.

In Figure 2 we introduce a similar notation for the "intermediate" products of a cooperative 4-way strand exchange reaction, in that case $(m, l^*, n; X : t, Y^* : t^*)$ and $(n^*, l, m^*; Y^* : v^*, X : s)$. Again the semicolon distinguishes the three open toeholds, listed from 5' to 3' end, from the bound long domain-toehold pairs; each of those pairs is listed as the domains that appear first in 5' to 3' order. Thus the full reaction is

$$X(t, s; m) + Y(v, t, n)^* \rightleftharpoons (m, l^*, n; X : t, Y^* : t^*) + (n^*, l, m^*; Y^* : v^*, X : s)$$

assuming the appropriate fuel (top center), which we do not give a notation to and omit from the reaction, is present.

## 3 Chemical Reaction Network implementations

The above motifs can be combined in various ways to construct implementations of arbitrary Chemical Reaction Networks. To implement arbitrary CRNs, the reaction $A + B \to C + D$ (or $A + B \to C$ and $A \to B + C$) is sufficient; for arbitrary reversible CRNs, the reaction $A + B \rightleftharpoons C$ (or *a fortiori*, $A + B \rightleftharpoons C + D$) is sufficient [26]. From a logical perspective, "join" and "fork" operations are sufficient; the above reactions represent those logics.

We take modular CRN bisimulation [19] as the definition of a "correct" CRN implementation scheme. Given that a scheme is correct, there are a number of other conditions that would be useful to satisfy for various reasons, theoretical and practical. CRN implementations typically have *signal* complexes that are the primary form of a given formal species, and *fuel* complexes that are assumed to be always present and drive the reactions. For a CRN to have "only 2-stranded inputs", as desired in this work, means that all signal complexes and fuel complexes are single strands or 2-stranded. We implicitly assume that we are discussing *systematic* CRN implementations, where we give a template for a generic reaction and construct larger CRNs by combining independent copies of the template with different domain identities. In such a case we can ask how the number of toehold domains scales, i.e. whether different reactions can use the same toeholds or have to create new ones; as toeholds are limited in length by thermodynamics, a system with $O(n)$ toeholds may be able to implement small CRNs but a system with $O(1)$ toeholds is better if possible. Whether a scheme requires cooperative mechanisms is worth noting. Finally, it is desirable for reversible reactions ($A + B \rightleftharpoons C + D$) to be implemented with physically reversible mechanisms, so that going forward and backward multiple times does not consume fuel; to be truly reversible, the 2-stranded fuel criterion should include the reverse fuels as well. For further discussion and formal definitions of these criteria, see [18], which also contains a proof that no CRN implementation scheme using only 4-way branch migration can satisfy all of them.

### Toehold Exchange-based CRNs

Existing CRN implementations [34, 27, 4] are often based on toehold exchange mechanisms where e.g. $A + B \rightarrow C$ is implemented by a toehold exchange reaction with $A$ opening a toehold on the gate for a reaction involving $B$. These schemes can be understood in light of the motifs previously discussed: the property of toehold exchange that a different toehold on the gate is opened allows join and fork logic. The property that the released strand has a different long domain/toehold combination is used to pass signals between gates. The same shared-toehold logic could also be used with 4-way branch migration instead of toehold exchange, similar to the 4-way-based AND gate [16] (although that gate itself uses a toehold hidden in a loop rather than a toehold shared between adjacent long domains, which is a line of investigation to be explored elsewhere).

Such a shared-toehold mechanism seems to require a 3-stranded complex for the gate molecule to achieve join logic, so it does not meet the goal of this paper, but is worth mentioning as the current state of the art. Another relevant mechanism using toehold exchange is the seesaw gate [28], where transduction logic combines with threshold logic to check whether the total amount of signal is more than either $A$ or $B$ can produce by itself. This achieves join logic for macroscopic signals but cannot satisfy criteria such as CRN bisimulation for individual molecules.

### 3-way Cooperative CRNs

The symmetric cooperative hybridization is $A + B \rightleftharpoons C + D$ logic, if we consider the same long domain in a different toehold context to be a different signal. Since toehold exchange reactions depend on the combination of long domain and toehold, this is valid. Thachuk et al. use a combination of symmetric cooperative hybridization and toehold exchange to implement leakless $A + B \rightarrow C + D$ reactions in exactly this manner [38, 39, 42]).

From our perspective, the only problem is that symmetric cooperative hybridization with 1-stranded inputs produces 2-stranded products, and toehold exchange with a 2-stranded input signal produces a 3-stranded reverse gate. For physically reversible reactions, this

**Table 1** List of species for the 4-way $O(n)$-toeholds reaction $A + B \rightleftharpoons C + D$, in the abstracted notation. Species in columns $A$, $B$, $C$, and $D$ represent the given formal species. Species in columns labeled $\emptyset$ are fuels and assumed to be always present. $a_i$ domains are toeholds specific to species $A$, and similarly for $B$, $C$, and $D$; $r_i$ domains are specific to the reaction $A + B \rightleftharpoons C + D$; this ensures no crosstalk with other pathways.

| $A$ | $\emptyset$ | $B$ | $\emptyset$ |
|---|---|---|---|
| $(a_1, a_2; a_3)$ | $(a_2, a_1; r_5)$ | $(b_1, b_2; b_3)$ | $(b_2, b_1; r_6)$ |
| $(r_5, a_3; a_1)^*$ | $(a_3, r_5; a_2)^*$<br>$(a_3, r_5; r_2)^*$ | $(r_6, b_3; b_1)^*$ | $(b_3, r_6; b_2)^*$<br>$(b_3, r_6; r_1)^*$ |
| $(r_2, a_1; r_5)$ | $(a_1, r_2; a_3)$<br>$(a_1, r_2; r_3)$ | $(r_1, b_1; r_6)$ | $(b_1, r_1; b_3)$<br>$(b_1, r_1; r_4)$ |
| $(r_3, r_5; r_2)^*$ | $(r_5, r_3; a_1)^*$<br>$(r_5, r_3; r_1)^*$ | $(r_4, r_6; r_1)^*$ | $(r_6, r_4; b_1)^*$<br>$(r_6, r_4; r_2)^*$ |
| $(r_1, r_2; r_3)$ | $(r_2, r_1; r_5)$ | $(r_2, r_1; r_4)$ | $(r_1, r_2; r_6)$ |

| $C$ | $\emptyset$ | $D$ | $\emptyset$ |
|---|---|---|---|
| $(c_1, c_2; c_3)$ | $(c_2, c_1; r_3)$ | $(d_1, d_2; d_3)$ | $(d_2, d_1; r_4)$ |
| $(c_3, r_3; c_2)^*$ | $(r_3, c_3; c_1)^*$<br>$(r_3, c_3; r_2)^*$ | $(d_3, r_4; d_2)^*$ | $(r_4, d_3; d_1)^*$<br>$(r_4, d_3; r_1)^*$ |
| $(c_2, r_2; r_3)$ | $(r_2, c_2; c_3)$<br>$(r_2, c_2; r_4)$ | $(d_2, r_1; r_4)$ | $(r_1, d_2; d_3)$<br>$(r_1, d_2; r_3)$ |
| $(r_3, r_4; r_2)^*$ | $(r_4, r_3; c_2)^*$ | $(r_4, r_3; r_1)^*$ | $(r_3, r_4; d_2)^*$ |

3-stranded gate would be considered a reverse fuel, and the system would not be made with entirely 2-stranded fuels. Thus this mechanism meets all our criteria for irreversible CRNs, but not reversible CRNs.

### 4-way-based CRNs with $O(n)$ toeholds

The two-toehold-mediated 4-way strand exchange mechanism effectively exchanges toeholds on a common long domain; note that while the inputs both have $t$ and $s$ toeholds, the outputs have one with only $t$ and one with only $s$. When a signal complex goes through multiple copies of this reaction with different fuels, it can turn any combination of toeholds into any other combination. When two signals with complementary pairs of toeholds meet in this reaction, it produces two signals with different combinations in $A + B \rightleftharpoons C + D$ logic. So for example, we can turn $(a_1, a_2; a_3)$ into $(r_1, r_2; r_3)$ and $(b_1, b_2; b_3)$ into $(r_2, r_1; r_4)$, which will react and produce $(r_3, r_4; r_2)^*$ and $(r_4, r_3; r_1)^*$, which can be turned into $(c_1, c_2; c_3)$ and $(d_1, d_2; d_3)$ respectively. Thus two-toehold-mediated 4-way strand exchange alone can implement arbitrary reversible CRNs if we allow $O(n)$ toeholds.

A list of all species involved is given in Table 1. Note that fuels $(r_2, r_1; r_5)$ and $(r_1, r_2; r_6)$ can interact, but the products can do nothing but reverse the reaction, and the same is true for $(r_4, r_3; c_2)^*$ with $(r_3, r_4; d_2)^*$.

### 4-way Cooperative CRNs

The cooperative 4-way strand exchange motif in Figure 2, when its products recombine with products of a different instance of the reaction, *simultaneously* exchanges the toehold combinations on a complex with long domain $X$ and a complex with long domain $Y$. If

$A(t,s;m)$ is the signal molecule for $A$, then simultaneously breaking the $(t,s)$ combination on $A$ and putting together a $(u,v)$ combination on some long domain $R$ is effectively converting $A(t,s;m) \rightleftharpoons R(v,u;n)^*$ if all other molecules involved are considered fuels. Where $R$ is unique to the reaction $A+B \rightleftharpoons C+D$, we can convert the four signal species from their own long domains to the $R$ domain, then use a two-toehold-mediated 4-way strand exchange reaction to implement the reaction itself. In contrast to the previous implementation scheme, that each reaction has a different long domain allows the toeholds ($u$, $v$, etc.) to be universal, using $O(1)$ toeholds at the expense of requiring cooperative hybridization. In the notation used in Figure 2, this quadruplet of reactions (with the appropriate top-center fuels assumed present but not written) is

$$A(t,s;m) + R(s,u;n)^* \rightleftharpoons (m,l^*,n;A:t,R^*:u^*) + (n^*,l,m^*:R^*:s^*,A:s)$$
$$A(u,v;m) + R(v,t;n)^* \rightleftharpoons (m,l^*,n;A:u,R^*:t^*) + (n^*,l,m^*:R^*:v^*,A:v)$$
$$(m,l^*,n;A:u,R^*:t^*) + (n^*,l,m^*:R^*:s^*,A:s) \rightleftharpoons A(u,s;m) + R(s,t;n)^*$$
$$(m,l^*,n;A:t,R^*:u^*) + (n^*,l,m^*:R^*:v^*,A:v) \rightleftharpoons A(t,v;m) + R(v,u;n)^*$$

where $A(t,s;m)$ and $R(v,u;n)^*$ are the designated meaningful complexes. The other 2-stranded complexes – $A(u,v;m)$, $A(u,s;m)$, $A(t,v;m)$, $R(s,u;n)^*$, $R(v,t;n)^*$, and $R(s,t;n)^*$ are treated as fuels and assumed always present. If this motif works as hypothesized and without leak, $R(v,u;n)^*$ can only be produced by consuming $A(t,s;m)$ and vice versa.

As this scheme is based on the $O(n)$-toehold scheme, we reuse the mechanism from Table 1. Assume all complexes in that list have long domain $R$, unique to the reaction $A+B \rightleftharpoons C+D$. To the toeholds listed, add toeholds $t,s,m,n,l$, and let $a_3 = b_3 = c_3 = d_3 = n^*$, with $u$ and $v$ in the above quadruplet renamed appropriately. Then use cooperative 4-way strand exchange to convert $A(t,s;m) \rightleftharpoons (R^*(a_1^*,a_2^*;n))^* = R(a_1,a_2;n^*)$ (the fuel will have $R^*$ on the "top" strand with $A$), $B(t,s;m) \rightleftharpoons R(b_1,b_2;n^*)$, $C(t,s;m) \rightleftharpoons R(c_1,c_2;n^*)$, and $D(t,s;m) \rightleftharpoons R(d_1,d_2;n^*)$. This gives a mechanism with one long domain per species, one long domain per reaction, and a total of 19 toeholds. Because the long domains now indicate species/reaction identity, the toeholds can be shared between all species and reactions without crosstalk.

## 4 Correctness of the schemes

The correctness of the schemes can be verified by CRN bisimulation, a formal definition of correctness of a CRN implementation that implies several desirable properties [19]. Below we give an intuitive explanation of why the schemes are correct that parallels the definition of CRN bisimulation; readers familiar with CRN bisimulation can fill in the details of the formal proof. Intuitively, CRN bisimulation consists of *interpreting* each DNA complex as zero or more formal species, then confirming that the behavior of the formal system and the interpreted DSD system are the same from any initial state. That is to say, any reaction of the DNA complexes should be interpreted as a reaction of formal species that is either valid or trivial ("anything that can happen, should"), and any reaction of the formal interpretation of a set of DNA complexes should be possible, perhaps after some trivial reactions, starting from that set of DNA complexes ("anything that should happen, can").

Table 1 is effectively a proof of the correctness of the $O(n)$-toehold 4-way-based scheme according to CRN bisimulation [19]. For each $A+B \rightleftharpoons C+D$ reaction, construct a copy of this mechanism with unique $r_i$ domains, but any $a_i$ domains in common with other reactions using the same formal species; reactions with fewer reactants or products can have one of $A$, $B$, $C$, or $D$ as a fuel; reactions with more reactants or products should

be broken into steps with at most 2 of each [26]. DNA complexes in columns labeled $A$, $B$, $C$, or $D$ are interpreted as one copy of the corresponding species, while complexes in columns labeled $\emptyset$ are fuels. Formally, fuels are assumed always present and removed from the enumerated implementation CRN before bisimulation verification; so for example the physical pathway $(r_2, a_2; r_3) + (a_2, r_2; r_5) \rightleftharpoons (r_5, r_3; r_2)^* + (r_3, r_5; a_2)^*$ would be represented as $(r_2, a_2; r_3) \rightleftharpoons (r_5, r_3; r_2)^*$, and then interpreted as the trivial reaction $A \rightleftharpoons A$. Using the abstraction for 4-way strand exchange notation, the table is structured such that each non-fuel species can interact with the (usually two) fuel species in the same row, producing the corresponding fuel+non-fuel pair above or below it; that the final $A + B$ forms react to produce the final $C + D$ forms, while their fuels also have a spurious-but-harmless reaction with each other; and that, given the uniqueness of the domains, no other intra-module or inter-module reactions exist. In CRN bisimulation, we say that a reaction interpreted as, for example, $A \rightleftharpoons A$ is "trivial", and in this case all reactions are trivial except $(r_1, r_2; r_3) + (r_2, r_1; r_4) \rightleftharpoons (r_3, r_4; r_2)^* + (r_4, r_3; r_1)^*$ which is interpreted as the desired reaction $A + B \rightleftharpoons C + D$. With $(a_1, a_2; a_3)$ etc. as the signal species, one can see that the signal species can implement the formal reaction, and any intermediate species can turn into the common species with the same interpretation by interacting with only fuels. Intuitively this is a good argument for correctness, and readers familiar with CRN bisimulation will recognize the above as a sufficient condition for modular CRN bisimulation with respect to the signal species as common species.

For the cooperative 4-way scheme, the same bisimulation logic applies. In the notation used in Figure 2 and Section 3, in e.g. $A(t, s; m) \rightleftharpoons R(a_1, a_2, n^*)$ the signal complex $A(t, s; m)$, output complex $R(a_1, a_2, n^*)$, and intermediate $(m, l^*, n; A : t, R : a_2)$ all interpreted as $A$, while the other three intermediates and all the fuels will each be interpreted as nothing. From there the bisimulation proof follows the $O(n)$-toeholds case. In this case the lack of crosstalk between modules is assured by the distinct long domains; even if toehold combinations are identical, different long domains will make the reaction unproductive. The remaining caveat is with the cooperative 4-way mechanism itself. We designed the system so that the toeholds along the cooperative reaction are *always* $m, l, n$. Thus, we *assume* that intermediates of the cooperative pathway will all have the matching $m, l, n$ toeholds, and all three toeholds will bind and dissociate as a unit. Whether this is actually true or not will be determined experimentally; if not, there may be problematic crosstalk between, for example, an $(A, R_1)$ and $(A, R_2)$ pair of long domains which leads to temporarily duplicated signals. If it is true, however, then the result of such a crosstalk will be a release of one side with the other suspended, one of which carries the signal, and the system will be correct according to bisimulation.

## 5 Discussion

We discussed the use of DNA Strand Displacement to implement Chemical Reaction Networks, and the desire to create larger, more robust DSD CRN implementations. We then presented 2-stranded DSD motifs which we used to build 2-stranded CRN implementations, in the hope that they would be more robust than those which rely on 3-or-more-stranded complexes. There is some indication that 2-stranded DSD systems in general are more robust (as we briefly reviewed in the introduction), but whether these particular systems are more robust than the current state-of-the-art CRN implementations is an open question.

We can compare Soloveichik et al.'s original CRN scheme [34, 36] (which is reasonably representative of other toehold exchange schemes), our $O(n)$-toehold 4-way strand exchange scheme, and our ($O(1)$-toehold) cooperative 4-way strand exchange scheme. While 3- and

4-stranded complexes may be less robust, in other aspects the toehold exchange scheme is simpler than our two schemes: it uses one long domain per formal species, one long domain per reaction, and can be done with a single, universal toehold. To go from reactant signal species to product signal species in the toehold exchange scheme (as implemented experimentally [36]) takes 4 toehold exchange steps in an $A + B \to C + D$ reaction, and generalizes naturally to $n + m$ steps in an $n$-reactant $m$-product reaction. In contrast, while the cooperative 4-way scheme also uses one long domain per formal species and reaction, as described above it uses 19 universal toeholds and takes 30 reactions for $A + B \to C + D$. (By "reaction" we mean roughly one condensed reaction as described in Peppercorn [17], generalized to include trimolecular reactions. So one toehold exchange or one 2-toehold-mediated 4-way strand exchange is one reaction, as is the cooperative 4-way strand exchange shown in Figure 2; note that using that mechanism to exchange e.g. $A(t, s; m) \rightleftharpoons R(a_1, a_2; n^*)$ takes 4 such reactions.) The $O(n)$-toeholds scheme takes only 14 reactions for $A + B \to C + D$, but with one universal long domain it takes 3 toeholds per species and 6 per reaction, which may run out of design space for large CRNs. Also, 14 reactions is still much more than 4. These pathways are not provably optimal; we suspect they can be reduced to less than 14 and 30, but still more than 4.

The increase in number of reactions to implement $A + B \to C + D$ may just be a cost of using 2-stranded complexes. The fundamental question is, given a complex of a certain size, how much information can it store? How can complexes meant to represent $A$, $C$, and an $E$ from another reaction all present different enough open and bound domains that none can undergo a reaction meant for a different one? With 3-stranded complexes and toehold exchange, the long domain identity and open toehold does this very efficiently. With 2-stranded complexes and 4-way strand exchange, we use pairs of toehold identity to represent signal identity, which means we need extra reactions to (a) change the toehold identity one strand at a time, and (b) ensure that intermediates of different pathways don't try to pass through the same toehold combination.

This question, then, connects to another work of ours. The final result of that work was a proof that a systematic CRN implementation that satisfies certain desirable conditions, including using only 2-stranded inputs and the other conditions discussed at the beginning of Section 3, cannot be done with DSD using only 4-way branch migration [18]. The steps taken to prove that result involve questions of what sort of transformations are possible with DSD reactions, and how and whether the possibility of certain transformations can depend on the features of the strands. This "dependence" is in the sense that the release of a strand in toehold exchange "depends on" the incoming strand having the correct toehold and long domain identities, or the way we have to structure our CRN implementations so that production of the output species depends on the inputs having the correct toehold identity pairs. Thus, further exploration of that line of investigation might help answer some of the questions suggested by the mechanisms in this paper, of whether 2-stranded complex based CRN implementations inherently require longer pathways, and quantitatively how much longer. Moreover, the investigation could be expanded to include other CRN implementations involving enzymes. For example, transcriptional circuits [20, 21], PEN-DNA toolbox [23, 1], primer exchange reaction cascades [22], and strand-displacing polymerase systems [35, 32, 33] all have elementary reactions that can be abstracted as motifs and are candidates for formal analysis. In these systems, it is possible to start with fewer and simpler fuel molecules (e.g. single strands only) while more complex molecules can be generated by DNA polymerase to carry out desired reactions. In addition to 3-way and 4-way strand displacement with standard toeholds, other mechanisms could also be investigated,

including remote [15], associative [6], and allosteric [43] toeholds. These mechanisms may allow further simplification of the implementations as they enrich the design space with alternative representations of signals.

It is also worth discussing how we discovered the cooperative 4-way strand exchange motif and associated CRN implementation in the process of working out the impossibility proof in [18]. We give an intuitive list of those conditions at the beginning of Section 3, but readers desiring a formal list of conditions should see [18]. Two of the conditions are using only $O(1)$ toeholds and not using cooperative mechanisms, so both the $O(n)$ toeholds uncooperative 4-way strand exchange based scheme and the $O(1)$ toeholds cooperative 4-way strand exchange based scheme satisfy all but one of the conditions, each failing to satisfy a different one. Thus in some sense this paper is the positive counterpart to the previous negative result, forming a tight upper and lower bound on the complexity of DSD implementations of CRNs. But this pair of results also has implications for design of DSD systems. The cooperative 4-way strand exchange motif and the process by which we came up with it is potentially a proof of concept that, in systematically eliminating possibilities in DSD systems, we can find new motifs in whatever remains. How exactly this can be generalized we do not know, but if it can be, it may make the process of designing DSD systems faster and more systematic.

Another aspect worth mentioning is the focus on motifs before building up CRN implementations. We argued that each of the 5 motifs has certain abstract behaviors, and that larger systems such as CRN implementations can be thought of in terms of those behaviors. When building large systems, it is much easier if one can build mid-sized building blocks out of the fundamental units, then build larger systems out of the mid-sized building blocks. Motifs take that role between fundamental DSD steps (bind, unbind, 3-way branch migration, 4-way branch migration) and systems on the scale of CRN implementations. To the extent that we were able to describe our CRN implementations in terms of the motifs rather than in terms of the underlying DSD steps, this approach should be considered for future DSD system design.

## References


1 Nathanaël Aubert, Clément Mosca, Teruo Fujii, Masami Hagiya, and Yannick Rondelez. Computer-assisted design for scaling up systems based on DNA reaction networks. *Journal of The Royal Society Interface*, 11(93):20131167, 2014.

2 Charles H Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973.

3 Charles H Bennett. The thermodynamics of computation—a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1982.

4 Luca Cardelli. Two-domain DNA strand displacement. *Mathematical Structures in Computer Science*, 23(02):247–271, 2013.

5 Sherry Xi Chen, David Yu Zhang, and Georg Seelig. Conditionally fluorescent molecular probes for detecting single base changes in double-stranded DNA. *Nature Chemistry*, 5(9):782, 2013.

6 Xi Chen. Expanding the rule set of DNA circuitry with associative toehold activation. *Journal of the American Chemical Society*, 134(1):263–271, 2012.

7 Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature Nanotechnology*, 8(10):755–762, 2013.

8 Kevin M Cherry and Lulu Qian. Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. *Nature*, 559(7714):370, 2018.

**9** Anne Condon, Alan J Hu, Ján Maňuch, and Chris Thachuk. Less haste, less waste: on recycling and its limits in strand displacement systems. *Interface Focus*, 2(4):512–521, 2012.

**10** Nadine L Dabby. *Synthetic molecular machines for active self-assembly: prototype algorithms, designs, and experimental study*. PhD thesis, California Institute of Technology, February 2013.

**11** Robert M Dirks and Niles A Pierce. Triggered amplification by hybridization chain reaction. *Proceedings of the National Academy of Sciences*, 101(43):15275–15278, 2004.

**12** Abeer Eshra, Shalin Shah, Tianqi Song, and John Reif. Renewable DNA hairpin-based logic circuits. *IEEE Transactions on Nanotechnology*, 18:252–259, 2019.

**13** Sudhanshu Garg, Shalin Shah, Hieu Bui, Tianqi Song, Reem Mokhtar, and John Reif. Renewable time-responsive DNA circuits. *Small*, 14(33):1801470, 2018.

**14** Anthony J Genot, Jonathan Bath, and Andrew J Turberfield. Reversible logic circuits made of DNA. *Journal of the American Chemical Society*, 133(50):20080–20083, 2011.

**15** Anthony J Genot, David Yu Zhang, Jonathan Bath, and Andrew J Turberfield. Remote toehold: a mechanism for flexible control of DNA hybridization kinetics. *Journal of the American Chemical Society*, 133(7):2177–2182, 2011.

**16** Benjamin Groves, Yuan-Jyue Chen, Chiara Zurla, Sergii Pochekailov, Jonathan L Kirschman, Philip J Santangelo, and Georg Seelig. Computing in mammalian cells with nucleic acid strand exchange. *Nature Nanotechnology*, 11(3):287, 2016.

**17** Casey Grun, Karthik Sarma, Brian Wolfe, Seung Woo Shin, and Erik Winfree. A domain-level DNA strand displacement reaction enumerator allowing arbitrary non-pseudoknotted secondary structures. *CoRR*, 2015. URL: `http://arxiv.org/abs/1505.03738`, `arXiv:1505.03738`.

**18** Robert F. Johnson. Impossibility of sufficiently simple chemical reaction network implementations in DNA strand displacement. In Ian McQuillan and Shinnosuke Seki, editors, *Unconventional Computation and Natural Computation*, pages 136–149. Springer International Publishing, 2019. `doi:10.1007/978-3-030-19311-9_12`.

**19** Robert F Johnson, Qing Dong, and Erik Winfree. Verifying chemical reaction network implementations: A bisimulation approach. *Theoretical Computer Science*, 2018. `doi:10.1016/j.tcs.2018.01.002`.

**20** Jongmin Kim, John Hopfield, and Erik Winfree. Neural network computation by in vitro transcriptional circuits. In *Advances in Neural Information Processing systems*, pages 681–688, 2005.

**21** Jongmin Kim and Erik Winfree. Synthetic in vitro transcriptional oscillators. *Molecular Systems Biology*, 7(1):465, 2011.

**22** Jocelyn Y Kishi, Thomas E Schaus, Nikhil Gopalkrishnan, Feng Xuan, and Peng Yin. Programmable autonomous synthesis of single-stranded DNA. *Nature Chemistry*, 10(2):155, 2018.

**23** Kevin Montagne, Raphael Plasson, Yasuyuki Sakai, Teruo Fujii, and Yannick Rondelez. Programming an in vitro DNA oscillator using a molecular networking strategy. *Molecular Systems Biology*, 7(1):466, 2011.

**24** Richard A Muscat, Jonathan Bath, and Andrew J Turberfield. A programmable molecular robot. *Nano letters*, 11(3):982–987, 2011.

**25** Igor G Panyutin and Peggy Hsieh. The kinetics of spontaneous DNA branch migration. *Proceedings of the National Academy of Sciences*, 91(6):2021–2025, 1994.

**26** Tomislav Plesa. Stochastic approximation of high-molecular by bi-molecular reactions. *arXiv preprint arXiv:1811.02766*, 2018.

**27** Lulu Qian, David Soloveichik, and Erik Winfree. Efficient Turing-universal computation with DNA polymers. In Yasubumi Sakakibara and Yongli Mi, editors, *DNA Computing and Molecular Programming*, volume 6518 of Lecture Notes in Computer Science, pages 123–140. Springer, 2011.

**28** Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

**29** Dominic Scalise, Nisita Dutta, and Rebecca Schulman. DNA strand buffers. *Journal of the American Chemical Society*, 140(38):12069–12076, 2018.

**30** Dominic Scalise and Rebecca Schulman. Designing modular reaction-diffusion programs for complex pattern formation. *Technology*, 2(01):55–66, 2014.

**31** Dominic Scalise and Rebecca Schulman. Emulating cellular automata in chemical reaction-diffusion networks. *Natural Computing*, 15(2):197–214, 2016.

**32** Shalin Shah, Tianqi Song, Xin Song, Ming Yang, and John Reif. Implementing arbitrary CRNs using strand displacing polymerase. In *International Conference on DNA Computing and Molecular Programming*, pages 21–36. Springer, 2019.

**33** Shalin Shah, Jasmine Wee, Tianqi Song, Luis Ceze, Karin Strauss, Yuan-Jyue Chen, and John Reif. Using strand displacing polymerase to program chemical reaction networks. *Journal of the American Chemical Society*, 2020.

**34** David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

**35** Tianqi Song, Abeer Eshra, Shalin Shah, Hieu Bui, Daniel Fu, Ming Yang, Reem Mokhtar, and John Reif. Fast and compact DNA logic circuits based on single-stranded gates using strand-displacing polymerase. *Nature Nanotechnology*, 14(11):1075–1081, 2019.

**36** Niranjan Srinivas, James Parkin, Georg Seelig, Erik Winfree, and David Soloveichik. Enzyme-free nucleic acid dynamical systems. *Science*, 358:doi:10.1126/science.aal2052, 2017.

**37** Chris Thachuk and Anne Condon. Space and energy efficient computation with DNA strand displacement systems. In *International Workshop on DNA-Based Computers*, pages 135–149. Springer, 2012.

**38** Chris Thachuk and Erik Winfree. A fast, robust, and reconfigurable molecular circuit breadboard. 15th Annual Conference on Foundations of Nanoscience, invited talk, 2018. URL: `https://thachuk.com/talk/2018-fnano-invited/2018-FNANO-invited.pdf`.

**39** Chris Thachuk, Erik Winfree, and David Soloveichik. Leakless DNA strand displacement systems. In Andrew Phillips and Peng Yin, editors, *DNA Computing and Molecular Programming*, volume 9211 of Lecture Notes in Computer Science, pages 133–153. Springer, 2015.

**40** Anupama J Thubagere, Chris Thachuk, Joseph Berleant, Robert F Johnson, Diana A Ardelean, Kevin M Cherry, and Lulu Qian. Compiler-aided systematic construction of large-scale DNA strand displacement circuits using unpurified components. *Nature Communications*, 8:14373, 2017.

**41** Suvir Venkataraman, Robert M Dirks, Paul WK Rothemund, Erik Winfree, and Niles A Pierce. An autonomous polymerization motor powered by DNA hybridization. *Nature Nanotechnology*, 2(8):490, 2007.

**42** Boya Wang, Chris Thachuk, Andrew D Ellington, Erik Winfree, and David Soloveichik. Effective design principles for leakless strand displacement systems. *Proceedings of the National Academy of Sciences*, 115(52):E12182–E12191, 2018.

**43** Xiaolong Yang, Yanan Tang, Sarah M Traynor, and Feng Li. Regulation of DNA strand displacement using an allosteric DNA toehold. *Journal of the American Chemical Society*, 138(42):14076–14082, 2016.

**44** Peng Yin, Harry MT Choi, Colby R Calvert, and Niles A Pierce. Programming biomolecular self-assembly pathways. *Nature*, 451(7176):318–322, 2008.

**45** Bernard Yurke and Allen P Mills. Using DNA to power nanostructures. *Genetic Programming and Evolvable Machines*, 4(2):111–122, 2003.

**46** David Yu Zhang. Cooperative hybridization of oligonucleotides. *Journal of the American Chemical Society*, 133(4):1077–1086, 2010.

**47** David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature Chemistry*, 3(2):103–113, 2011.

**48** David Yu Zhang and Erik Winfree. Control of DNA strand displacement kinetics using toehold exchange. *Journal of the American Chemical Society*, 131(47):17303–17314, 2009.

# Composable Computation in Leaderless, Discrete Chemical Reaction Networks

**Hooman Hashemi**
The University of British Columbia, Vancouver, Canada

**Ben Chugg**
Stanford University, CA, USA
benchugg@stanford.edu

**Anne Condon**
The University of British Columbia, Vancouver, Canada
condon@cs.ubc.ca

## Abstract

We classify the functions $f : \mathbb{N}^d \to \mathbb{N}$ that are stably computable by leaderless, output-oblivious discrete (stochastic) Chemical Reaction Networks (CRNs). CRNs that compute such functions are systems of reactions over species that include $d$ designated input species, whose initial counts represent an input $\mathbf{x} \in \mathbb{N}^d$, and one output species whose eventual count represents $f(\mathbf{x})$. Chen et al. showed that the class of functions computable by CRNs is precisely the semilinear functions. In output-oblivious CRNs, the output species is never a reactant. Output-oblivious CRNs are easily composable since a downstream CRN can consume the output of an upstream CRN without affecting its correctness. Severson et al. showed that output-oblivious CRNs compute exactly the subclass of semilinear functions that are eventually the minimum of quilt-affine functions, i.e., affine functions with different intercepts in each of finitely many congruence classes. They call such functions the output-oblivious functions. A leaderless CRN can compute only superadditive functions, and so a leaderless output-oblivious CRN can compute only superadditive, output-oblivious functions. In this work we show that a function $f : \mathbb{N}^d \to \mathbb{N}$ is stably computable by a leaderless, output-oblivious CRN if and only if it is superadditive and output-oblivious.

**2012 ACM Subject Classification** Theory of computation → Models of computation; Theory of computation → Formal languages and automata theory

**Keywords and phrases** Chemical Reaction Networks, Stable Function Computation, Output-Oblivious, Output-Monotonic

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.3

**Funding** *Hooman Hashemi*: Supported by an NSERC Discovery Grant.
*Ben Chugg*: Supported by an NSERC Undergraduate Research Award.
*Anne Condon*: Supported by an NSERC Discovery Grant.

**Acknowledgements** This work benefited greatly from conversations with Eric Severson and David Doty. Thanks also to David Haley and Eric Severson for help in generating the figures.

## 1 Introduction

Chemical Reaction Networks (CRNs) have proven to be very valuable as a programming language for describing how computations can ensue when molecules react. There is now a rich complexity theory of computation with the CRN model, as well as the closely related population protocol model of distributed computing [2, 4, 7, 10, 11, 17]. This theory helps us understand what types of computational or engineered dynamic processes are possible with molecules, since CRNs can be "compiled" down to DNA strand displacement systems, which in turn can be implemented with real DNA strands in a test tube [5, 15, 18, 19].


It is natural to ask: If CRNs $C$ and $C'$ compute functions $f$ and $f'$, respectively, can we compose the CRNs to compute the composition $f' \circ f$? In this paper we study this question for leaderless, discrete CRNs, resolving an open question of Chugg et al. [9], Severson et al. [16], and Chalk et al. [6]. Here we first describe the CRN model, background and motivation for the work, and then describe our result in more detail.

We focus on discrete CRNs (also called stochastic CRNs), which are described as a finite set of chemical reactions among abstract species. Discrete CRNs *stably* compute functions $f : \mathbb{N}^d \to \mathbb{N}$ in the following sense. An input $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{N}^d$ is represented by initial counts of $d$ designated molecular species. A single copy of a so-called leader molecule may also be present initially. Reactions of the CRN ensue, changing the species counts over time. Eventually, regardless of the order of reactions, the count of a designated output species $Y$ equals $f(\mathbf{x})$ and does not subsequently change. See Figure 1. Here and throughout, we assume without loss of generality that the range of $f$ is $\mathbb{N}$, since functions that map $\mathbb{N}^d$ to $\mathbb{N}^l$ for some $l > 1$ can be computed by first cloning $l$ distinct copies of the inputs, and then for each $1 \leq i \leq l$, computing the $i$th output from the $i$th copy of the inputs.

$$\begin{array}{lll} X_1' + X_1' & \to 2Y' & \\ X_1' + X_2' & \to 2Y' & \\ X_2' + X_2' & \to 2Y' & \\ \text{(a)} & & \end{array} \qquad \begin{array}{lll} & & \\ X_2'' & \to 2Y'' & \\ & & \\ & \text{(b)} & \end{array} \qquad \begin{array}{lll} X_1 & \to X_1' + X_1'' & \\ X_2 & \to X_2' + X_2'' & \\ Y' + Y'' & \to Y & \\ & \text{(c)} & \end{array}$$

**Figure 1** Examples of Chemical Reaction Networks (CRNs) for stable function computation. (a) A CRN $C_1$ for $f(x_1, x_2) = x_1 + x_2 + ((x_1 + x_2) \mod 2)$, with inputs $X_1', X_2'$ and output $Y'$. (b) A CRN $C_2$ for $f'(x_1, x_2) = 2x_2$, with inputs $X_1'', X_2''$ and output $Y''$. (The input $X_1''$ does not appear in the reaction.) (c) A CRN $C$ for the function $\min\{f(x_1, x_2), 2x_2\}$. $C$ converts its inputs $X_1, X_2$ to those needed by CRNs $C_1$ and $C_2$ of parts (a) and (b), and then computes the function $\min\{f(x_1, x_2), 2x_2\}$ from the outputs of $C_1$ and $C_2$, demonstrating function composition. All three CRNs are leaderless.

Exactly the semilinear predicates and functions are stably computable by discrete CRNs [2, 7]. Such functions are linear on each of a finite number of semilinear domains – subsets of $\mathbb{N}^d$ that are defined using $\geq$ or mod. See Figure 2.

Let $C$ and $C'$ be discrete CRNs that stably compute functions $f : \mathbb{N}^d \to \mathbb{N}$ and $f' : \mathbb{N} \to \mathbb{N}$. Suppose furthermore that $C$ is *output-oblivious*: That is, the output species of $C$ is not a reactant of any reaction of $C$. This condition ensures that outputs produced by $C$ can be consumed as inputs by a downstream CRN, without affecting the correctness of $C$. Then if the output species of $C$ is the input species of $C'$, and there is no other species common to $C$ and $C'$, the CRN $C \cup C'$ computes $f' \circ f$.

More generally, suppose that CRNs $C_1, C_2, \ldots, C_{d'}$ stably compute the functions $f_1$, $f_2$, …, $f_{d'} : \mathbb{N}^d \to \mathbb{N}$, and CRN $C'$ stably computes $f' : \mathbb{N}^{d'} \to \mathbb{N}$. Suppose also that the $C_i$ are output-oblivious, the output of $C_i$ is the $i$th input to $C'$ and there is no other species common to the CRNs. Then $C_1 \cup C_2 \ldots C_{d'} \cup C'$ computes $f'(f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_{d'}(\mathbf{x}))$. For example, combining the reactions of the CRN of Figure 1 parts (a), (b) and (c) results in a CRN to compute the function $f'(x_1, x_2) = \min\{f(x_1, x_2), 2x_2\}$.

If a function $f$ is stably computable by an output-oblivious CRN with a leader, we say that $f$ is *obliviously-computable*. Obliviously-computable functions must be nondecreasing, because a CRN on input $\mathbf{x} + \mathbf{x}'$ can produce $f(\mathbf{x})$ Y's (by ignoring inputs representing $\mathbf{x}'$), and if $Y$'s are never consumed, the stable output $f(\mathbf{x} + \mathbf{x}')$ that is eventually produced must then be at least $f(\mathbf{x})$. However, not all nondecreasing semilinear functions are obliviously-computable, the max function being an interesting counterexample. Chugg et al. [9] characterized the

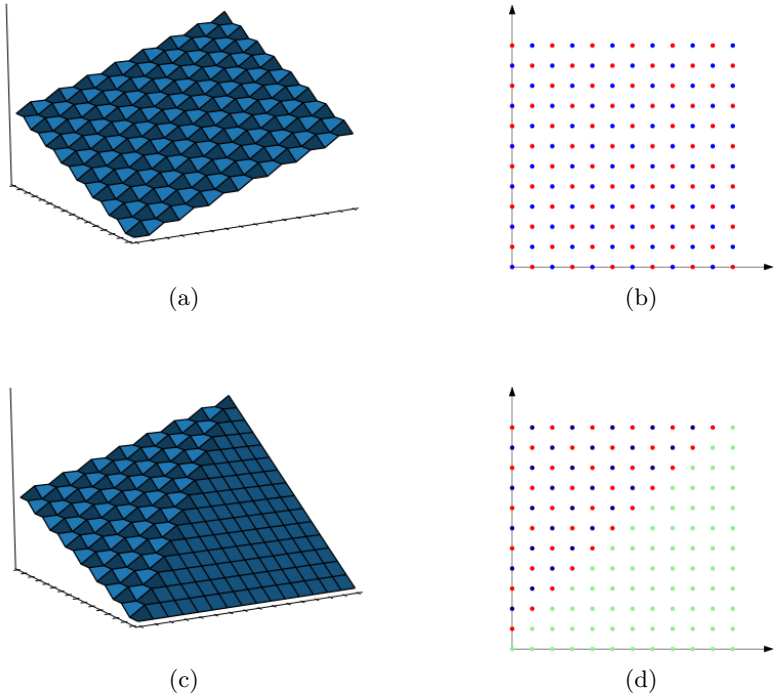



**Figure 2** Illustrations of quilt-affine functions with domain $\mathbb{N}^2$. (a) The function $h(\mathbf{x}) = x_1 + x_2 - ((x_1 + x_2) \mod 2)$. (b) Domains of the function $h$ of part (a). $h(\mathbf{x}) = x_1 + x_2$ on the domain $\text{Dom}_1 = \{\mathbf{x} \in \mathbb{N}^2 \mid x_1 + x_2 = 0 \pmod 2\}$, shown in blue. $\text{Dom}_1$ is linear since it equals $\{\alpha_1(2,0) + \alpha_2(0,2) + \alpha_3(1,1) + (0,0) \mid \alpha_1, \alpha_2, \alpha_3 \in \mathbb{N}\}$. Also, $h(\mathbf{x}) = x_1 + x_2 - 1$ on the domain $\text{Dom}_2 = \{\mathbf{x} \in \mathbb{N}^2 \mid x_1 + x_2 = 1 \pmod 2\}$, shown in red. The domain $\text{Dom}_2$ is the union of two linear sets, namely $\{\alpha_1(2,0) + \alpha_2(0,2) + \alpha_3(1,1) + (0,1) \mid \alpha_1, \alpha_2, \alpha_3 \in \mathbb{N}\}$ and $\{\alpha_1(2,0) + \alpha_2(0,2) + \alpha_3(1,1) + (1,0) \mid \alpha_1, \alpha_2, \alpha_3 \in \mathbb{N}\}$. (c) The function $f(\mathbf{x}) = \min\{h(x_1, x_2), 2x_2\}$. (d) Domains of the function $f$ of part (c). $f(\mathbf{x}) = 2x_2$ on the domain $\text{Dom}_3 = \{\mathbf{x} \in \mathbb{N}^2 \mid x_2 + 1 \leq x_1\}$, shown in green. $\text{Dom}_3$ is linear since it equals $\{\alpha_1(1,0) + \alpha_2(1,1) + (1,0) \mid \alpha_1, \alpha_2 \in \mathbb{N}\}$. Also, $f(x_1, x_2) = h(x_1, x_2)$ on the semilinear domains $\text{Dom}'_1 = \text{Dom}_1 \cap \{\mathbf{x} \in \mathbb{N}^2 \mid x_1 \leq x_2\}$ and $\text{Dom}'_2 = \text{Dom}_2 \cap \{\mathbf{x} \in \mathbb{N}^2 \mid x_1 \leq x_2\}$, shown in red and blue.

subclass of obliviously-computable functions with two inputs, i.e., functions $f : \mathbb{N}^2 \to \mathbb{N}$. Severson et al. [16] gave a general characterization of obliviously-computable functions $f : \mathbb{N}^d \to \mathbb{N}$, for any $d$; such functions are eventually the min of *quilt-affine* functions, defined as nondecreasing linear functions with a periodic intercept, see Figure 2. See Section 2 for formal definitions of quilt-affine and obliviously-computable functions.

The results of Chugg et al. and Severson et al. described so far concern discrete, output-oblivious CRNs with leaders. What about leaderless CRNs? Output-oblivious functions computed by a leaderless CRN $C$ must be superadditive, i.e., $f(\mathbf{x}) + f(\mathbf{x}') \geq f(\mathbf{x} + \mathbf{x}')$. This is because on input $\mathbf{x} + \mathbf{x}'$, reactions of a leaderless CRN could be used to independently compute both $f(\mathbf{x})$ and $f(\mathbf{x}')$, resulting in $f(\mathbf{x}) + f(\mathbf{x}')$ output molecules, so this quantity must be less than or equal to the eventual stable output, namely $f(\mathbf{x} + \mathbf{x}')$. This raises the question: Is the class of functions $f : \mathbb{N}^d \to \mathbb{N}$ that can be stably computed by leaderless output-oblivious CRNs exactly the superadditive obliviously-computable functions? Severson et al. showed that this is indeed the case when $d = 1$, but the more general case was left as an open problem. In this paper we show that the answer is "yes" for all $d$:

▶ **Theorem 1.** *Functions that are stably computable by leaderless output-oblivious CRNs are exactly the superadditive obliviously-computable functions.*

Our proof of Theorem 1 has two parts. First, building on the previous work of Severson et al. and Chugg et al., we provide in Claim 5 a new characterization of superadditive, obliviously-computable functions as the minimum of superadditive quilt-affine functions on *well-ordered* domains, which we define in Section 2. Then in Claim 14 we construct a leaderless, output-oblivious CRN for superadditive, obliviously-computable functions, using the well-ordered domain representation.

Our result has strong parallels with that of Chalk et al. [6] who studied composability of function-computing CRNs for the *continuous* (also called mass-action) CRN model. In this model, real-valued species concentrations, rather than discrete species counts, evolve over time, according to a finite set of reactions. Earlier, Chen et al. [8] showed that continuous CRNs can stably (i.e., regardless of actual reaction rates) compute positive-continuous, piecewise rational linear functions. Chalk et al. showed that such functions are obliviously-computable by continuous CRNs if and only if they are superadditive. However, the proof techniques for the discrete and continuous CRN models are quite different.

## 2 The CRN Model and Obliviously-Computable Functions

Following a summary of useful notation, we describe Chemical Reaction Networks (CRNs), stable CRN function computation, and output-oblivious function computation. We then describe the result of Severson et al. [16] that characterizes the class of functions that are stably computable by output-oblivious CRNs with a leader, i.e., obliviously-computable functions, in terms of quilt-affine functions. Finally, we provide a new, alternative characterization of obliviously-computable functions that is useful for our main results.

### 2.1 Notation

We use $\mathbb{N}$ to denote the set of nonnegative integers, $\mathbb{N}_+$ the positive integers, $\mathbb{Z}$ the integers, $\mathbb{Q}$ the rationals, and $\mathbb{Q}_{\geq 0}$ the nonnegative rationals. Where $d$ is understood, we use boldface to represent $d$-dimensional vectors $\mathbf{x} \in \mathbb{N}^d$, and $x_i$ to denote the $i$th component of $\mathbf{x}$, $1 \leq i \leq d$. We write $\mathbf{x} \leq \mathbf{x}'$ to denote that $x_i \leq x'_i$, for all $i, 1 \leq i \leq d$, and $\mathbf{x} < \mathbf{x}'$ to denote that $\mathbf{x} \leq \mathbf{x}'$ and for some $i, 1 \leq i \leq d$, $x_i < x'_i$. For $1 \leq i \leq d$, we let $\mathbf{e}_i$ denote the $d$-dimensional unit vector $(e_{i1}, \ldots, e_{id})$ in which all components are zero except that $e_{ii} = 1$. We denote the $d$-dimensional vector of all zero's by $\mathbf{0}$.

For $d, p \in \mathbb{N}_+$, $\mathbb{Z}^d/p\mathbb{Z}^d$ denotes the additive group of $\mathbb{Z}^d$ modulo $p$. Each element of $\mathbb{Z}^d/p\mathbb{Z}^d$ is a congruence class of the form $\{\mathbf{n} + p\mathbf{z} \mid \mathbf{z} \in \mathbb{Z}^d\}$ for some $\mathbf{n} \in \mathbb{N}^d$, and we denote this set by $\overline{\mathbf{n}}$.

### 2.2 Chemical Reaction Networks and Stable Function Computation

A discrete Chemical Reaction Network (CRNs) is specified as a finite set $\mathcal{Z} = \{Z_1, \ldots, Z_m\}$ of *species*, plus a finite set of $\mathcal{R}$ of *reactions* $(\mathbf{s}, \mathbf{t}) = ((s_1, \ldots, s_m), (t_1, \ldots, t_m)) \in \mathbb{N}^{\mathcal{Z}} \times \mathbb{N}^{\mathcal{Z}}$ of the form

$$\sum_{k:s_k>0} s_k Z_k \to \sum_{k:t_k>0} t_k Z_k,$$

where for at least one $j$, $s_j \neq t_j$. The species $Z_k$ with $s_k > 0$ are the *reactants*, which are *consumed*, while those with $t_k > 0$ are the *products*. (A species may be both a reactant and product of the same reaction). A *configuration* $\mathbf{c} \in \mathbb{N}^m$ describes counts of species in $\mathcal{Z}$, and

$\mathbf{c}(Z)$ denotes the count of species $Z \in \mathcal{Z}$. Reaction $(\mathbf{s}, \mathbf{t})$ is *applicable* to configuration $\mathbf{c}$ if $\mathbf{s} \leq \mathbf{c}$, i.e., sufficiently many copies of each reactant are present. Application of the reaction to $\mathbf{c}$ results in the configuration $\mathbf{c}' = \mathbf{c} - \mathbf{s} + \mathbf{t}$, and we write $\mathbf{c} \to \mathbf{c}'$. If $\mathbf{c}_0 \to \mathbf{c}_1 \to \ldots \mathbf{c}_t$ then we say that $\mathbf{c}_t$ is *reachable* from $\mathbf{c}_0$ and call $\mathbf{c}_0 \to \mathbf{c}_1 \to \ldots \mathbf{c}_t$ an *execution* of the CRN.

A CRN $\mathcal{C}$ to stably compute a function $f : \mathbb{N}^d \to \mathbb{N}$ has designated input species, say $X_1, \ldots, X_d$, a designated output species, say $Y$, and may or may not have a designated leader species, $L \in \mathcal{Z} \setminus \mathcal{I}$. Leaderless function computation on input $\mathbf{x} \in \mathbb{N}^d$ starts from a valid initial configuration $\mathbf{c}_0 = \mathbf{c}_0(\mathbf{x})$, where $\mathbf{c}_0(X_i) = x_i$ for $1 \leq i \leq d$, and the count of any other species is 0. CRN computation with a leader differs only in that the initial count of the leader species $L$ is 1, i.e., $\mathbf{c}_0(L) = 1$. We say that $\mathcal{C}$ *stably computes* $f$ if for every valid initial configuration $\mathbf{c}_0 = \mathbf{c}_0(\mathbf{x})$ for some $\mathbf{x}$, and for every configuration $\mathbf{c}$ reachable from $\mathbf{c}_0$, there exists a stable configuration $\mathbf{c}'$ reachable from $\mathbf{c}$ such that $f(\mathbf{x}) = \mathbf{c}'(Y)$. Here, $\mathbf{c}'$ is *stable* if for every $\mathbf{c}'' \in \mathbb{N}^m$ reachable from $\mathbf{c}'$, $\mathbf{c}'(Y) = \mathbf{c}''(Y)$. That is, once configuration $\mathbf{c}'$ is reached, the count of the output species does not change. Stable computation with a leader is defined in the same way, except that in the initial configuration the count of a designated leader species $L$ is 1.

Chen et al. [7] (building on related work of Angluin et al. [2, 4] on predicate computation by population protocols) showed that exactly the *semilinear* functions are stably computable by CRNs. A semilinear function is the union of partial affine functions on linear domains. A domain $E \subset \mathbb{N}^d$ is *linear* if $E = \{\sum_{\mathbf{z} \in F} \alpha_{\mathbf{z}} \mathbf{z} + \mathbf{o} : \alpha_{\mathbf{z}} \in \mathbb{N}\}$ for some finite set $F \subset \mathbb{N}^d$ and $\mathbf{o} \in \mathbb{N}^d$. Thus, if $E_1, E_2, \ldots, E_m$ are linear sets, $\cup_{i=1}^m E_i = \mathbb{N}^d$, and for $1 \leq i \leq m$ $f_i : E_i \to \mathbb{N}$ is a partial affine function, then the function $f : \mathbb{N}^d \to \mathbb{N}$ where $f(\mathbf{x}) = f_i(\mathbf{x})$ if $\mathbf{x} \in E_i$ is semilinear. Figure 2 shows examples of linear sets and semilinear functions, illustrating show the union of linear sets can be defined using $\geq$ or mod. Doty and Hajiaghayi [11] showed that leaderless CRNs also stably compute the semilinear functions.

## 2.3 Obliviously-Computable Functions As Quilt-Affine Functions

A CRN $\mathcal{C}$ is *output-oblivious* if no reaction consumes the output species. A function $f$ is *obliviously-computable* if some output-oblivious CRN with a leader stably computes $f$. A subclass of the obliviously-computable functions are the *leaderless obliviously-computable* functions, that can be stably computed by leaderless output-oblivious CRNs.

Severson et al. [16] defined a *quilt-affine function* $h : \mathbb{N}^d \to \mathbb{Z}$ to be a nondecreasing function that is the sum of a rational linear function and a periodic function. That is, for some $\nabla_h \in \mathbb{Q}^d_{\geq 0}$, called the gradient of $h$, some $p \in \mathbb{N}^+$, called the period, and some $B : \mathbb{Z}^d / p\mathbb{Z}^d \to \mathbb{Q}$, called the periodic intercept,

$$h(\mathbf{x}) = \nabla_h \cdot \mathbf{x} + B(\overline{\mathbf{x}}).$$

For example, the 2D function $h(\mathbf{x}) = x_1 + x_2 - ((x_1 + x_2) \mod 2)$ of Figure 2 is quilt-affine, since it can be written as $h(\mathbf{x}) = (1, 1) \cdot (x_1, x_2) + B(\overline{\mathbf{x}})$, where $B(0, 0) = B(1, 1) = 0$ and $B(0, 1) = B(1, 0) = -1$. Severson et al. [16] proved the following result.

▶ **Theorem 2.** (Severson et al. [16]) *A function $f : \mathbb{N}^d \to \mathbb{N}$ is obliviously-computable if and only if it satisfies the following three properties:*

- **(i)** *$f$ is* nondecreasing*, i.e., $f(\mathbf{x}) \leq f(\mathbf{x}')$ for all $\mathbf{x} \leq \mathbf{x}'$.*
- **(ii)** *There exist (nondecreasing) quilt-affine functions $h_1, \ldots, h_m : \mathbb{N}^d \to \mathbb{N}$ and $\mathbf{k}_f \in \mathbb{N}^d$ such that for all $\mathbf{x} \geq \mathbf{k}_f$, $f(\mathbf{x}) = \min_i\{h_i(\mathbf{x})\}$.*

**(iii)** *All fixed-input restrictions of $f$ are obliviously-computable. Here, a fixed-input restriction of $f$ is a function on $d-1$ inputs defined as*

$$f_{[x_i \to j]}(\mathbf{x}) = f(x_1, x_2, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_d),$$

*for some $1 \leq i \leq d$ and $j \in \mathbb{N}$.*

## 2.4 Obliviously-Computable Functions As Well-Ordered Quilt-Affine Functions

Here we adapt Severson et al.'s result to obtain a slightly different characterization of obliviously-computable functions, as the union of *partial* quilt-affine functions over *well-ordered* domains sets. This result lays the foundation for the rest of the paper. Claim 4 in Section 3 demonstrates that these partial quilt-affine functions may be assumed to be superadditive which, coupled with Theorem 2, implicitly proves one direction of Theorem 1. Additionally, the well-ordered domain sets will be further refined by the CRN construction in Section 4, enabling a quilt-affine function to be expressed simply as a piecewise affine function. A *partial quilt-affine function* is simply a quilt-affine function that is defined only over a subset of $\mathbb{N}^d$.

Next we define well-ordered domain sets. Let $\mathbf{w} \in \mathbb{N}^d$ be fixed and let $\mathbf{0} \leq \mathbf{o} \leq \mathbf{w}$. Let

$$\mathrm{Dom}_{\mathbf{o}} \ (= \mathrm{Dom}_{\mathbf{o},\mathbf{w}}) = \{\mathbf{x} \in \mathbb{N}^d \mid \ \mathbf{x} \geq \mathbf{o} \text{ and } x_i = o_i \text{ if } o_i < w_i\}. \tag{1}$$

The sets $\mathrm{Dom}_{\mathbf{o}}$ for $\mathbf{0} \leq \mathbf{o} \leq \mathbf{w}$, are disjoint and their union is $\mathbb{N}^d$. We call the set of sets $\{\mathrm{Dom}_{\mathbf{o}} \mid \mathbf{0} \leq \mathbf{o} \leq \mathbf{w}\}$ a *well-ordered domain set*, and we denote this set by $\mathcal{WO}_{\mathbf{w}}$. The sets are ordered in the sense that if $\mathbf{x} \in \mathrm{Dom}_{\mathbf{o}}$, $\mathbf{x}' \in \mathrm{Dom}_{\mathbf{o}'}$ and $\mathbf{x} \leq \mathbf{x}'$ then $\mathbf{o} \leq \mathbf{o}'$. Figure 3 (b) shows a well-ordered domain set for $\mathbb{N}^2$ where $\mathbf{w} = (4, 4)$. We will later use the following property of well-ordered domains:

▶ **Lemma 3.** *Let $\mathrm{Dom}_{\boldsymbol{o}}$ and $\mathrm{Dom}_{\boldsymbol{o}'}$ be domains of a well-ordered set defined by $\boldsymbol{w}$, and let $\mathrm{Dom}_{\boldsymbol{o}''}$ be the domain containing $\boldsymbol{o} + \boldsymbol{o}'$. Then for any $\mathbf{x} \in \mathrm{Dom}_{\boldsymbol{o}}$, and $\mathbf{x}' \in \mathrm{Dom}_{\boldsymbol{o}'}$, $\mathbf{x}'' = \mathbf{x} + \mathbf{x}' \in \mathrm{Dom}_{\boldsymbol{o}''}$.*

**Proof.** Since $\mathbf{x} \in \mathrm{Dom}_{\mathbf{o}}$ and $\mathbf{x}' \in \mathrm{Dom}_{\mathbf{o}'}$ we have that $\mathbf{x} \geq \mathbf{o}$ and $\mathbf{x}' \geq \mathbf{o}'$. Additionally, since $\mathbf{o} + \mathbf{o}' \in \mathrm{Dom}_{\mathbf{o}''}$, we have that $\mathbf{x} + \mathbf{x}' \geq \mathbf{o} + \mathbf{o}' \geq \mathbf{o}''$. So $\mathbf{x}''$ satisfies the first condition of membership in $\mathrm{Dom}_{\mathbf{o}''}$.

It remains to show that if $o''_i < w_i$, then $x_i = o''_i$. So suppose that $o''_i < w_i$. Then it must also be the case that $o_i < w_i$ and $o'_i < w_i$, that $o''_i = o_i + o'_i$, and that $x_i = o_i$ and $x'_i = o'_i$. The result follows. ◀

A *well-ordered quilt-affine function* is the finite union of partial quilt-affine functions, each of which is defined on a domain of a well-ordered domain set.

▷ Claim 4. Any obliviously-computable function is the minimum of a finite number of nondecreasing, well-ordered quilt-affine functions.

Proof. First, from the characterization of obliviously-computable functions of Severson et al. [16] given in Theorem 2 above, we identify a finite set of partial quilt-affine functions $\mathcal{H}$, as follows. We include in $\mathcal{H}$ the functions $h_1, h_2, \ldots, h_m$, each with domain $\mathrm{Dom}_{h_i} = \{\mathbf{x} \in \mathbb{N}^d \mid \mathbf{x} \geq \mathbf{k}_f\}$, described in property (ii) of Theorem 2.3. Then we recursively augment $\mathcal{H}$ by considering each of the fixed-input restrictions $f_{[x_i \to j]}$ of $f$ of part (iii) of the definition, for each choice of $j < k_{f,i}$, and adding the functions corresponding to $f_{[x_i \to j]}$ from property (ii)

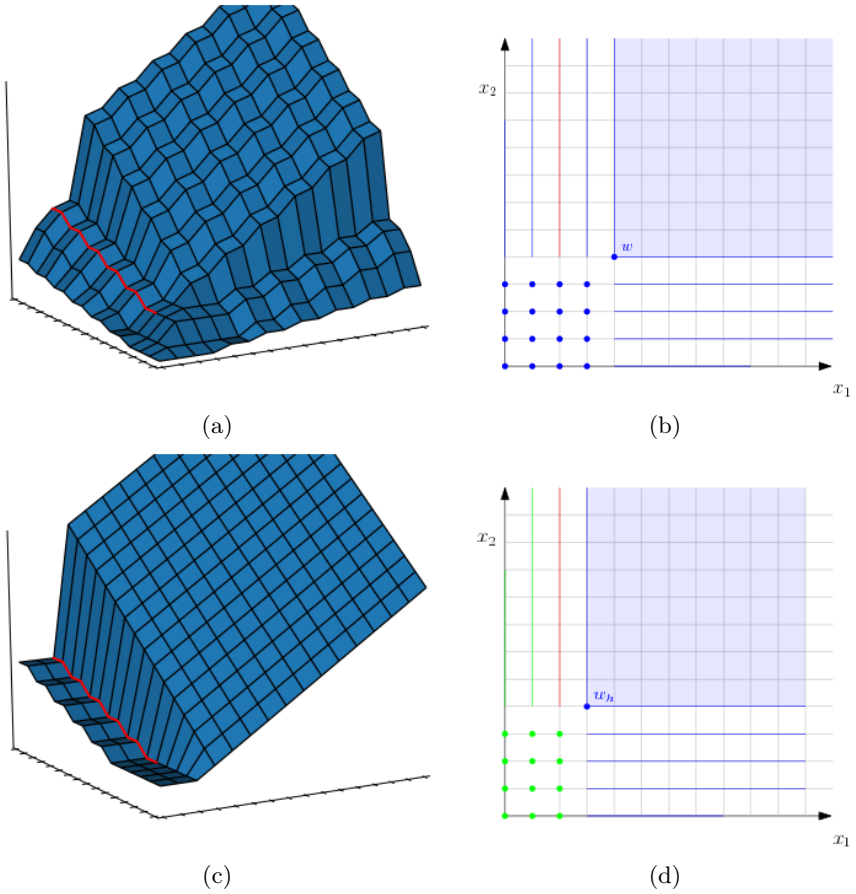



**Figure 3** (a) A well-ordered, superadditive function $f$ with domain set $\mathcal{WO}_{\mathbf{w}}$ for $\mathbf{w} = (4, 4)$. Here, $f(x, y) = h(x, y) = y - (y \mod 2)$ on the red line and $f = 2y - (y \mod 2) + 2x - (x \mod 2) - 8$ on the large 2 dimensional area. The red line corresponds to the domain $\text{Dom}_{(3,4)}$. (b) The well-ordered domain set for the function $f$ of part (a), with $\mathbf{w} = (4, 4)$. There is one 2D domain, eight 1D domains, and twelve 0D domains, i.e., points. (c) The function $h_{\text{WO}}$ obtained from $h$ via the construction of Claim 4. (d) The three domains $\text{Dom}_h$ (in red), $\text{Dom}_{\text{big}}$ (in blue) and $\text{Dom}_{\text{small}}$ (in green), for the function $h$ of part (c). Here, $w_h = (3, 4)$.

of Theorem 2. There are $d$ levels of recursion; the functions that are recursively added to $\mathcal{H}$ have at least one and up to $d$ fixed inputs, and the remaining (non-fixed) inputs are lower bounded by some constant. Thus, for each function $h$ added to $\mathcal{H}$, the domain of $h$ has the form

$$\text{Dom}_h = \{\mathbf{x} \in \mathbb{N}^d \mid x_i = k_{h,i} \text{ if } i \in D_h \text{ and } x_i \geq k_{h,i} \text{ otherwise}\}, \tag{2}$$

for some $\mathbf{k}_h \in \mathbb{N}^d$ and $D_h \subseteq [1, \ldots, d]$. We can assume without loss of generality that all functions $h \in \mathcal{H}$ have the same period, since we can always take the least common multiple of the periods and redefine each $h$ with respect to this least common multiple.

For each such $h \in \mathcal{H}$ we will construct a nondecreasing, well-ordered quilt-affine function $h_{\text{WO}} : \mathbb{N}^d \to \mathbb{N}$ such that $h_{\text{WO}}(\mathbf{x}) = h(\mathbf{x})$ for all $\mathbf{x} \in \text{Dom}_h$, and also $f(\mathbf{x}) \leq h_{\text{WO}}(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^d - \text{Dom}_h$. Then $f = \min_{\{h \in \mathcal{H}\}} h_{\text{WO}}$, and the claim follows.

We'll use the following notation when describing $h_{\text{WO}}$. Let $\nabla_h = (\lambda_{h,1}, \lambda_{h,2}, \ldots, \lambda_{h,k}) \in \mathbb{Q}^d$ be the gradient of $h$, let $\lambda_{\max} = \lceil \max_{h \in \mathcal{H}, 1 \leq i \leq d} \{\lambda_{h,i}\} \rceil$ and let

$$\nabla_{\max} = (\lambda_{\max}, \ldots, \lambda_{\max}).$$

Similarly, let $B_h$ be the periodic intercept of $h$ and let

$$B_{\max} = \left\lceil \max_{h\in\mathcal{H},\mathbf{x}\in\mathbb{N}^d} \{B_h(\overline{\mathbf{x}} \mod p)\} \right\rceil.$$

We partition $\mathbb{N}^d$ into three domains:

- $\mathrm{Dom}_h$, defined in Equation (2), where $\mathbf{k}_h \in \mathbb{N}^d$ and $D_h \subseteq [1,\ldots,d]$.
- $\mathrm{Dom}_{\mathrm{small}} = \{\mathbf{x} \in \mathbb{N}^d \mid \ x_i \le k_{h,i},\ 1 \le i \le d\} - \mathrm{Dom}_h$;
- $\mathrm{Dom}_{\mathrm{big}} = \mathbb{N}^d - \mathrm{Dom}_{\mathrm{small}} - \mathrm{Dom}_h$.

Also, for $\mathbf{x} \in \mathbb{N}^d$, we let

$$\mathrm{pr}(\mathbf{x}) = (\mathrm{pr}(x_1), \mathrm{pr}(x_2), \ldots, \mathrm{pr}(x_d)),$$

where $\mathrm{pr}(x_i) = k_i$ if $x_i \le k_i$ and $\mathrm{pr}(x_i) = x_i$ otherwise. Note that for $\mathbf{x} \in \mathrm{Dom}_{\mathrm{small}}$ we have $\mathrm{pr}(\mathbf{x}) \in \mathrm{Dom}_h$. We can now define $h_{\mathrm{WO}}$ as follows.

$$h_{\mathrm{WO}}(\mathbf{x}) = \begin{cases} h(\mathbf{x}), & \text{for all } \mathbf{x} \in \mathrm{Dom}_h, \\ \nabla_{\max}\cdot\mathbf{x} + B_{\max}, & \text{for all } \mathbf{x} \in \mathrm{Dom}_{\mathrm{big}}, \text{ and} \\ h(\mathrm{pr}(\mathbf{x})), & \text{for all } \mathbf{x} \in \mathrm{Dom}_{\mathrm{small}}. \end{cases}$$

Figure 3 shows an example of the construction of $h_{\mathrm{WO}}$ from $h$.

First we show that $f(\mathbf{x}) \le h_{\mathrm{WO}}(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^d$. There are three cases, depending on whether $\mathbf{x}$ is in $\mathrm{Dom}_h$, $\mathrm{Dom}_{\mathrm{big}}$, or $\mathrm{Dom}_{\mathrm{small}}$. (1) By definition, for $\mathbf{x} \in \mathrm{Dom}_h$ we have $f(\mathbf{x}) \le h(\mathbf{x}) = h_{\mathrm{WO}}(\mathbf{x})$. (2) For $\mathbf{x} \in \mathrm{Dom}_{\mathrm{small}}$, we know that $\mathbf{x} \le \mathrm{pr}(\mathbf{x})$ and so $f(\mathbf{x}) \le f(\mathrm{pr}(\mathbf{x}))$. Also, $\mathrm{pr}(\mathbf{x}) \in \mathrm{Dom}_h$, and so we know from case (1) that $f(\mathrm{pr}(\mathbf{x})) \le h_{\mathrm{WO}}(\mathrm{pr}(\mathbf{x}))$. (3) For $\mathbf{x} \in \mathrm{Dom}_{\mathrm{big}}$ we know that $f(\mathbf{x}) = h'(\mathbf{x})$ for some $h' \in \mathcal{H}$, and also by our choice of $\nabla_{\max}$ and $B_{\max}$ we have that $h'(\mathbf{x}) \le \nabla_{\max}\cdot\mathbf{x} + B_{\max} = h_{\mathrm{WO}}(\mathbf{x})$. Putting these together, we have that

$$f(\mathbf{x}) = h'(\mathbf{x}) \le h_{\mathrm{WO}}(\mathbf{x}).$$

Next we show that $h_{\mathrm{WO}}$ is non-decreasing, that is, $h_{\mathrm{WO}}(\mathbf{x}) \le h_{\mathrm{WO}}(\mathbf{x}')$ for all $\mathbf{x}, \mathbf{x}' \in \mathbb{N}^d$ with $\mathbf{x} \le \mathbf{x}'$. We consider the possible cases for the domains of $\mathbf{x}$ and $\mathbf{x}'$:

1. $\mathbf{x} \in \mathrm{Dom}_h$ and $\mathbf{x}' \in \mathrm{Dom}_h$. Then $h_{\mathrm{WO}}(\mathbf{x}) \le h_{\mathrm{WO}}(\mathbf{x}')$ since $h_{\mathrm{WO}} = h$ on $\mathrm{Dom}_h$ and $h$ is nondecreasing.
2. $\mathbf{x} \in \mathrm{Dom}_{\mathrm{big}}$ and $\mathbf{x}' \in \mathrm{Dom}_{\mathrm{big}}$. Then
$$h_{\mathrm{WO}}(\mathbf{x}) = \nabla_{\max}(\mathbf{x}) + B_{\max} \le \nabla_{\max}(\mathbf{x}') + B_{\max} = h_{\mathrm{WO}}(\mathbf{x}').$$
3. $\mathbf{x} \in \mathrm{Dom}_h$ and $\mathbf{x}' \in \mathrm{Dom}_{\mathrm{big}}$. Then
$$h_{\mathrm{WO}}(\mathbf{x}) = \nabla_h(\mathbf{x}) + B(\overline{\mathbf{x}}) \le \nabla_{\max}(\mathbf{x}) + B_{\max} \le \nabla_{\max}(\mathbf{x}') + B_{\max} = h_{\mathrm{WO}}(\mathbf{x}').$$
4. $\mathbf{x} \in \mathrm{Dom}_{\mathrm{small}}$ and $\mathbf{x}' \in \mathrm{Dom}_{\mathrm{small}}$. Then $\mathrm{pr}(\mathbf{x}) \le \mathrm{pr}(\mathbf{x}')$ and both $\mathrm{pr}(\mathbf{x})$ and $\mathrm{pr}(\mathbf{x}')$ are in $\mathrm{Dom}_h$, so
$$h_{\mathrm{WO}}(\mathbf{x}) = h_{\mathrm{WO}}(\mathrm{pr}(\mathbf{x})) \le h_{\mathrm{WO}}(\mathrm{pr}(\mathbf{x}')) = h_{\mathrm{WO}}(\mathbf{x}'),$$
where the inequality holds because of case 1.
5. $\mathbf{x} \in \mathrm{Dom}_{\mathrm{small}}$ and $\mathbf{x}' \in \mathrm{Dom}_h$. Then $\mathrm{pr}(\mathbf{x}) \in \mathrm{Dom}_h$ and $\mathrm{pr}(\mathbf{x}) \le \mathbf{x}'$, so
$$h_{\mathrm{WO}}(\mathbf{x}) = h_{\mathrm{WO}}(\mathrm{pr}(\mathbf{x})) = h(\mathrm{pr}(\mathbf{x})) \le h(\mathbf{x}') = h_{\mathrm{WO}}(\mathbf{x}').$$
6. $\mathbf{x} \in \mathrm{Dom}_{\mathrm{small}}$ and $\mathbf{x}' \in \mathrm{Dom}_{\mathrm{big}}$. Then
$$h_{\mathrm{WO}}(\mathbf{x}) = h_{\mathrm{WO}}(\mathrm{pr}(\mathbf{x})) \le \nabla_{\max}(\mathbf{x}) + B_{\max} \le \nabla_{\max}(\mathbf{x}') + B_{\max} = h_{\mathrm{WO}}(\mathbf{x}').$$

Finally, we show that $h_{\text{WO}}$ is a well ordered quilt-affine function with offset $\mathbf{w}_h$, where we define $\mathbf{w}_h \in \mathbb{N}^d$ as $w_{h,i} = k_{h,i}$ if $i \in D_h$ and $w_{h,i} = k_{h,i} + 1$ otherwise. Consider any $\mathbf{o} \leq \mathbf{w}_h$. We need to show that $h_{\text{WO}}$ is quilt-affine on the domain $\text{Dom}_{\mathbf{o}}$ (defined in Equation (1)). There are three cases:

1. If $\mathbf{o} = \mathbf{k}_h$ ($\leq \mathbf{w}_h$) then $\text{Dom}_{\mathbf{o}} = \text{Dom}_h$. By construction, $h_{\text{WO}} = h$ on $\text{Dom}_h$, and $h$ is quilt-affine.
2. If $\mathbf{o} \leq \mathbf{k}_h$ but $\mathbf{o} \neq \mathbf{k}_h$, then $\mathbf{o}$ is in $\text{Dom}_{\text{small}}$. Let $\mathbf{o} = \mathbf{k}_h - \mathbf{k}'_h$, where $\mathbf{k}'_h \in \mathbb{N}^d$. For each $\mathbf{x} \in \text{Dom}_{\mathbf{o}}$ we have $\mathbf{x} \in \text{Dom}_{\text{small}}$, and so also $\text{pr}(\mathbf{x}) = \mathbf{x} + \mathbf{k}'_h \in \text{Dom}_h$. Therefore,

$$\begin{aligned} h_{\text{WO}}(\mathbf{x}) &= h_{\text{WO}}(\text{pr}(\mathbf{x})) \\ &= h(\text{pr}(\mathbf{x})) \\ &= h(\mathbf{x} + \mathbf{k}'_h) \\ &= \nabla_h \cdot (\mathbf{x} + \mathbf{k}'_h) + B(\overline{\mathbf{x} + \mathbf{k}'_h}) \\ &= \nabla_h \cdot \mathbf{x} + \nabla_h \cdot \mathbf{k}'_h + B(\overline{\mathbf{x} + \mathbf{k}'_h}) \\ &= \nabla_h \cdot \mathbf{x} + B'(\overline{\mathbf{x}}), \end{aligned}$$

   where $B'(\overline{\mathbf{x}}) = \nabla_h \cdot \mathbf{k}'_h + B(\overline{\mathbf{x} + \mathbf{k}'_h})$. Thus $h_{\text{WO}}$ is quilt-affine.
3. If $\mathbf{o} \in \text{Dom}_{\text{big}}$, then since all $\mathbf{x} \geq \mathbf{o}$ are in $\text{Dom}_{\text{big}}$, the function $h_{\text{WO}}$ on $\text{Dom}_{\mathbf{o}}$ is affine and therefore quilt-affine with period $p$. ◁

## 3 Superadditive, Obliviously-Computable Functions as Quilt-Affine Functions

In Claim 4, we showed that an obliviously-computable function $f$ can be represented as the min of finitely many well-ordered quilt-affine functions. However, even if $f$ is superadditive, the quilt-affine functions constructed in Claim 4 may not be superadditive. In this section we strengthen that result to show in Claim 5 that if $f$ is superadditive, then $f$ is the min of finitely many *superadditive* well-ordered quilt-affine functions, thereby proving the first half of our main result, Theorem 1.

▷ Claim 5. Any superadditive, obliviously-computable function is the minimum of a finite number of superadditive, well-ordered quilt-affine functions.

Proof. Let $f : \mathbb{N}^d \to \mathbb{N}$ be a superadditive, obliviously-computable function. From Claim 4, we know that $f = \min\{h_{\text{WO}}\}$, where each of the finitely many $h_{\text{WO}} : \mathbb{N}^d \to \mathbb{N}$ is a non-decreasing, well-ordered quilt-affine function. Let $p$ be the period of the functions $f$ and the $h_{\text{WO}}$'s. Since the $h_{\text{WO}}$'s may not be superadditive, we construct a superadditive, well-ordered quilt-affine function $h_{\text{S}}$ from each $h_{\text{WO}}$, such that $f = \min\{h_{\text{S}}\}$.

With respect to some fixed $h_{\text{WO}}$ and its well-ordered domain representation, say $\mathcal{WO}_{\mathbf{w}}$, we first partition the well-ordered domains into new types of domains that we will call *patches*. Then we define a superadditive function $h_{\text{S}}$ as the union of partial affine functions on patches, such that $f(\mathbf{x}) \leq h_{\text{S}}(\mathbf{x}) \leq h_{\text{WO}}(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^d$. Finally we further partition the patches into well-ordered domains to show that $h_{\text{S}}$ is well-ordered quilt-affine, completing the proof of the claim.

We define a patch as follows. Let $\overline{\mathbf{n}}$ be a congruence class mod $p$, i.e., $\overline{\mathbf{n}} = \{\mathbf{n} + p\mathbf{z} \mid \mathbf{z} \in \mathbb{Z}^d\}$, where $\mathbf{n} \in \mathbb{N}^d$. The patch defined by a *corner* $\mathbf{q} \in \mathbb{N}^d \cap \overline{\mathbf{n}}$, a finite *set of excluding points* $Q \subset \mathbb{N}^d$, and $\overline{\mathbf{n}}$ is

$$P(\mathbf{q}, Q, \overline{\mathbf{n}}) = \{\mathbf{x} \in \mathbb{N}^d \cap \overline{\mathbf{n}} \mid \mathbf{q} \leq \mathbf{x} \text{ and } \mathbf{x} \not\geq \mathbf{q}', \forall \mathbf{q}' \in Q\}.$$

Figure 6 of the appendix illustrates a patch, and our overall transformation from $h_{\text{WO}}$ to $h_{\text{S}}$.

For each domain Dom of the well-ordered representation of $h_{\text{WO}}$ and each congruence class $\overline{\mathbf{n}}$ in $\mathbb{Z}^d/p\mathbb{Z}^d$, we cover $\text{Dom} \cap \overline{\mathbf{n}}$ with a finite number patches as follows. Initially, let the set $Q$ of excluding points be the set of offsets of domains of $h_{\text{WO}}$ that are greater than the offset of Dom. This ensures that only points in Dom are included in the constructed patches. While not all of $\text{Dom} \cap \overline{\mathbf{n}}$ is covered, select from the uncovered points the lexicographically first minimal point $\mathbf{q}$ that minimizes $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$. Here, by minimal $\mathbf{q}$ we mean that there is no point $\mathbf{q}' < \mathbf{q}$, $\mathbf{q}' \in \text{Dom} \cap \overline{\mathbf{n}}$ with $h_{\text{WO}}(\mathbf{q}') - f(\mathbf{q}') \leq h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$, and if $\mathbf{q}_1$ and $\mathbf{q}_2$ are two distinct such minimal points then the lexicographically first one is the one with the smaller value at the first index between 1 and $d$ where the two points differ. Since $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q}) \geq 0$, the minimum exists if $\text{Dom} \cap \overline{\mathbf{n}}$ is not empty. Create the patch $P(\mathbf{q}, Q, \overline{\mathbf{n}})$. Then add $\mathbf{q}$ to $Q$ (so that future patches exclude points in already-created patches), and repeat until all points of $\text{Dom} \cap \overline{\mathbf{n}}$ are covered.

Since the above algorithm is deterministic, for a given a patch corner, the associated set of excluding points and congruence class are uniquely determined, so we simply refer to a patch by its corner. Moreover, the number of patches generated by the algorithm is finite. To see why, we use the following lemma from Angluin et al., which is in turn a corollary of Higman's Lemma [13].

▶ **Lemma 6.** (Angluin et al. [2], Higman [13].) *Every subset of $\mathbb{N}^d$ under the inclusion ordering $\leq$ has finitely many minimal elements.*

The algorithm selects patch corners $\mathbf{q}$ with nondecreasing value of $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$. The function $f$ is bounded above by $h_{\text{WO}}$. So a lower bound for $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$ is 0. If $\mathbf{x}_0$ is the minimum point of $\text{Dom} \cap \overline{\mathbf{n}}$, then when $\mathbf{x}_0$ is selected as a patch corner the algorithm must terminate. So the upper bound for $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$ is $h_{\text{WO}}(\mathbf{x}_0) - f(\mathbf{x}_0)$. Since $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$ is always integral, there are at most $h_{\text{WO}}(\mathbf{x}_0) - f(\mathbf{x}_0)$ different values for $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$ during the algorithm. Consider the set of points $\mathbf{q}$ in $\mathbb{N}^d$ with the same value $h_{\text{WO}}(\mathbf{q}) - f(\mathbf{q})$. By Lemma 6 this set has a finite number of minimal points. So the number of patches produced by the algorithm is equal to the sum of the sizes of these finite minimal point sets, summed over the finite different values in the range $0, \ldots, h_{\text{WO}}(\mathbf{x}_0) - f(\mathbf{x}_0)$. Thus the algorithm terminates after a finite number of steps, when run on each $\text{Dom} \cap \overline{\mathbf{n}}$, and $\mathbb{N}^d$ is covered by the union of all the patches, taken over all domains of $\mathcal{WO}_{\mathbf{w}}$ and congruence classes $\overline{\mathbf{n}}$.

We define $h_{\text{S}} : P(\mathbf{q}, Q, \overline{\mathbf{n}}) \to \mathbb{N}$ by $h_{\text{S}}(\mathbf{x}) = h_{\text{WO}}(\mathbf{x}) - h_{\text{WO}}(\mathbf{q}) + f(\mathbf{q})$. If $\mathbf{q}$ is in domain Dom of $h_{\text{WO}}$'s well-ordered representation, where on domain $\text{Dom} \cap \overline{\mathbf{n}}$ we have that $h_{\text{WO}}(\mathbf{x})$ is the affine function $h_{\text{WO}}(\mathbf{x}) = \nabla \cdot \mathbf{x} + b$, then we can write

$$h_{\text{S}}(\mathbf{x}) = \nabla \cdot \mathbf{x} + b - h_{\text{WO}}(\mathbf{q}) + f(\mathbf{q}). \tag{3}$$

That is, $h_{\text{S}} : P(\mathbf{q}, Q, \overline{\mathbf{n}}) \to \mathbb{N}$ is an affine function with gradient $\nabla$ and intercept $b - h_{\text{WO}}(\mathbf{q}) + f(\mathbf{q})$. Finally, we define $h_{\text{S}} : \mathbb{N}^d \to \mathbb{N}$ to be the union of these partial affine functions on patches. Next we prove several useful properties of $h_{\text{S}}$.

▶ **Lemma 7.** *For each patch corner $\boldsymbol{q}$, $h_S(\boldsymbol{q}) = f(\boldsymbol{q})$.*

**Proof.** Follows directly from the definition of $h_{\text{S}}$, since $h_{\text{S}}(\mathbf{q}) = h_{\text{WO}}(\mathbf{q}) - h_{\text{WO}}(\mathbf{q}) + f(\mathbf{q})$. ◀

▶ **Lemma 8.** *For all $\mathbf{x} \in \mathbb{N}^d$, $h_S(\mathbf{x}) \leq h_{WO}(\mathbf{x})$.*

**Proof.** Let $\mathbf{x}$ be in the patch with corner $\mathbf{q}$. Then $h_{\text{S}}(\mathbf{x}) = h_{\text{WO}}(\mathbf{x}) - h_{\text{WO}}(\mathbf{q}) + f(\mathbf{q}) \leq h_{\text{WO}}(\mathbf{x})$, since $h_{\text{WO}}(\mathbf{q}) \geq f(\mathbf{q})$. ◀

▶ **Lemma 9.** *For all $\mathbf{x} \in \mathbb{N}^d$, $f(\mathbf{x}) \leq h_S(\mathbf{x})$.*

**Proof.** Let $\mathbf{x}$ be in the patch with corner $\mathbf{q}$. Then by our choice of $\mathbf{q}$, $h_{\mathrm{WO}}(\mathbf{q}) - f(\mathbf{q}) \leq h_{\mathrm{WO}}(\mathbf{x}) - f(\mathbf{x})$. Rearranging the terms, we have that $f(\mathbf{x}) \leq h_{\mathrm{WO}}(\mathbf{x}) - h_{\mathrm{WO}}(\mathbf{q}) + f(\mathbf{q}) = h_{\mathrm{S}}(\mathbf{x})$. ◀

▶ **Lemma 10.** *Let $\mathbf{x}, \mathbf{x}' \in \mathbb{N}^d$ and let $\mathbf{x} \leq \mathbf{x}'$. Then the gradient of $h_S$ on the patch containing $\mathbf{x}$ is less than or equal to the gradient of $h_S$ on the patch containing $\mathbf{x}'$.*

**Proof.** Suppose that $\mathbf{x}$ and $\mathbf{x}'$ are in domains $\mathrm{Dom}_{\mathbf{o}}$ and $\mathrm{Dom}_{\mathbf{o}'}$ in the well-ordered domain representation of $h_{\mathrm{WO}}$. Then since $\mathbf{x} \leq \mathbf{x}'$, the gradient of $h_{\mathrm{WO}}$ on $\mathrm{Dom}_{\mathbf{o}}$ is less than or equal to the gradient of $h_{\mathrm{WO}}$ on $\mathrm{Dom}_{\mathbf{o}'}$ (the construction of Claim 4 satisfies this property). By construction of $h_{\mathrm{S}}$ in Equation (3), the gradient of $h_{\mathrm{S}}$ on a patch equals the gradient of $h_{\mathrm{WO}}$ in the domain containing the patch, and so the lemma follows. ◀

▶ **Lemma 11.** *Let $\mathbf{x}, \mathbf{x}' \in \mathrm{Dom} \cap \overline{\boldsymbol{n}}$, for some $\mathrm{Dom} \in \mathcal{WO}_{\boldsymbol{w}}$ and congruence class $\overline{\boldsymbol{n}}$. Suppose also that $\mathbf{x} \leq \mathbf{x}'$. Then the intercept of $h_S$ on $\mathbf{x}$ is less than or equal to the intercept of $h_S$ on $\mathbf{x}'$.*

**Proof.** The stated conditions of the lemma on $\mathbf{x}$ and $\mathbf{x}'$ imply that either $\mathbf{x}$ and $\mathbf{x}'$ are in the same patch, or the patch containing $\mathbf{x}$ is constructed *after* the patch containing $\mathbf{x}'$. The intercepts of $h_{\mathrm{S}}$ on patches within $\mathrm{Dom} \cap \overline{\mathbf{n}}$ are nonincreasing in the order of patch construction. ◀

▶ **Lemma 12.** *$h_S$ is superadditive.*

**Proof.** Let $\mathbf{x}_1$ and $\mathbf{x}_2$ be in patches $\mathbf{q}_1$ and $\mathbf{q}_2$, respectively. Then $\mathbf{q}_1 + \mathbf{q}_2 \leq \mathbf{x}_1 + \mathbf{x}_2$ and $\mathbf{q}_1 + \mathbf{q}_2$ and $\mathbf{x}_1 + \mathbf{x}_2$ are in the same congruence class. Also, by Lemma 3, the points $x_1 + x_2$ and $q_1 + q_2$ lie in the same domain of $h_{\mathrm{WO}}$. Let $\mathbf{x}_1 + \mathbf{x}_2$ be in the patch with corner $\mathbf{q}$.

On the patches with corners $\mathbf{q}_1$, $\mathbf{q}_2$, and $\mathbf{q}$, let $h_{\mathrm{S}}(\mathbf{x}_1) = \nabla_1 \cdot \mathbf{x} + b_1$, $h_{\mathrm{S}}(\mathbf{x}_2) = \nabla_2 \cdot \mathbf{x} + b_2$, and $h_{\mathrm{S}}(\mathbf{x}) = \nabla \cdot \mathbf{x} + b$, respectively. By Lemma 10, $\nabla_1 \leq \nabla$ and $\nabla_2 \leq \nabla$. Also, we have that

$$\begin{aligned} h_{\mathrm{S}}(\mathbf{q}_1) + h_{\mathrm{S}}(\mathbf{q}_2) &= f(\mathbf{q}_1) + f(\mathbf{q}_2) && \text{(by Lemma 7)} \\ &\leq f(\mathbf{q}_1 + \mathbf{q}_2) && \text{(since } f \text{ is superadditive)} \\ &\leq h_{\mathrm{S}}(\mathbf{q}_1 + \mathbf{q}_2) && \text{(by Lemma 9)} \\ &\leq \nabla \cdot (\mathbf{q}_1 + \mathbf{q}_2) + b, \end{aligned}$$

where the last inequality follows by Lemmas 10 and 11. Then

$$\begin{aligned} h_{\mathrm{S}}(\mathbf{x}_1) + h_{\mathrm{S}}(\mathbf{x}_2) &= h_{\mathrm{S}}(\mathbf{x}_1) - h_{\mathrm{S}}(\mathbf{q}_1) + h_{\mathrm{S}}(\mathbf{x}_2) - h_{\mathrm{S}}(\mathbf{q}_2) + h_{\mathrm{S}}(\mathbf{q}_1) + h_{\mathrm{S}}(\mathbf{q}_2) \\ &= \nabla_1 \cdot (\mathbf{x}_1 - \mathbf{q}_1) + \nabla_2 \cdot (\mathbf{x}_2 - \mathbf{q}_2) + h_{\mathrm{S}}(\mathbf{q}_1) + h_{\mathrm{S}}(\mathbf{q}_2) \\ &\leq \nabla \cdot (\mathbf{x}_1 - \mathbf{q}_1) + \nabla \cdot (\mathbf{x}_2 - \mathbf{q}_2) + \nabla \cdot (\mathbf{q}_1 + \mathbf{q}_2) + b \\ &= \nabla \cdot (\mathbf{x}_1 + \mathbf{x}_2) + b \\ &= h_{\mathrm{S}}(\mathbf{x}_1 + \mathbf{x}_2). \end{aligned}$$ ◀

▶ **Lemma 13.** *$h_S$ is well-ordered quilt-affine.*

**Proof.** Define $\mathbf{w}'$ to be the vector whose $i$th component $w'_i$ is $\max_{\mathbf{q}} q_i$, rounded up to be 0 mod $p$. The domain set $\mathcal{WO}_{\mathbf{w}'}$ is a refinement of the original domain set $\mathcal{WO}_{\mathbf{w}}$ of $h_{\mathrm{WO}}$'s representation. Let $\mathrm{Dom}_{\mathbf{o}'}$ be one of the domains of $\mathcal{WO}_{\mathbf{w}'}$ (where $\mathbf{o}' \leq \mathbf{w}'$), and let $\mathrm{Dom}_{\mathbf{o}'} \subset \mathrm{Dom}_{\mathbf{o}}$, where $\mathrm{Dom}_{\mathbf{o}} \in \mathcal{WO}_{\mathbf{w}}$.

Fix any congruence class $\overline{\mathbf{n}}$ of $\mathbb{Z}^d / pZ^d$. If $\mathrm{Dom}_{\mathbf{o}'} \cap \overline{\mathbf{n}}$ is not empty, let $\mathbf{m}$ be the smallest point in $\mathrm{Dom}_{\mathbf{o}'} \cap \overline{\mathbf{n}}$. Let $\mathbf{q}$ be the corner of the patch containing $\mathbf{m}$. Note that $\mathbf{q}$ is in $\mathrm{Dom}_{\mathbf{o}}$.

We claim that $\mathrm{Dom}_{\mathbf{o}'} \cap \overline{\mathbf{n}}$ is contained in the patch with corner $\mathbf{q}$. This is trivially true if $\mathrm{Dom}_{\mathbf{o}'} \cap \overline{\mathbf{n}}$ is finite, and thus a single point. Consider the case where $\mathrm{Dom}_{\mathbf{o}'}$ is infinite. Let

$\mathbf{x} \in \mathrm{Dom}_{\mathbf{o}'} \cap \overline{\mathbf{n}}$ and let $\mathbf{q}'$ be the corner of the patch containing $\mathbf{x}$. We claim that $\mathbf{q}' \leq \mathbf{m}$. To see why, note that if $x_i > m_i$ then it must be that $m_i \geq w'_i$ and by our choice of $\mathbf{w}'$, $q'_i \leq m_i$. Otherwise, $x_i \leq m_i$ and so $q'_i \leq x_i \leq m_i$. But then $\mathbf{q} = \mathbf{q}'$, since $\mathbf{q}$ is the corner of the patch containing $\mathbf{m}$. Therefore all of $\mathrm{Dom}_{\mathbf{o}'} \cap \overline{\mathbf{n}}$ is in the patch with corner $\mathbf{q}$. It follows that $h_{\mathrm{S}}$ on domain $\mathrm{Dom}_{\mathbf{o}'} \cap \overline{\mathbf{n}}$ is a single affine function, namely that associated with the patch with corner $\mathbf{q}$. Moreover, the gradient of this function is the gradient of the function $h_{\mathrm{WO}}$ on domain $\mathrm{Dom}_{\mathbf{o}} \in \mathcal{WO}_{\mathbf{w}}$. Since this is true for any congruence class $\overline{\mathbf{n}}$, the function $h_{\mathrm{S}}$ on domain $\mathrm{Dom}_{\mathbf{o}'}$ is a quilt-affine function whose gradient is the same as that of $f$ on $\mathrm{Dom}_{\mathbf{o}}$, completing the proof. ◀

From Lemmas 8 and 9, we have that $f = \min\{h_{\mathrm{S}}\}$ where the min is taken over a finite number of functions $h_{\mathrm{S}}$. Moreover, from Lemma 12, each $h_{\mathrm{S}}$ is superadditive and from Lemma 13, each $h_{\mathrm{S}}$ is well-ordered quilt-affine. The proof of Claim 5 follows. ◁

## 4 A Leaderless Output-Oblivious CRN for Superadditive, Obliviously-Computable Functions

Here we show the second half of our main result, Theorem 1, by constructing a leaderless, output-oblivious CRN for any superadditive, well-ordered quilt-affine function.

▷ Claim 14. Any superadditive, well-ordered quilt-affine function can be stably computed by a leaderless, output-oblivious CRN.

Proof. Let $f : \mathbb{N}^d \to \mathbb{N}$ be a superadditive, obliviously-computable function. From Claim 5, we know that $f = \min\{h_{\mathrm{S}}\}$, where each of the finitely many $h_{\mathrm{S}}$ is a superadditive, well-ordered quilt-affine function. Below we show that any such function has a leaderless, output-oblivious CRN, say $\mathcal{C}_{h_{\mathrm{S}}}$. A leaderless, output-oblivious CRN for $f$ can then be obtained from the $\mathcal{C}_{h_{\mathrm{S}}}$'s via the following steps: (i) for each function $h_{\mathrm{S}}$, create a unique replica $X_{h_{\mathrm{S}},i}$ of each input species $X_i$ ; (ii) adapt $\mathcal{C}_{h_{\mathrm{S}}}$ by replacing input species $X_i$ by the replica $X_{h_{\mathrm{S}},i}$, in every reaction and for each $i$ and replacing the output species $Y$ of $\mathcal{C}_{h_{\mathrm{S}}}$ with $Y_{h_{\mathrm{S}}}$ in every reaction; and (iii) adding the reaction $\sum_{h_{\mathrm{S}}} Y_{h_{\mathrm{S}}} \to Y$, which implements the min function.

Fix any superadditive, well-ordered quilt-affine function $h$, and a representation of $h$ with well-ordered domain set $\mathcal{WO}_{\mathbf{w}}$ and period $p \in \mathbb{N}^+$. To simplify our proof we will assume without loss of generality that $p > 1$. Recall that there is one domain $\mathrm{Dom}_{\mathbf{o}}$ in $h$'s representation for each $\mathbf{o} \leq \mathbf{w}$. We partition these domains by taking intersections with congruence classes mod $p$. For each $\mathrm{Dom}_{\mathbf{o}} \in \mathcal{WO}_{\mathbf{w}}$ and each congruence class $\overline{\mathbf{x}}$ of $\mathbb{Z}^d/p\mathbb{Z}^d$ such that $\mathrm{Dom}_{\mathbf{o}} \cap \overline{\mathbf{x}}$ is non-empty, let $\mathbf{m} = \mathbf{m}(\mathbf{o}, \overline{\mathbf{x}})$ be the minimum point in the subdomain $\mathrm{Dom}_{\mathbf{o}} \cap \overline{\mathbf{x}}$, and denote this subdomain by $\mathrm{Dom}'_{\mathbf{m}}$. Let $\mathcal{N}$ be the set of all such $\mathbf{m}$. By our assumption that $p > 1$, it must be that all unit vectors $\mathbf{e}_i$ are in $\mathcal{N}$, $1 \leq i \leq d$. Since $h$ is quilt-affine with period $p$, we have that $h(\mathbf{x})$ on $\mathrm{Dom}'_{\mathbf{m}}$ is a partial affine function, which we denote by $h_{\mathbf{m}}(\mathbf{x}) = \nabla_{\mathbf{m}}(\mathbf{x}) + b_{\mathbf{m}}$, where $\nabla_{\mathbf{m}} = \nabla_{\mathbf{o}}$ if $\mathbf{m} = \mathbf{m}(\mathbf{o}, \overline{\mathbf{x}})$.

Our CRN has input species $X_1, X_2, \ldots, X_d$ and an output species $Y$. We will use $\mathbf{x}$ to denote the vector of counts of input species consumed, and $y$ to denote the number of $Y$'s produced, during an execution of the CRN. Our CRN also has a *leader* species $L_{\mathbf{m}}$ for $\mathbf{m} \in \mathcal{N}$, and a *distance* species $P_{\mathbf{m},i}$ for each $\mathbf{m} \in \mathcal{N}$ and each $i \in \{1, \ldots, d\}$. We will use $\#L_{\mathbf{m}}$ and $\#P_{\mathbf{m},i}$ to denote counts of leader and distance species, during an execution of the CRN.

The leader and distances species will track how much input has been consumed by reactions. To build intuition on how this works, it may be helpful first to imagine that there is just one leader. In this case, if the input $\mathbf{x}$ consumed so far is in domain $\mathrm{Dom}'_{\mathbf{m}}$, then our

reactions will ensure that the leader is $L_{\mathbf{m}}$ and that for $1 \leq i \leq d$, $\#P_{\mathbf{m},i} = (x_i - m_i)/p$, i.e., the distance of the consumed input $\mathbf{x}$ from $\mathbf{m}$, along the $i$th dimension. (Since $\mathbf{x} \in \mathrm{Dom}'_{\mathbf{m}}$, $x_i - m_i$ is a multiple of $p$.) Thus,

$$\mathbf{x} = \mathbf{m} + p \sum_{1 \leq i \leq d} \#P_{\mathbf{m},i} \times \mathbf{e}_i.$$

Generalizing to the leaderless scenario, consumption of input will produce many leaders; we can imagine that the consumed input is distributed over many domains $\mathrm{Dom}_{\mathbf{m}}$. Our reactions will ensure that a generalization of the above equality holds:

$$\mathbf{x} = \sum_{\mathbf{m} \in \mathcal{N}} \left( \#L_{\mathbf{m}} \times \mathbf{m} + p \sum_{i \in \{1,\ldots,d\}} \#P_{\mathbf{m},i} \times \mathbf{e}_i \right), \tag{4}$$

and we call the term on the right hand side of this invariant the *input value* of the CRN configuration. The invariant trivially holds initially since both $\mathbf{x}$ and the input value are $\mathbf{0}$. Our reactions will also maintain the following *output invariant*:

$$y = \sum_{\mathbf{m} \in \mathcal{N}} \left( \#L_{\mathbf{m}} \times h(\mathbf{m}) + \sum_{i \in \{1,\ldots,d\}} \#P_{\mathbf{m},i} \times \nabla_{\mathbf{m},i} \right). \tag{5}$$

We call the term on the right hand side of this invariant the *output value*. Initially both $y$ and the output value are 0. We will show that once our CRN stabilizes, the output value is the function $h$ applied to the input value, and so these invariants ensure that $y = h(\mathbf{x})$ upon stabilization.

Our CRN has three types of reactions. We next describe these, and show that each respects the input and output invariants. Figure 4, included in the appendix, shows an example of a function $h$, a quilt-affine representation and the partitioning of the quilt-affine domains (via intersections with congruence classes), and Figure 5, also in the appendix, illustrates part of our CRN construction for the function of Figure 4.

## 4.1 Input-Consuming Reactions

These reactions consume inputs and produce leader species. There is one reaction for each $i, 1 \leq i \leq d$:

$$X_i \to L_{\mathbf{e}_i} + h(\mathbf{e}_i) \times Y.$$

This reaction consumes input $\mathbf{e}_i$, and recall that by our assumption that $p > 1$, $\mathbf{e}_i \in \mathcal{N}$. So, no distance species are needed to ensure that the input invariant holds. Producing $h(\mathbf{e}_i)$ Y's ensures that the output invariant holds.

## 4.2 Merge Reactions

Merge reactions reduce the number of leader species, effectively electing a single leader:

$$L_{\mathbf{m}} + L_{\mathbf{m}'} \to L_{\mathbf{m}''} + \delta \times Y + \sum_{j \in \{1..d\}} \delta'_j P_{\mathbf{m}'',j}.$$

Here, $\mathbf{m}''$ is chosen such that $\mathrm{Dom}'_{\mathbf{m}''}$ contains $\mathbf{m} + \mathbf{m}'$. To ensure that the input invariant holds upon a merge reaction, we choose $\delta'_j = (n_j + n'_j - n''_j)/p$. Plugging this value into the input invariant (4) shows that the input value is unchanged, which is necessary since no

input is consumed. To ensure that the output invariant holds, we set $\delta$ to be equal to the change in the output value as a result of the reaction (increase due to addition of products minus decrease due to removal of reactants):

$$\begin{aligned}\delta &= h(\mathbf{m}'') + p\textstyle\sum_{j\in\{1..d\}} \delta'_j \times \nabla_{\mathbf{m}'',j} - h(\mathbf{m}) - h(\mathbf{m}') \\ &= h(\mathbf{m}+\mathbf{m}') - h(\mathbf{m}) - h(\mathbf{m}').\end{aligned}$$

In this case, $\delta$ is non-negative because $f$ is superadditive.

## 4.3 Exchange Reactions

The exchange reactions ensure that, once there is a single leader molecule, say $L_{\mathbf{m}}$, eventually all of the distance species $P_{\mathbf{m}',i}$ are such that $\mathbf{m}' = \mathbf{m}$. Let $\mathbf{m}$ and $\mathbf{m}'$ be in $\mathcal{N}$, with $\mathbf{m} \neq \mathbf{m}'$. Let $\mathbf{m}$ and $\mathbf{m}'$ be in the well-ordered domains of $\mathcal{WO}_{\mathbf{w}}$ with offsets $\mathbf{o}$ and $\mathbf{o}'$, respectively. Recall that $\nabla_{\mathbf{m}} = \nabla_{\mathbf{o}}$ and $\nabla_{\mathbf{m}'} = \nabla_{\mathbf{o}'}$. Suppose without loss of generality that $\mathbf{o} \leq \mathbf{o}'$ (in which case $\nabla_{\mathbf{o}} \leq \nabla_{\mathbf{o}'}$), and that if $\mathbf{o} = \mathbf{o}'$ then $\mathbf{m} \leq \mathbf{m}'$. Then we add the following reactions, for $1 \leq i \leq d$:

$$L_{\mathbf{m}} + P_{\mathbf{m}',i} \to L_{\mathbf{m}} + \delta \times Y + P_{\mathbf{m},i}.$$

Each exchange reaction preserves the input invariant because the input value is unchanged and no input is consumed. To ensure that the output invariant holds, we set $\delta$ to equal the change in the output value as a result of the reaction (increase due to addition of products minus decrease due to removal of reactants):

$$\begin{aligned}\delta &= h(\mathbf{m}) + p\nabla_{\mathbf{m},i} - h(\mathbf{m}) - p\nabla_{\mathbf{m}',i} \\ &= p(\nabla_{\mathbf{m},i} - \nabla_{\mathbf{m}',i}) \\ &= p(\nabla_{\mathbf{o},i} - \nabla_{\mathbf{o}',i}) \\ &\geq 0, && \text{since } \nabla_{\mathbf{o}} \geq \nabla_{\mathbf{o}'}.\end{aligned}$$

This completes the description of the reactions of the CRN.

## 4.4 Correctness

A "leader dominance" invariant that is maintained by all reactions is that for any $P_{\mathbf{m}',i}$ with positive count, there is also some leader $L_{\mathbf{m}}$ with positive count, such that if $\mathrm{Dom}_{\mathbf{o}}$ and $\mathrm{Dom}_{\mathbf{o}'}$ are the well-ordered domains containing $\mathbf{m}$ and $\mathbf{m}'$, respectively then $\mathbf{o} \geq \mathbf{o}'$. The input consuming and exchange reactions trivially maintain this invariant. Consider a merge reaction with reactants $L_{\mathbf{m}}$ and $L_{\mathbf{m}'}$ that produces $L_{\mathbf{m}''}$. Suppose that $\mathbf{m}$, $\mathbf{m}'$, and $\mathbf{m}''$ are in the well-ordered domains with offsets $\mathrm{Dom}_{\mathbf{o}}$, $\mathrm{Dom}_{\mathbf{o}'}$ and $\mathrm{Dom}_{\mathbf{o}''}$, respectively. Then by Lemma 3, since $\mathrm{Dom}'_{\mathbf{m}''}$ contains $\mathbf{m}+\mathbf{m}'$, $\mathrm{Dom}_{\mathbf{o}''}$ must contain $\mathbf{o}+\mathbf{o}'$. Therefore, $\mathbf{o}'' = (\mathbf{o}+\mathbf{o}') \diamond \mathbf{w}$, where we use $\diamond$ to denote the element-wise min. So it must be that $\mathbf{o}'' \geq \mathbf{o}'$, and the leader dominance invariant must hold upon a merge reaction.

Next we show that the CRN stabilizes. First note that eventually all input species are consumed by the input-consuming reactions, at which point no more leaders will be produced. Also, eventually there is exactly one leader, because of the merge reactions. At this point, the only possible reactions are exchange reactions. Each exchange reaction reduces the number of $P_{\mathbf{m}',i}$ with $\mathbf{m}' \neq \mathbf{m}$. By the leader dominance invariant, this number will eventually reach zero, at which point no more exchange reactions are possible.

Suppose that, once no more reactions are possible, the leader is $L_{\mathbf{m}}$, in which case the only distance species with count greater than zero are species $P_{\mathbf{m},i}$ for some $i$. As a result, we have that

$$\begin{aligned} y &= h(\mathbf{m}) + p\textstyle\sum_{i\in 1..d} \#P_{\mathbf{m},i} \times \nabla_{\mathbf{m},i} && \text{from the output invariant} \\ &= h(\mathbf{m}) \nabla_{\mathbf{n}} p(\textstyle\sum_{i\in 1..d} \#P_{\mathbf{m},i} \times \mathbf{e}_i + \mathbf{m} - \mathbf{m}) \\ &= h(\mathbf{m}) + \nabla_{\mathbf{m}} \times (\mathbf{x} - \mathbf{m}) && \text{from the input invariant} \\ &= h(\mathbf{x}). \end{aligned}$$

This ensures that the output is correct once the CRN has stabilized, completing the proof. ◁

## 5 Conclusion

We have classified the functions $f : \mathbb{N}^d \to \mathbb{N}$ which are stably computable by CRNs that are (a) leaderless, and (b) never consume their own output. This result sheds light on the fundamental limitations of discrete CRNs. Indeed, together with previous work on CRNs with leaders [16], this has completed the classification of functions which are stably computable by output-oblivious CRNs – with and without leaders. Such results inform the larger question of composability in this model of computation, and to what extent such systems can be comprised of smaller, modular components.

While composition with guaranteed correctness seems dubious for functions which are not output-oblivious, we emphasize that there are nevertheless routes to composition with a high probability of correctness. Phase-clocks for example, a ubiquitous tool in population protocols (e.g., [3, 12, 1]), may be used to prohibit a CRN from being activated for some number of time steps. Kosowski and Uznański recently demonstrated how to build hierarchies of phase clocks; these could be leveraged to construct an arbitrarily long series of CRN compositions [14].

A question raised by our results is the extent to which the theory of discrete and continuous CRNs can be reconciled. As mentioned in the introduction, our results mirror those for continuous CRNs, but our techniques are quite distinct. It would be useful to know whether and under what conditions certain statements apply to both models. Is their a theoretical framework allowing both continuous and discrete CRNs to be studied simultaneously?

A separate question is whether CRNs which compute output-oblivious functions, but are not themselves output-oblivious, can be augmented with reactions to make them so. For CRNs implemented as strand displacement systems, for instance, it may be easier to add reactions than to change the underlying network entirely. Understanding the limitations of being able to edit in this way would shed light on the possibility of building CRNs incrementally instead of requiring that the design be understood beforehand.

## References

1 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2221–2239. SIAM, 2018.

2 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 292–299, New York, NY, USA, 2006. ACM Press. doi:10.1145/1146381.1146425.

3 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

4 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

5 Stefan Badelt, Seung Woo Shin, Robert F. Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. A general-purpose CRN-to-DSD compiler with formal verification, optimization, and simulation capabilities. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, pages 232–248, Cham, 2017. Springer International Publishing.

6 Cameron Chalk, Niels Kornerup, Wyatt Reeves, and David Soloveichik. Composable rate-independent computation in continuous chemical reaction networks. In Milan Ceska and David Safránek, editors, *Computational Methods in Systems Biology*, pages 256–273, Cham, 2018. Springer International Publishing.

7 Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural Computing*, 13(4):517–534, December 2014.

8 Ho-Lin Chen, David Doty, and David Soloveichik. Rate-independent computation in continuous chemical reaction networks. In *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science*, ITCS 2014, pages 313–326, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2554797.2554827.

9 Ben Chugg, Hooman Hashemi, and Anne Condon. Output-oblivious stochastic chemical reaction networks. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, volume 125 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.OPODIS.2018.21.

10 Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. *Algorithmic Bioprocesses*, pages 543–584, 2009.

11 David Doty and Monir Hajiaghayi. Leaderless deterministic chemical reaction networks. *Natural Computing*, 14(2):213–223, 2015.

12 Leszek Gąsieniec and Grzegorz Staehowiak. Fast space optimal leader election in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2653–2667. SIAM, 2018.

13 G Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(2):326–336, 1952.

14 Adrian Kosowski and Przemysław Uznański. Population protocols are fast. *arXiv preprint arXiv:1802.06872*, 2018.

15 Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

16 Eric E. Severson, David Haley, and David Doty. Composable computation in discrete chemical reaction networks. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 14–23. ACM, 2019. doi:10.1145/3293611.3331615.

17 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7, 2008.

18 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

19 David Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature chemistry*, 3:103–13, February 2011. doi:10.1038/nchem.957.

# A Appendix

$$h(x_1, x_2) = \begin{cases} x_1, & x_2 = 0 \\ x_2, & x_1 = 0 \\ h'(x_1, x_2), & x_1 \geq 1, x_2 \geq 1. \end{cases}$$

(a) A superadditive, output-oblivious function $h$, where $h'(x_1, x_2) = 2x_1 + 2x_2 - ((x_1 + x_2) \bmod 2)$.

$$h(x_1, x_2) = \begin{cases} 0, & (x_1, x_2) \in \mathrm{Dom}_{00} = \{(0,0)\} \\ x_1, & (x_1, x_2) \in \mathrm{Dom}_{01} = \{(x_1, 0) + (1,0) \mid x_1 \in \mathbb{N}\} \\ x_2, & (x_1, x_2) \in \mathrm{Dom}_{10} = \{(0, x_2) + (0,1) \mid x_2 \in \mathbb{N}\} \\ h'(x_1, x_2), & (x_1, x_2) \in \mathrm{Dom}_{11} = \{(x_1, x_2) + (1,1) \mid x_1, x_2 \in \mathbb{N}\}. \end{cases}$$

(b) A well-ordered, quilt-affine representation of $h$. The domain set $\mathcal{WO}_{\mathbf{w}}$ has period 2, $\mathbf{w} = 11$ and contains four domains $\mathrm{Dom}_{\mathbf{o}}$ as shown, for $\mathbf{o} \in \{00, 01, 10, 11\}$.

$$h(x_1, x_2) = \begin{cases} 0, & (x_1, x_2) \in \mathrm{Dom}'_{00} = \mathrm{Dom}_{00} \cap \overline{00} \\ x_1, & (x_1, x_2) \in \mathrm{Dom}'_{01} = \mathrm{Dom}_{01} \cap \overline{01} \\ x_1, & (x_1, x_2) \in \mathrm{Dom}'_{02} = \mathrm{Dom}_{01} \cap \overline{00} \\ x_2, & (x_1, x_2) \in \mathrm{Dom}'_{10} = \mathrm{Dom}_{10} \cap \overline{10} \\ x_2, & (x_1, x_2) \in \mathrm{Dom}'_{20} = \mathrm{Dom}_{10} \cap \overline{00} \\ 2x_1 + 2x_2, & (x_1, x_2) \in \mathrm{Dom}'_{11} = \mathrm{Dom}_{11} \cap \overline{11} \\ 2x_1 + 2x_2 - 1, & (x_1, x_2) \in \mathrm{Dom}'_{12} = \mathrm{Dom}_{11} \cap \overline{10} \\ 2x_1 + 2x_2 - 1, & (x_1, x_2) \in \mathrm{Dom}'_{21} = \mathrm{Dom}_{11} \cap \overline{01} \\ 2x_1 + 2x_1, & (x_1, x_2) \in \mathrm{Dom}'_{22} = \mathrm{Dom}_{11} \cap \overline{00} \end{cases}$$

(c) Representation of $h$ on nonempty domains of the form $\mathrm{Dom}'_{\mathbf{n}} = \mathrm{Dom}'_{\mathbf{n}(\mathbf{o}, \overline{\mathbf{z}})} = \mathrm{Dom}_{\mathbf{o}} \cap \overline{\mathbf{z}}$, for each congruence class $\overline{\mathbf{z}}$ of $\mathbb{Z}^2/2\mathbb{Z}^2$, where $\overline{\mathbf{z}} = \{2(x_1, x_2) + \mathbf{z} \mid x_1, x_2 \in \mathbb{N}\}$ for each $\mathbf{z} = 00, 01, 10, 11$, and $\mathbf{n} = \mathbf{n}(\mathbf{o}, \overline{\mathbf{z}})$ is the minimum point in $\mathrm{Dom}_{\mathbf{o}} \cap \overline{\mathbf{z}}$.

**Figure 4** (a) A superadditive, output-oblivious function $h$. (b) Quilt-affine representation of $h$. (c) Representation of $h$ used in our leaderless CRN construction. Here as in Figure 5, we use strings to denote vectors, e.g. 11 denotes $(1, 1)$.

| Input-consuming Reactions | Sample Merge Reactions (involving $L_{01}$ or $L_{11}$) | Sample Exchange Reactions (involving $L_{11}$ or $L_{22}$) |
|---|---|---|
| $X_1 \to L_{10} + Y$<br>$X_2 \to L_{01} + Y$ | $L_{01} + L_{10} \to L_{11} + 2Y$<br>$L_{01} + L_{x1} \to L_{x2}, x \in \{0,1\}$<br>$L_{01} + L_{21} \to L_{22} + 2Y$<br>$L_{01} + L_{02} \to L_{01} + P_{01,2}$<br>$L_{01} + L_{x2} \to L_{x1} + 2Y + P_{x1,2}, x \in \{1,2\}$<br><br>$L_{11} + L_{01} \to L_{21}$<br>$L_{11} + L_{11} \to L_{22}$<br>$L_{11} + L_{21} \to L_{12} + P_{21,1}$<br>$L_{11} + L_{12} \to L_{21} + P_{21,2}$<br>$L_{11} + L_{22} \to L_{11} + 2Y + P_{11,x}, x \in \{1,2\}$ | $L_{11} + P_{01,x} \to P_{11,x} + 2Y$<br>$L_{11} + P_{10,x} \to P_{11,x} + 2Y$<br><br>$L_{22} + P_{01,x} \to P_{22,x} + 2Y$<br>$L_{22} + P_{10,x} \to P_{22,x} + 2Y$<br>$L_{22} + P_{11,x} \to P_{22,x}$<br>$L_{22} + P_{21,x} \to P_{22,x}$<br>$L_{22} + P_{12,x} \to P_{22,x}$ |

**Figure 5** Sample reactions of the leaderless, output-oblivious CRN for the function $h$ of Figure 4, obtained from our construction of Claim 14.

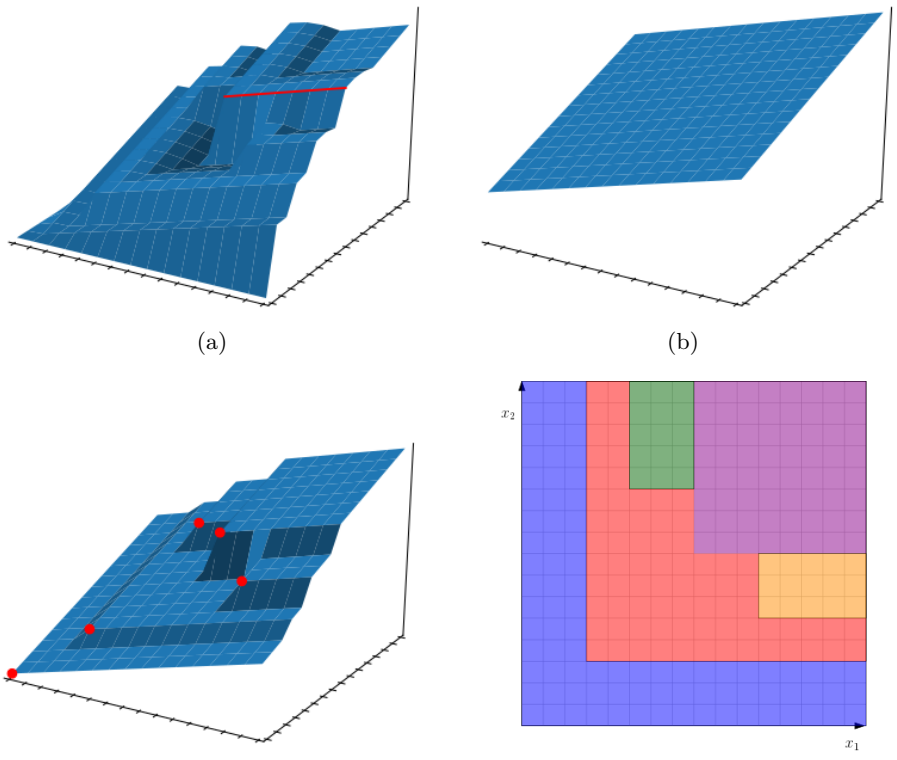



**Figure 6** (a) An output-oblivious function $f(\mathbf{x})$. (b) By Claim 4, the function $f$ of part (a) can be written as $f = \min\{h_{\text{WO}}\}$, where each of the finitely many functions $h_{\text{WO}}$ is nondecreasing, well-ordered quilt-affine. One of these functions is shown here. This function happens to be quite simple, with period 1 and one domain, namely $\mathbb{N}^2$, and $f = h_{\text{WO}}$ on the red line shown in part (a). (c) The superadditive, obliviously-computable function $h_{\text{S}}$ that is derived from the function $h_{\text{WO}}$ of part (b) via the construction of Claim 5. Patch corners are shown as red dots. The function $h_{\text{S}}$ has the same gradient as $h_{\text{WO}}$ on each patch, but has different intercepts. (d) Each coloured region is a patch on $\mathbb{N}^2$ (i.e., the congruence class has period 1). These patches correspond to the corners of part (c).

# CRNs Exposed: A Method for the Systematic Exploration of Chemical Reaction Networks

**Marko Vasic**
The University of Texas at Austin, TX, USA
https://marko-vasic.github.io/
vasic@utexas.edu

**David Soloveichik**
The University of Texas at Austin, TX, USA
http://users.ece.utexas.edu/~soloveichik/
david.soloveichik@utexas.edu

**Sarfraz Khurshid**
The University of Texas at Austin, TX, USA
https://users.ece.utexas.edu/~khurshid/
khurshid@utexas.edu

## Abstract

Formal methods have enabled breakthroughs in many fields, such as in hardware verification, machine learning and biological systems. The key object of interest in systems biology, synthetic biology, and molecular programming is *chemical reaction networks* (CRNs) which formalizes *coupled chemical reactions* in a well-mixed solution. CRNs are pivotal for our understanding of biological regulatory and metabolic networks, as well as for programming engineered molecular behavior. Although it is clear that small CRNs are capable of complex dynamics and computational behavior, it remains difficult to explore the space of CRNs in search for desired functionality. We use Alloy, a tool for expressing structural constraints and behavior in software systems, to enumerate CRNs with declaratively specified properties. We show how this framework can enumerate CRNs with a variety of structural constraints including biologically motivated catalytic networks and metabolic networks, and seesaw networks motivated by DNA nanotechnology. We also use the framework to explore analog function computation in rate-independent CRNs. By computing the desired output value with stoichiometry rather than with reaction rates (in the sense that $X \to Y + Y$ computes multiplication by 2), such CRNs are completely robust to the choice of reaction rates or rate law. We find the smallest CRNs computing the *max*, *minmax*, *abs* and *ReLU* (rectified linear unit) functions in a natural subclass of rate-independent CRNs where rate-independence follows from structural network properties.

**2012 ACM Subject Classification** Theory of computation

**Keywords and phrases** molecular programming, formal methods

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.4

**Supplementary Material** We release the source code of our tool at `https://github.com/marko-vasic/crnsExposed` to enable others make use of it, and extend it further.

**Acknowledgements** This work was supported in part by NSF grants CCF-1901025 to DS and CCF-1718903 to SK.

## 1 Introduction

Formal methods have enabled breakthroughs in many fields, e.g., in hardware verification [15], machine learning [23, 32], and biological systems [5, 24, 29, 40, 61]. In this paper we apply formal methods to *Chemical Reaction Networks* (CRNs), which have been objects of intense study in systems and synthetic biology. CRNs are widely used in modeling biological regulatory networks, and essentially identical models are also widely used in ecology [60],


distributed computing [2], and other fields. More recently, CRNs have been directly used as a programming language for engineering molecules obeying prescribed interaction rules via DNA strand displacement cascades [6, 12, 53, 55, 57].

It is clear that small CRNs can exhibit very complex behavior. Dynamical systems, e.g., oscillatory, chaotic, and bistable systems, typically contain only a few reactions. Small CRNs also exhibit interesting computational behavior. For example, the approximate majority population protocol studied in distributed computing [1] was later identified with a variety of biological networks [7]. Can we systematically explore the power of small reaction networks?

We present a method that exhaustively enumerates small CRNs in different classes that are relevant for biology and for synthetic engineering systems. The enumeration is performed using Alloy, a powerful tool for modeling structural constraints and behavior in software systems using first-order logic with transitive closure [33]. The Alloy tool performs *scope-bounded* analysis [35]. Given an Alloy model and a *scope*, i.e., a bound on the universe of discourse, the analyzer translates the Alloy model to a propositional satisfiability (SAT) formula and invokes an off-the-shelf SAT solver [20] to analyze the model. Alloy is used in a wide range of areas in software engineering, including software design [21, 34], analysis [19, 22, 36, 38], testing [44], and security [37]. We show how Alloy can be used to conveniently model interesting classes of CRNs for biology and bioengineering, and we use the Alloy analyzer to search for CRNs with specific desired functionality.

As examples of the method we first focus on a number of classes: elementary, catalytic, metabolic. We say *elementary reactions* are CRNs with at most two reactants and products. (We allow reactions to be irreversible; reversible reactions are represented by two irreversible reactions.) *Catalytic networks* are those elementary CRNs in which the reactants and products are not disjoint; i.e., the reaction is catalyzed by some species that is not consumed in the reaction. Catalytic networks (e.g., transcriptional, phosphorylation, etc.) regulate many aspects of the cell's behavior [42, 48]. In general protein-protein interactions, proteins can catalytically modify other proteins, which in turn can be catalysts in other interactions. An important subclass of catalytic networks are *metabolic networks*, where the enzymes are proteins while the substrates are small molecules; these catalytic CRNs are "bipartite" in the sense that a species is either always a catalyst or never a catalyst. *Autocatalytic networks* are another interesting subclass of catalytic networks in which the (auto)catalyst generates another copy of itself. Autocatalysis is useful for exponential amplification and oscillation.

We then turn our attention to classes of CRNs especially relevant for synthetic reaction networks, showing how abstract molecular structure can be modeled in Alloy. In particular, we focus on DNA strand displacement cascades, which have proved to be a uniquely programmable technology for cell-free DNA-only systems [64]. Strand displacement interactions correspond to reactions between two types of molecules: "gates" and "strands", where the reacting strand displaces the strand previously sequestered in the gate complex. A simple, yet very scalable, class of strand displacement circuits uses a simple motif called *seesaw* gates [13, 49, 50] that makes use of a reversible strand displacement reaction. We designed an Alloy model to enumerate such strand displacement reactions, showing that abstract molecular structure can be incorporated into the Alloy modeling formalism.

In the second part of the paper, we use our enumeration framework to search for specific desired functionality in a class of CRNs. In particular, we focus on the class of rate-independent CRNs [11]. Consider the reaction $X \to Y + Y$, and think of the concentrations of species $X$ and $Y$ as input and output respectively. This reaction computes the function of "multiplication by 2" since in the limit of time going to infinity it produces two units of $Y$ for every unit of $X$ initially present. Similarly the reaction $X_1 + X_2 \to Y$ computes the

$$
\begin{aligned}
A &\longrightarrow Z_1 + Y \\
B &\longrightarrow Z_2 + Y \\
Z_1 + Z_2 &\longrightarrow K \\
Y + K &\longrightarrow \emptyset
\end{aligned}
$$

**Figure 1** CRN computing Max. We think of the initial amount of $A$ and $B$ as inputs, and the converging amount of $Y$ as the output. The amount of $Y$ eventually produced in reactions 1 and 2 is the sum of the initial amounts of $A$ and $B$. The amount of $K$ eventually produced in reaction 3 is the minimum of the initial amounts of $A$ and $B$. Reaction 4 subtracts the minimum from the sum, yielding the maximum. (The 4th reaction generates waste species, which are not named.)

"minimum" function since the amount of $Y$ eventually produced will be the minimum of the initial amounts of $X_1$ and $X_2$. Note that such computation makes no assumption on the rate law, such as whether the reaction obeys mass-action kinetics[1] or not, allowing the computation to be correct in a wide variety of chemical contexts. (We use the continuous CRN model where concentrations are real-valued quantities.)

A natural subclass of CRNs whose structure enforces rate independence are those that satisfy two constraints: feed-forward, and non-competitive.[2] Intuitively, the first condition ensures that the CRN converges to a static equilibrium where no reaction can occur. The second condition ensures that no matter what the rates are, the system converges to the *same* static equilibrium. More precisely, we define feed-forward as follows: there exists a total ordering on the reactions such that no reaction consumes[3] a species produced by a reaction later in the ordering. We define non-competitive as follows: if a species is consumed in a reaction then it cannot appear as a reactant somewhere else. Such constraints on the structure of the network can be easily encoded in the Alloy specification. We also require each reaction to consume at least one species (boundedness condition). We show in Appendix A that these conditions ensure that the CRN is rate-independent.

Focusing on the class of feed-forward, non-competitive CRNs, we search for the smallest reaction networks implementing *max*, *minmax*, *abs*, and *ReLU* (rectified linear unit) functions. As an example of the kind of computation we achieve, consider the *max* computing CRN shown in Figure 1. This CRN was previously studied [10, 11]; our result shows that it is indeed the smallest. The maximum function serves an important role in rate-independent computation since together with minimum, multiplication and division by a constant it forms a complete basis set [9, 11]. The *ReLU* function was first introduced due to the biological motivations explaining functioning of neurons in the brain cortex [27]. Since then, it was used with great success in the machine learning community, particularly in deep learning [25, 41] for realizing artificial neural networks. The simplicity of its implementation suggests that CRNs can naturally realize neural computation [58]. To our knowledge, the smallest implementations of *abs* (absolute value), and *minmax* (a two output function computing both minimum and maximum of two inputs) that we find are novel and have not been previously published.

[1] "Mass-action" kinetics refers to the best-studied case where the reaction rate is proportional to the product of the concentration of the reactants.

[2] Feed-forward and non-competitive conditions are sufficient for rate-independence, but are not necessary. However, most known examples of rate independent computation satisfy these conditions.

[3] We say a reaction *produces* (resp. *consumes*) a species $S$ if there is net stoichiometric gain (resp. loss) of $S$. Thus a catalyst in a reaction is neither consumed nor produced.

**Listing 1** General Alloy model of CRNs. "−−" indicate start of a comment.

```
module crn

abstract sig Species {}
abstract sig Reaction { reactants, products: seq Species }

-- Basic semantic constraints -- for all CRNs
fact AtLeastOneReactant { -- each reaction has >=1 reactant
  all r: Reaction | some r.reactants }

fact UniqueReactions { -- each reaction is unique
  all disj r1, r2 : Reaction | ReactionsDifferent[r1, r2] }

pred ReactionsDifferent[r1, r2: Reaction] {
  SpeciesSeqDifferent[r1.reactants, r2.reactants]
  or SpeciesSeqDifferent[r1.products, r2.products] }

pred SpeciesSeqDifferent[seq1, seq2: seq Species] {
  some s : Species | #indsOf[seq1, s] != #indsOf[seq2, s] }

fact ReactantsDifferentThanProducts {
  all r: Reaction | SpeciesSeqDifferent[r.reactants, r.products] }

fact AllSpeciesUsed { -- each species is used in some reaction
  Int.(Reaction.(reactants + products)) = Species }

pred ContainsAsReactant[r: Reaction, s: Species] { s in Int.(r.reactants) }
pred ContainsAsProduct[r: Reaction, s: Species] { s in Int.(r.products) }
```

Much ongoing work explores the computational power of CRNs. Previous work showed the implementation of numerous complex behaviors, such as mapping polynomials to chemical reactions [51], programming logic gates [43], mapping discrete, control flow, algorithms [31], and a molecular programming language translating high-level specifications to chemical reactions [59]. However the complexity of these reaction systems can be infeasible, asking for novel techniques that answer what is the natural way to compute "in reactions". To help answer this question we can take a different, bottom-up approach, and explore what small CRNs naturally do. We believe that insight we get from exploring reactions will help in design of higher-level primitives that naturally map to reactions, and will provide knowledge for more efficient design of high-level languages.

## 2 Modeling CRNs in Alloy

This section describes our approach to modeling chemical reaction networks (CRNs) in Alloy. (See Appendix B for additional background on Alloy.) We first introduce a general model to represent the broadest class of CRNs (allowing arbitrary number of reactants and products), and next show specializations of the model for different classes such as *elementary*, *catalytic*, *metabolic*, *autocatalytic*, and *feed-forward non-competitive* reactions. Next, we present models that encode abstract molecular structure, including *strands and gates* model and a *seesaw* model built on top of it. Our approach naturally admits a hierarchical structuring of models where a model builds on and specializes another model – e.g., metabolic reactions are structurally more constrained reactions than elementary. This allows a systematic exploration of the design space of models as this section illustrates.

**General model.** Our general model captures CRNs consisting of reactions with arbitrarily many reactants and products. To model this in Alloy we define a set of species, a set of reactions, two relations that characterize the reactants and products, and logical constraints that define the basic structural requirements for well-formed CRNs. Listing 1 specifies the general model in Alloy. The keyword `module` allows naming the model, which can be imported in other models. The keyword `sig` declares a basic type and introduces a set of indivisible atoms that do not have any internal structure. The model declares two sets: a set of species (`Species`) and a set of reactions (`Reaction`). The signature declaration of `Reaction` introduces two *fields*, `reactants` and `products`, each of type *sequence* (`seq`) of `Species`. Alloy models a sequence as a binary relation from (non-negative) integer indices to atoms. Thus, each of these field declarations introduces a ternary relation of type: `Reaction` $\times$ `Int` $\times$ `Species`. In a case of reaction $R0 : X \rightarrow Y + Y$, the value of products relation would be the set: $\{R0 \times 0 \times Y, R0 \times 1 \times Y\}$. Note that we model reactants and products with *seq* instead of *set* to support repetition of a species as a reactant or product, as in the above reaction.

After defining the basic structure, we use Alloy *facts* to add constraints ensuring that enumerated CRNs are well-formed. A `fact` paragraph states a constraint that must always be satisfied, i.e., every solution found (CRN enumerated) must satisfy each fact (and may satisfy additional constraints as desired). For example, the fact `AtLeastOneReactant` requires that every reaction contains at least one reactant. We use universal quantification (`all`) to require that the reactants in each reaction form a non-empty sequence. The keyword `some` in formula "`some E`" for expression `E` constrains it to represent a non-empty set. The operator '`.`' is relational join; specifically, if `r` and `s` are binary relations where the domain of `r` is the same as co-domain of `s`, `r.s` is relational composition, and if `x` is a scalar and `t` is a binary relation where the type of `x` is the co-domain of `t`, `x.t` is relational image of `x` under `t`. Thus, `r.reactants` represents a sequence of reactants in a reaction $r$.

We ensure that there are no two identical reactions in a CRN using the fact `UniqueReactions`. For all distinct (`disj`) reactions we require that predicate `ReactionsDifferent` holds. A *predicate* (`pred`) paragraph is a named formula that may have parameters. The predicate `ReactionsDifferent` uses logical disjunction (`or`) and invokes `SpeciesSeqDifferent` to constrain its parameters (reactions) `r1` and `r2` to be different.

The predicate `SpeciesSeqDifferent` is true if the two sequences of species are different. It uses existential quantification (`some`). The operator '`#`' represents set cardinality. The Alloy library function `indsOf` represents the set of indices where the atom argument (e.g., `s`) appears in the sequence argument (e.g., `seq1`). Intuitively, this predicate compares the number of appearances of species in two sequences, and returns true if exists a species that appears a different number of times in the two sequences.

The fact `ReactantsDifferentThanProducts` requires each reaction to have non-identical reactants and products. Finally, the fact `AllSpeciesUsed` states that all species must be a part of some reaction. `Int` represents the set of integers.

The predicate `ContainsAsReactant` is true if a given reaction contains a given species as a reactant. Similar holds for `ContainsAsProduct` and reaction products.

**Illustrating the General Model.** To illustrate using the Alloy analyzer, consider generating an instance of the constraints modeled. The following `Generate` command instructs the analyzer to create an instance with respect to a universe that contains exactly 2 reactions and 2 species, and 2-bit integers, and conforms to all the facts in the model:

```
Generate: run {} for exactly 2 Reaction, exactly 2 Species, 2 int
```

**Listing 2** Elementary reactions.

```
module elementary
open crn
pred Elementary() { MaxReactantsNum[2] and MaxProductsNum[2] }
pred MaxReactantsNum[num: Int] { all r: Reaction | lte[#r.reactants, num] }
pred MaxProductsNum[num: Int] { all r: Reaction | lte[#r.products, num] }
```

**Listing 3** Catalytic reactions.

```
module catalytic
open elementary
pred Catalytic[] { all r: Reaction | CatalyticReaction[r] }
pred CatalyticReaction[r: Reaction] { some elems[r.reactants] & elems[r.products] }
run { Catalytic and Elementary } for 2
```

Executing the command `Generate` and enumerating the first three instances creates the following CRNs where $S0$ and $S1$ are species, and $\emptyset$ are waste species [4]:

$S_1 \longrightarrow S_0$
$S_0 \longrightarrow S_1$
(a)

$S_1 \longrightarrow \emptyset$
$S_1 \longrightarrow S_0$
(b)

$S_1 \longrightarrow \emptyset$
$S_0 \longrightarrow S_1$
(c)

While quite small, these three instances exhibit interesting properties, CRN in (a) models a reversible reaction $S1 \longleftrightarrow S0$; CRN in (b) is rate-dependent, where amount of $S1$ in a limit of time going to infinity is 0, but amount of $S0$ is dependent on reaction rates; and CRN in (c) is rate-independent, where concentrations of both $S0$ and $S1$ converge to 0.

**Elementary reactions.** Elementary reactions have at most 2 reactants and at most 2 products. Elementary reactions are arguably the ones commonly occurring in nature, as it is unlikely that 3 (or more) molecules react or split at the same exact time. Also, reactions with more than 2 reactants can be represented with elementary reactions; e.g. reaction $A + B + C \to D$ can be constructed with two elementary reactions: $A + B \to T$ and $T + C \to D$. (Similarly for products.)

Listing 2 shows the Alloy model of *elementary* reactions, which specializes (restricts) the general CRN model `crn`. The Alloy model `elementary` imports (`open`) the `crn` model and defines the predicate `Elementary`, which uses the conjunction (`and`) of two helper predicates `MaxReactantsNum` and `MaxProductsNum` to characterize elementary reactions. The predicate `lte` is a standard Alloy utility predicate and represents the $\leq$ comparison.

**Catalytic reactions.** Next, we model catalytic reactions (Listing 3). The predicate `Catalytic` uses the helper predicate `CatalyticReaction` to require each reaction to be catalytic, i.e., have some species that is both a reactant and a product in that reaction. The Alloy utility function `elems` represents the set of elements in its argument sequence; the operator '`&`' represents set intersection. The `run` command instructs the analyzer to create an instance

[4] Alloy shows each instance as a valuation to the sets and relations declared in the model, and also supports visualizing the instances as graphs. We write the reactions here using their natural representation for clarity.

**Listing 4** Metabolic reactions.

```
module metabolic
open catalytic

pred Metabolic[] {
  Catalytic[] and
  all s: Species | (some r: Reaction | IsCatalyst[s, r]) implies
                   all x: Reaction | Contains[x, s] implies IsCatalyst[s, x] }

pred IsCatalyst[s: Species, r: Reaction] { s in Int.(r.reactants) & Int.(r.products) }
pred Contains[r: Reaction, s: Species] { ContainsAsReactant[r, s] or ContainsAsProduct[
    r, s] }
```

**Listing 5** Strands and gates.

```
module strandsandgates
open crn

sig Strand, Gate extends Species {}
fact { Strand + Gate = Species } -- strands and gates partition species

pred StrandsAndGates() {
  ExactReactantsNum[2] and ExactProductsNum[2] and
  all r: Reaction {
    some Int.(r.reactants) & Strand and some Int.(r.reactants) & Gate
    some Int.(r.products) & Strand and some Int.(r.products) & Gate }}

pred ExactReactantsNum[num: Int] { all r: Reaction | eq[#r.reactants, num] }
pred ExactProductsNum[num: Int] { all r: Reaction | eq[#r.products, num] }
```

that is both a catalytic and an elementary reaction within a scope of 2, i.e., at most 2 atoms in each sig. An example instance created by executing the command is:

$$S_0 + S_1 \to S_0 + S_0$$
$$S_0 + S_1 \to S_1 + S_1$$

We also model autocatalytic reactions shown in Appendix C.

**Metabolic reactions.** In metabolic networks catalysts are proteins that act upon substrates that are small molecules. Thus metabolic reactions are a form of catalytic reactions in which if a species appears as a catalyst in a reaction, then it has to be a catalyst in all reactions in which the species occurs. The predicate `Metabolic` in Listing 4 specifies metabolic reactions.

**Strands and gates.** We next model synthetic CRNs which use DNA strand displacement cascades for its implementation. Strand displacement interactions correspond to reactions between two types of molecules: "gates" and "strands", where the reacting strand displaces the strand previously sequestered in the gate complex. We first capture the bipartite nature of the reactions: Listing 5 declares strands and gates as disjoint subsets (`extends`) that partition species. The predicate `StrandsAndGates` requires that each reaction has exactly 2 reactants and 2 products, and moreover has a strand and a gate as a reactant, and a strand and a gate as a product.

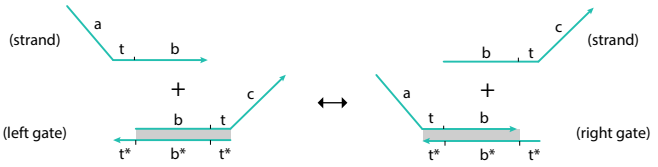



**Figure 2** DNA strand displacement reaction with the seesaw gate motif. There are two reactants (a strand and a gate) and two products (a strand and a gate). A gate consists of two strands bound together. (For simplicity the usual helical structure of DNA is not shown.) Labels show binding sites (domains); a star indicates Watson-Crick complement such that domain $x$ binds $x^*$. In order for the reaction to happen, the complementary domains must match as shown. Such reactions can be cascaded since the strands $< a, t, b >$ and $< b, t, c >$ can react with other seesaw gates.

**Seesaw networks.** A simple yet powerful subclass of DNA strand displacement reactions is the "seesaw" model. Seesaw reactions have been used to create some of the largest synthetic biochemical reaction networks, including logic circuits and neural networks [13, 49]. The molecular structure schematic for a seesaw reaction is shown in Figure 2. Listing 6 models seesaw reactions by specializing the model of strands and gates (Listing 5), capturing the abstract molecular structure in an Alloy model. The signature `Domain` models the binding domains. The signature `DNASpecies` is a subset (`in`) of species, and `left` and `right` are binary relations that map `DNASpecies` to their left and right domains respectively. The keyword `lone` constraints the relations to be partial functions. The signatures `RightGate` and `LeftGate` partition gates. The fact `UseAll` requires all species to be DNA species, and requires all domains to be a part of some species . The fact `UniqueSpecies` enforces that strands and gates are unique, i.e., there cannot be two or more strands (or left/right gates) with matching left and right domains. The fact `OneDomain` requires strands and gates to have exactly one left and exactly one right domain. The predicate `CanReactStrandAndLeftGate` is true if inputs (reactants) conform to the interaction rules of a strand and a left gate, similar holds for the predicate `CanReactStrandAndRightGate` on strands and right gates. The predicate `CanReact` is true if inputs (reactants) satisfy either `CanReactStrandAndLeftGate` or `CanReactStrandAndRightGate`. The predicate `ReactStrandAndLeftGate` is true if inputs (reactants and products) conform to the interaction rules of a strand and a left gate, specifically `s` and `lg` interact, i.e., the right domain of `s` matches the left domain of `lg`, and produce `s'` and `rg'` where the left and right domains of `s'` match those of `lg`, and left and right domains of `rg'` match those of `s`; likewise, `ReactStrandAndRightGate` specifies the interaction of a strand and a right gate. The functions `ReactantsSet` and `ProductsSet` returns a set of reactants (products) in a reaction. The predicate `Seesaw` specifies: (a) each reaction to be a seesaw reaction by enforcing the predicate `React` on every reaction; (b) that all possible reactions exist, i.e., if two species can interact based on seesaw interaction rules (predicate `CanReact`) than a reaction containing those species as reactants (or products) must exist; (c) that reactions only in one direction exist (to reduce number of solutions we enforce that only one direction of reaction exist in enumerated CRNs knowing that seesaw reactions are always reversible); (d) that reactions have a left gate as a reactant (this is to prevent multiple redundant solutions, since all reactions are reversible we can enforce that left gate is always on the left hand side).

An instance generated by Alloy running the predicate with command `GenSeesaw` is $S_{ab} + LG_{bc} \rightarrow S_{bc} + RG_{ab}$, where $S_{ab}$ and $S_{bc}$ are strands, $LG_{bc}$ left gate, $RG_{ab}$ right gate, while left and right domains $\{a, b, c\}$ are denoted in subscript. Note that this reaction is equivalent to the one shown in Figure 2.

**Listing 6** Seesaw model.

```
open strandsandgates

sig Domain {}
sig DNASpecies in Species { left, right: lone Domain }
sig RightGate, LeftGate extends Gate {}

fact UseAll { DNASpecies = Species and DNASpecies.(left + right) = Domain }
fact UniqueSpecies {
  all s1, s2: Strand | s1.left = s2.left and s1.right = s2.right implies s1 = s2
  all s1, s2: RightGate | s1.left = s2.left and s1.right = s2.right implies s1 = s2
  all s1, s2: LeftGate | s1.left = s2.left and s1.right = s2.right implies s1 = s2 }
fact OneDomain { all s: Strand + LeftGate + RightGate | one s.left and one s.right }

pred CanReactStrandAndLeftGate[s: Strand, lg: LeftGate] {
  s in Strand and lg in LeftGate and s.right = lg.left }
pred CanReactStrandAndRightGate[s: Strand, rg: RightGate] {
  s in Strand and rg in RightGate and s.left = rg.right }
pred CanReact[r1: DNASpecies, r2: DNASpecies] {
  CanReactStrandAndLeftGate[r1, r2] or CanReactStrandAndRightGate[r1, r2] }

pred ReactStrandAndLeftGate[s: Strand, lg: LeftGate, s':Strand, rg': RightGate] {
  (s in Strand and lg in LeftGate and s' in Strand and rg' in RightGate
   and CanReactStrandAndLeftGate[s, lg]
   and s'.left = lg.left and s'.right = lg.right and rg'.left = s.left and rg'.right = s
       .right) }
pred ReactStrandAndRightGate[s: Strand, rg: RightGate, s': Strand, lg': LeftGate] {
  (s in Strand and rg in RightGate and s' in Strand and lg' in LeftGate
   and CanReactStrandAndRightGate[s, rg]
   and s'.left = rg.left and s'.right = rg.right and lg'.left = s.left and lg'.right = s
       .right) }
pred React[r1: Species, r2: Species, p1: Species, p2: Species] {
  ReactStrandAndLeftGate[r1, r2, p1, p2] or ReactStrandAndRightGate[r1, r2, p1, p2] }

fun ReactantsSet[r: Reaction]: set Species { Int.(r.reactants) }
fun ProductsSet[r: Reaction]: set Species { Int.(r.products) }

pred Seesaw {
  StrandsAndGates[]
  all r: Reaction { -- All reactions are seesaw reactions.
    let s = 0.(r.reactants), g = 1.(r.reactants), s' = 0.(r.products), g' = 1.(r.
        products) {
      React[s, g, s', g'] }}
  all s1, s2: Species { -- All possible reactions exist.
    CanReact[s1, s2] implies some r: Reaction {
      (s1 + s2) = ReactantsSet[r] or (s1 + s2) = ProductsSet[r] }}
  all s1, s2: Species | all rxn1, rxn2: Reaction { -- Prevent reverse direction.
    ((s1+s2) = ReactantsSet[rxn1]) implies ((s1+s2) != ProductsSet[rxn2]) }
  all r: Reaction { some LeftGate & ReactantsSet[r] }
}

GenSeesaw: run Seesaw for exactly 1 Reaction, exactly 3 Domain, exactly 4 Species
```

To reduce the enumeration overhead for seesaw, we updated the `Reaction` signature by removing the representation of reactants and products as a sequence (sequence introduces integers as an overhead), and adding two relations for reactants and products (as seesaw reactions are restricted to two reactants and two products). The updated `Reaction` signature is: `abstract sig Reaction { r1, r2, p1, p2:  Species }`

**Listing 7** Feed-forward, non-competitive CRNs in Alloy.

```
open elementary

one sig Graph { edges: Reaction -> Reaction }
{ all r1, r2: Reaction | r1->r2 in edges implies some s: Species |
    NetProduces[r1, s] and NetConsumes[r2, s]
  all s: Species | all r1, r2: Reaction |
    NetProduces[r1, s] and NetConsumes[r2, s] implies r1->r2 in edges }

pred DAG[] { all r: Reaction | r !in r.^(Graph.edges) }

pred NonCompetitive[] {
  all r1, r2: Reaction | all s : Species {
    (ContainsAsReactant[r1, s] and NetConsumes[r2, s]) implies r1 = r2 }}

pred NetProduces[r: Reaction, s: Species] { -- r net produces s
  lt[#indsOf[r.reactants,s], #indsOf[r.products,s]] }

pred NetConsumes[r: Reaction, s: Species] { -- r net consumes s
  gt[#indsOf[r.reactants,s], #indsOf[r.products,s]] }

pred MustConsume[] {
  all r: Reaction | some s: Species | NetConsumes[r, s] }

pred Feedforward[] { Elementary[] and DAG[] and NonCompetitive[] and MustConsume[] }
```

**Feed-forward, non-competitive CRNs.** Listing 7 models feed-forward, non-competitive CRNs. Recall, we define feed-forward as: there exists a total ordering on the reactions such that no reaction consumes a species produced by a reaction later in the ordering. Also, we define non-competitive as: every species is consumed by at most one reaction.

To model feed-forward constraints, one approach is to directly enforce a total ordering on the reactions with respect to the feed-forward property. Observe that there can be multiple valid total orderings of reactions for the same feed-forward CRN, which means that when enumerating instances for the resulting model, multiple unique instances are created for the same CRN. This is useful when finding all total orderings that exist for a CRN. However, our goal is to search for CRNs exhibiting desired functionality, and thus we aim to enumerate each CRN once, and as quickly as possible. To tackle this problem we achieve the total ordering by creating a graph of reaction dependencies, and enforce it to be *directed-acyclic*.

Our modeling of feed-forward constraints introduces a new *singleton* (`one`) sig, termed `Graph`, to model a dependency relation, termed `edges`, between reactions. The constraint paragraph that immediately follows the signature declaration implicitly introduces a fact that defines the edges. Specifically, there is an edge from reaction `r1` to reaction `r2` if and only if there is some species `s` such that `r1` produces `s` and `r2` consumes `s`. Total ordering is achieved by the predicate `DAG` that requires the graph to be *directed-acyclic*. The operator '`^`' is transitive closure and `r.^(Graph.edges)` represents the set of all reactions that are reachable from `r`. The predicate `NonCompetitive` enforces that if a species is used as a reactant in a reaction then it cannot be consumed by any other reaction. The predicate `MustConsume` enforces that every reaction consumes some species (boundedness condition). The predicate `Feedforward` defines elementary, feed-forward, and non-competitive reactions where each reaction must consume some species.

**Algorithm 1** Search Algorithm.

**Input:** Model (*model*), Generation bounds (*scope*), Function ($f$), Inputs ($N$).
**Output:** CRN that computes $f$ if found; otherwise, null.

```
1: procedure ExhaustiveSearch
2:     for each instance ∈ Alloy.findAllInstances(model, scope) do
3:         crn ← translate(instance)
4:         if ComputesF(crn, f, N) then return crn
5:     end for
6:     return null
7: end procedure
```

## 3 CRN Enumeration and Search

In this section we describe our algorithm (shown in Algorithm 1) that performs a bounded exhaustive search enumerating all CRNs in a given class and within a given bounds respecting properties defined by an Alloy model, to find the CRN implementing desired function.

Inputs to the algorithm are the Alloy model, the size of CRNs (e.g., number of reactions and species) defined by the *scope*, desired target function $f$, and the number of inputs to the function $N$. Function *findAllInstances* accepts the Alloy model definition and scope, and enumerates all possible instances that satisfy the Alloy model. Each Alloy instance is translated to CRN (step 3). Then, in step 4 we invoke the Algorithm 2 (Section 4) to check if CRN computes $f$. If CRN implementing given function is found then it is returned (step 4). If after checking all instances no satisfying CRN is found then the procedure returns *null*.

**Bounded exhaustive search** . To find the smallest CRN computing $f$ we conduct a bounded exhaustive search. Our goal is to find a smallest (in terms of numbers of species and reactions) feed-forward, non-competitive CRN that computes $f$. We use *iterative deepening* [26, 28, 30] where we start from a small scope and iteratively increase it to a larger scope until a desired CRN is found, where for each scope we invoke Algorithm 1.

## 4 CRN Analysis

In this section we describe our algorithm for checking if a CRN computes a function of interest ($f$).

**Conservation Equations.** We first construct a set of conservation equations for the CRN which describe concentrations of species in terms of their initial concentrations and reaction fluxes. A reaction flux is equal to the total "flow of material" through the reaction. We associate a flux variable to the each reaction, where $flux_i$ represents the flux of the reaction $i$. Then the concentration of a species $S$ can be expressed in terms of its initial concentration $S_0$ and reaction fluxes:

$$s = s_0 + \sum_{i=1}^{N} netGain(rxn_i, S) \cdot flux_i \tag{1}$$

where $netGain(rxn_i, S)$ is the net stoichiometric gain of species $S$ in the reaction $i$ (negative in the case of loss), and $N$ is the number of reactions in the CRN. For example, the CRN from Figure 1 generates the equations shown in 2. The variables on the left side of equations represent concentrations of species, variables with suffixes 0 represent initial concentrations of

species (e.g., $z_{10}$ is initial concentration of species $Z_1$), and finally $flux_i$ variables represent fluxes of reactions.

$$\begin{array}{ll} a = a_0 - flux_1 & b = b_0 - flux_2 \\ z_1 = z_{10} + flux_1 - flux_3 & z_2 = z_{20} + flux_2 - flux_3 \\ k = k_0 + flux_3 - flux_4 & y = y_0 + flux_1 + flux_2 - flux_4 \end{array} \tag{2}$$

**Equilibrium Condition.** We next use the above conservation equations to find equilibria. Since we focus on rate-independent computation, we search for static equilibria only (none of the reactions is occurring).[5] A static equilibrium corresponds to every reaction having at least one reactant in zero concentration. Thus, we create multiple systems of equations from the conservation equations, where each system corresponds to setting concentrations of a set of species to zero, where the set contains a reactant from each reaction. The solution of each such constructed system of equations represents concentrations of species at an equilibrium. Different equilibria will be reached from different initial conditions.

As an example, consider again the CRN shown in Figure 1. All combinations of species containing a reactant from each reaction are: $(A, B, Z_1, Y)$, $(A, B, Z_2, Y)$, $(A, B, Z_1, K)$, $(A, B, Z_2, K)$. For each combination we set its species concentrations to zero and solve the system 2. This results in 4 solutions shown in 3 (we do not show solutions for flux variables due to the space limits).

| $\boldsymbol{a}$ | $\boldsymbol{b}$ | $\boldsymbol{k}$ | $\boldsymbol{y}$ | $\boldsymbol{z_1}$ | $\boldsymbol{z_2}$ |
|---|---|---|---|---|---|
| 0 | 0 | $-b_0 + k_0 - y_0 + z_{10}$ | 0 | 0 | $-a_0 + b_0 - z_{10} + z_{20}$ |
| 0 | 0 | $-a_0 + k_0 - y_0 + z_{20}$ | 0 | $a_0 - b_0 + z_{10} - z_{20}$ | 0 |
| 0 | 0 | 0 | $b_0 - k_0 + y_0 - z_{10}$ | 0 | $-a_0 + b_0 - z_{10} + z_{20}$ |
| 0 | 0 | 0 | $a_0 - k_0 + y_0 - z_{20}$ | $a_0 - b_0 + z_{10} - z_{20}$ | 0 |

(3)

Although there are 4 solutions, for any particular initial concentrations of the species only one of the solutions is non-negative (concentrations of species must be non-negative), and thus feasible.

**Check whether CRN computes $f$.** We then check if the equilibrium solutions are equivalent to $f$. In general, we do not know which species correspond to the input and which to the output, and thus we need to check for all possible combinations of the input and the output species. First, we construct all input $n$-tuples without repeating elements from a set of species (where $n$ is the number of the inputs to $f$)[6]. Second, for all species that are not in the input tuple we set initial concentrations to zero. Third, for the output species we try any of the remaining species. Fourth, for a given set of input and output species, we construct a piecewise function, where each solution is valid if concentrations of species are non-negative. Finally, we use Mathematica's constraint solving procedure *FindInstance* to check if the constructed piecewise function differs from function $f$.

[5] In chemical kinetics, *static* equilibrium refers to an equilibrium where none of the reactions occur. In contrast, in *dynamic* equilibria, concentrations don't change over time because the effects of the different reactions cancel out. Note that dynamic equilibria are not rate-independent since changing a reaction rate affects the equilibrium concentrations of the species involved in that reaction.

[6] An input tuple ($a$,$b$) will be separately considered from ($b$,$a$). However, if the sought function is known to be commutative than the order of species can be ignored.

**Algorithm 2** ComputesF.

**Input:** CRN *crn*, Function $f$, Number of inputs $N$.
**Output:** *True* if *crn* computes $f$; *false* otherwise.

```
1: procedure ComputesF
2:     conservationEquations ← constructConservationEquations(crn)
3:     equilibriumSolutions ← ∅
4:     for each speciesSet ∈ getAllReactantCombinations(crn) do
5:         equilibriumEquations ← setConcToZero(conservationEquations, speciesSet)
6:         solution ← solve(equilibriumEquations)
7:         equilibriumSolutions.add(solution)
8:     end for
9:     for each {x_1, x_2, ..., x_N, y} ∈ getInputOutputSpecies(crn, N) do
10:        nonInputSpecies ← getOtherSpecies(crn, {x_1, x_2, ..., x_n})
11:        newSols ← setInitialConcToZero(equilibriumSolutions, nonInputSpecies)
12:        pwF ← constructPiecewise(newSols, y)
13:        counterExample ← FindInstance(pwF ≠ f(x_1, x_2, ..., x_N))
14:        if counterExample = null then return true
15:    end for
16:    return false
17: end procedure
```

To illustrate on our example, consider setting input species to $A$ and $B$, and output to $Y$. The system of equations 3 reduces to the system 4.

| $a$ | $b$ | $k$ | $y$ | $z_1$ | $z_2$ |
|---|---|---|---|---|---|
| 0 | 0 | $-b_0$ | 0 | 0 | $-a_0+b_0$ |
| 0 | 0 | $-a_0$ | 0 | $a_0-b_0$ | 0 |
| 0 | 0 | 0 | $b_0$ | 0 | $-a_0+b_0$ |
| 0 | 0 | 0 | $a_0$ | $a_0-b_0$ | 0 |

(4)

The first two solutions are infeasible since they result in species $k$ having negative concentration, $-b_0$ and $-a_0$. More precisely they are feasible only in the trivial case where $a_0 = 0 \wedge b_0 = 0$. The third solution is feasible when $b_0 \geq a_0$, in which case $y = b_0$; while fourth solution is feasible when $a_0 \geq b_0$, in which case $y = a_0$. Thus, we can construct the piecewise function unifying multiple equilibrium solutions into a single function:

$$y = \begin{cases} b_0 & b_0 \geq a_0 \\ a_0 & a_0 \geq b_0 \end{cases}$$

Next, once we constructed the equilibrium piecewise function ($y(a_0, b_0)$) we invoke the Mathematica's constraint solving procedure FindInstance to find an assignment of inputs $(a_0, b_0)$ for which $y$ differs from $f$, with additional condition that initial concentrations are non-negative ($a_0 \geq 0 \wedge b_0 \geq 0$). If no counterexample is found, then the CRN computes $f$ and we have finished our search. On the other hand, if a counterexample is found, then we repeat the procedure for the next combination of input and output species. When the list of input and output combinations is exhausted we can conclude that the CRN does not compute $f$.

**Algorithm.** We implement this functionality in Mathematica by defining *ComputesF* function described in Algorithm 2. In step 2, conservation equations are constructed, while in step 3 we initialize a set of equilibrium solutions *equilibriumSolutions* to an empty set. In steps 4–8, we iterate over all existing sets of species containing at least one reactant from each reaction. Specifically, function *getAllReactantCombinations* computes Cartesian product over sets of reactants from different reactions; and removes elements with the same sets of species. In step 5 we update the conservation equations by setting *speciesSet* concentrations to zero,

**Table 1** Number of enumerated feed-forward, non-competitive CRNs and wall-clock times (hh:mm:ss) for the enumeration procedure.

| | 1 *Reaction* | | 2 *Reactions* | | 3 *Reactions* | | 4 *Reactions* | |
|---|---|---|---|---|---|---|---|---|
| 1 *Species* | 3 | 00:00:00 | 0 | 00:00:00 | 0 | 00:00:00 | 0 | 00:00:00 |
| 2 *Species* | 10 | 00:00:00 | 22 | 00:00:00 | 0 | 00:00:00 | 0 | 00:00:00 |
| 3 *Species* | 6 | 00:00:00 | 199 | 00:00:00 | 287 | 00:00:00 | 0 | 00:00:00 |
| 4 *Species* | 1 | 00:00:00 | 391 | 00:00:00 | 4,666 | 00:00:05 | 5,643 | 00:00:07 |
| 5 *Species* | 0 | 00:00:00 | 291 | 00:00:00 | 17,509 | 00:00:19 | 140,064 | 00:03:57 |
| 6 *Species* | 0 | 00:00:00 | 100 | 00:00:00 | 27,257 | 00:00:32 | 817,742 | 00:30:35 |

and save the linear system in *equilibriumEquations*. In steps 6–7 we solve the system of linear equations and add it to the list of equilibrium solutions (note that since we are focused on feed-forward non-competitive reactions, a unique solution will always exist). Next, we iterate over all combinations of input and output species $\{x_1, x_2, ..., x_N, y\}$, where $x_1$, $x_2$, ..., $x_N$ represent input species, and $y$ output species. In step 10 we get all the species that are not in the input species set. In step 11 we modify the equilibrium solutions by setting initial concentrations of *nonInputSpecies* to zero, and we save the result in *newSols*. In step 12 we construct a piecewise function *pwF* out of *newSols*. Finally, in step 13 we invoke the *FindInstance* method to find input values for which *pwF* is different then $f$. If such solution is not found then *counterExample* is *null*, and constructed *pwF* is implementing $f$; in which case procedure returns *true*. If counterexample is found then the same steps are repeated for different set of input and output species. Finally, if all combinations are exhausted procedure returns *false*.

## 5 New Results

In this section we present new discoveries made using the proposed techniques. We focus on the class of feed-forward, non-competitive CRNs since they are always rate-independent.

**Smallest *max* CRN.** We perform bounded exhaustive search for 1 to 4 reactions, and 1–6 species, starting with smaller number of species and reactions, and iteratively increasing the scope until the *max* is found. Table 1 shows the number of enumerated CRNs and Alloy enumeration time for different scope sizes. We perform (not perfect) isomorphic breaking in Alloy by requiring lexicographic ordering on reactions among other things (details of symmetry breaking are shown in Appendix F). Note that while we perform some isomorphic breaking[7], not all isomorphic cases are pruned, and thus number of non-isomorphic instances may be less then numbers reported in Table 1. In spite of this, our approach is still exhaustive, meaning that all possible CRNs will be enumerated, but some may be enumerated multiple times. The first occurrence of *max* is found in the scope of 4 reactions and 6 species, and it was the $124,118^{th}$ instance Alloy enumerated in that scope. The CRN discovered is equivalent to the one shown in Figure 1, modulo reaction and species ordering.

[7] Alloy can generate *isomorphic* instances, i.e., two instances that are distinct but there exists a permutation on atoms, which maps one instance to the other

$$X^+ \longrightarrow M + Y^+$$
$$M + X^- \longrightarrow Y^-$$

$$X^+ \longrightarrow Y^+ + C$$
$$X^- \longrightarrow Y^+ + E$$
$$C + E \longrightarrow 2\,Y^-$$

$$X_1^+ \longrightarrow M_1 + Y_{max}^+$$
$$X_1^- \longrightarrow M_2 + Y_{min}^-$$
$$X_2^+ \longrightarrow M_2 + Y_{max}^+$$
$$X_2^- \longrightarrow M_1 + Y_{min}^-$$
$$M_1 + M_2 \longrightarrow Y_{max}^- + Y_{min}^+$$

**Figure 3** Minimal *ReLU* (left), *abs* (middle) and *minmax* (right) CRNs. (left) The *ReLU* CRN produces $x^+(0)$ amount of $M$ and $Y^+$ by the first reaction. The second reaction produces $min(x^+(0), x^-(0))$ amount of $Y^-$. Thus, the amount of output produced is: $y = y^+ - y^- = x^+(0) - min(x^+(0), x^-(0))$ which can be shown to be equal to $ReLU(x^+(0) - x^+(0)) = ReLU(x)$. (middle) The *abs* CRN produces $x^+(0)$ amount of $C$ and $E$ by the first and second reactions, respectively, $x^+(0) + x^-(0)$ amount of $Y^+$, and $2min(x^+(0), x^-(0))$ amount of $Y^-$. Thus, $y = x^+(0) + x^-(0) - 2min(x^+(0), x^-(0)) = abs(x^+(0), x^-(0)) = abs(x)$.

**Dual-rail convention.** Concentrations of species are always non-negative, making it impossible to represent negative values directly. However, there is a natural way to extend computation semantics to negative values. Instead of using a single species to represent a value, in dual-rail convention a value is represented by a difference between a two species (e.g., the output value is equal to the concentration of species $Y^+$ minus that of $Y^-$).

An additional requirement for CRN modules is to be composable, in the sense that the output of one can be input to another. Note, for example, that the *max* system (Figure 1) is not composable because the downstream module might consume some amount of $Y$ before it is consumed in its interaction with $K$ (last reaction). Composability can be ensured if the output species are never consumed [9, 14, 52]. Note that consuming $Y^+$ is logically equivalent to producing $Y^-$ (and vise versa for $Y^-$), and thus we restrict dual-rail computation in this way without losing expressibility.

**Smallest *ReLU* CRN.** Using the above described procedure we run experiments for finding the smallest CRN computing *ReLU* (rectified linear unit) function. We confirm that the CRN introduced in [58], which is shown in Figure 3, is indeed the smallest. Note that CRNs were already enumerated when searching for *max*, and that was no need to re-enumerate them as they were saved on disk.

Our analysis shows that the *ReLU* CRN is the smallest in the sense that there is no other CRN computing this function with fewer than 2 reactions or 5 species. In Appendix D we argue that our enumeration in Table 1 is sufficient to ensure that 5 species are necessary no matter how many reactions are allowed.

**Smallest *abs* CRN.** We conducted a similar experiment for finding the smallest CRN computing the absolute value function, finding CRN shown in Figure 3.

**Smallest *minmax* CRN.** *Minmax* CRN accepts two inputs and has two outputs, where one output computes *max*, and other output computes *min* of the inputs. Since species are in dual-rail form, there is 4 input and 4 output species. Thus, for *minmax* search we enumerated CRNs that have at least 8 species, where at least 4 species only appear as products (output species candidates), and at least 4 species which do not appear only as products (input species candidates). We have further restricted the CRNs to have a total of at most 16 reactants and products over all reactions. Enumeration results with those constraints are shown in Table 2 (isomorphic breaking is imperfect in this case as well). We discovered the

**Table 2** Number of enumerated feed-forward, non-competitive CRNs with at least two dual-rail inputs (4 actual species) and two outputs (4 actual species). Star (*) denotes that the scope has been partially enumerated.

| | 2 *Reactions* | | 3 *Reactions* | | 4 *Reactions* | | 5 *Reactions* | |
|---|---|---|---|---|---|---|---|---|
| 8 *Species* | 1 | 00:00:00 | 1,176 | 00:00:03 | 67,323 | 00:03:09 | 0 | 00:00:00 |
| 9 *Species* | 0 | 00:00:00 | 1,073 | 00:00:03 | 223,775 | 00:12:48 | 2,439,310 | 13:31:19 |
| 10 *Species* | 0 | 00:00:00 | 385 | 00:00:02 | 328,397 | 00:19:30 | 4,669,000* | 47:39:39 |

**Table 3** Number of enumerated seesaw reactions with different number of domains and reactions, and up to 20 distinct species.

| | 1 *Reaction* | | 2 *Reactions* | | 3 *Reactions* | | 4 *Reactions* | | 5 *Reactions* | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 *Domain* | 1 | 00:00:00 | 0 | 00:00:00 | 0 | 00:00:00 | 0 | 00:00:00 | 0 | 00:00:00 |
| 2 *Domains* | 1 | 00:00:00 | 4 | 00:00:00 | 0 | 00:00:00 | 2 | 00:00:01 | 1 | 00:00:03 |
| 3 *Domains* | 1 | 00:00:00 | 5 | 00:00:00 | 15 | 00:00:01 | 13 | 00:00:05 | 14 | 00:00:17 |
| 4 *Domains* | 0 | 00:00:00 | 9 | 00:00:01 | 33 | 00:00:02 | 92 | 00:00:18 | 121 | 00:01:58 |
| 5 *Domains* | 0 | 00:00:00 | 4 | 00:00:00 | 55 | 00:00:04 | 243 | 00:00:48 | 705 | 00:10:16 |
| 6 *Domains* | 0 | 00:00:00 | 1 | 00:00:00 | 43 | 00:00:10 | 436 | 00:06:40 | 2027 | 03:01:06 |

minimal *minmax* CRN, which is shown in Figure 3. We performed several optimizations to speed up the analysis phase which are described in Appendix E.

**Seesaw enumeration.** We enumerated all nonisomorphic seesaw CRNs up to specified bounds on the number of domains and reactions. Table 3 shows the number of enumerated CRNs restricted to 1-5 reactions, 1-6 domains, and up to 20 species. Since 5 seesaw reactions can have at most 20 distinct species this includes all possible seesaw CRNs in the scope of 1-5 reactions. For seesaw networks, we define isomorphic CRNs as those that can be obtained by: (a) swapping domain names, (b) changing order of reactants or products, (c) changing order of reactions, (d) swapping reactants with products (follows from the reversibility of seesaw reactions).

In order to check for isomorphisms while enumerating seesaw CRNs, we maintain a set of previously enumerated CRNs and all their isomorphisms. If a newly enumerated CRN is not found in the current set, we create the isomorphic class of the CRN by making all permutations of the CRN, and adding them to the set. Permutations are done only with respect to domains. Permuting the order of reactants and products, as well as swapping reactants and products, is not needed as we follow the convention of enumerating CRNs in a form $S_{??} + LG_{??} \leftrightarrow S_{??} + RG_{??}$. Permuting the order of reactions is not needed, as the set of CRNs is preserved as a hash table where a custom-made hash function is used for CRNs (a same hash value is returned for a CRN irrespective of the order of reactions). The isomorphic breaking is implemented as a post-processing step in Java. The run-times reported in Table 3 include both generation and isomorphic breaking times.

Note that we require that the CRN corresponding to a seesaw system contain all reactions that can occur. For illustration, we analyze seesaw CRNs with 2 domains and 1 reaction. Due to the reversibility of seesaw reactions we can limit our analysis to CRNs that have a left gate on the left hand side; thus our CRN will be of the form $S_{??} + LG_{??} \leftrightarrow S_{??} + RG_{??}$, where ? represent domains to be filled in. We denote two available domains with $a$ and $b$,

and we enforce that both domains are used in a CRN. The possible combinations for the domains of the first strand are $\{aa, ab, ba, bb\}$, where we can remove cases starting with $b$ as they are symmetrical. Choosing $S_{aa}$ as a first strand, the only option for left gate is $LG_{ab}$ as we have to use two domains and left domain of $LG$ must match right domain of $S$. This leads to a CRN: $S_{aa} + LG_{ab} \leftrightarrow S_{ab} + RG_{aa}$. Note that this CRN is not a valid one, as in this case $S_{aa}$ and $RG_{aa}$ can also interact creating additional reaction. Another option for the strand is $S_{ab}$, in which case there are two options for left gate $LG_{bb}$ and $LG_{ba}$. In a case of $LG_{bb}$ reaction is following: $S_{ab} + LG_{bb} \leftrightarrow S_{bb} + RG_{ab}$. This is also not a valid CRN since $S_{bb}$ and $LG_{bb}$ can interact creating additional reaction. The final option is $S_{ab} + LG_{ba} \leftrightarrow S_{ba} + RG_{ab}$, which is only valid seesaw CRN in a case of 2 domains and 1 reaction; thus Table 3 shows count 1 for seesaw CRNs with 2 domains and 1 reaction.

Similarly, note that there are 0 CRNs with 2 domains and 3 reactions, but there are 2 with 2 domains and 4 reactions. This is due to the fact that all 3 reaction CRNs with 2 domains have some other species that can also interact producing additional (spurious) reaction. A curious reader can check that removing any reaction from 4 reaction 2 domain seesaw CRNs (Table 4) will leave some species that can interact creating the fourth reaction.

**Table 4** Seesaw CRNs with 2 domains and 4 reactions.

$$S_{aa} + LG_{aa} \longleftrightarrow S_{aa} + RG_{aa}$$
$$S_{ba} + LG_{aa} \longleftrightarrow S_{aa} + RG_{ba}$$
$$S_{bb} + LG_{ba} \longleftrightarrow S_{ba} + RG_{bb}$$
$$S_{bb} + LG_{bb} \longleftrightarrow S_{bb} + RG_{bb}$$

$$S_{ab} + LG_{ba} \longleftrightarrow S_{ba} + RG_{ab}$$
$$S_{bb} + LG_{ba} \longleftrightarrow S_{ba} + RG_{bb}$$
$$S_{ab} + LG_{bb} \longleftrightarrow S_{bb} + RG_{ab}$$
$$S_{bb} + LG_{bb} \longleftrightarrow S_{bb} + RG_{bb}$$

## 6 Related Work

**CRN Enumeration.** Deckard et al. [18] developed an online library of reaction networks, which was extended [3] to catalog reactions of several classes. These approaches generate non-isomorphic bipartite graphs (two types of vertices for species and reactions) with undirected edges relying on Nauty library [45]. Each such constructed graph is then reified as multiple CRN instances. Recent generalization of this work gives the first complete count of all 2-species bimolecular CRNs, and counts for other classes of CRNs such as mass-conserving and reversible [56]. Rather than focusing on removing all isomorphisms and generating exact counts of non-isomorphic CRNs in each class, our work allows the user to flexibly specify and analyze structural properties of CRNs of interest (enabling direct generation of CRNs following the structure). For example, it is not clear how to encode molecular structure (such as we do for seesaw networks) using graph-based models.

**Minimal Systems with Desired Behavior.** Complementary to CRN enumeration, previous work also tackled the problem of finding minimal CRNs respecting some desired properties or exhibiting certain behavior. Wilhelm [62] discovers the smallest elementary CRN with bistability. Wilhelm and Heinrich [63] similarly detect the smallest CRN with Hopf bifurcation. In comparison with this line of work, our paper presents a more general framework that allows specifying structure and properties, including different functions, of CRNs to be explored.

Recent work due to Murphy et al [47] is close to ours in spirit, but focuses on discrete-state stochastic systems (integer molecular counts of the species), rate-dependent reactions, and does not guarantee that discovered CRNs are minimal. Cardelli et al [8] take a program synthesis approach to generate CRNs that follow properties provided by a certain "sketch" language (i.e., a template) using SMT solvers on the back end [4, 17].

**Computational power of CRNs.** Much ongoing work has explored computational power of CRNs [31,43,51,59]. It is shown how to map complex computation to CRNs, such as mapping polynomials to chemical reactions, mapping discrete algorithms, and even defining a high-level imperative languages that map to CRNs. We believe that by exploring CRNs bottom up, we may found answers of what the appropriate (more efficient) high-level primitives are to be used for implementing such high-level functionality.

## 7 Conclusion

We introduced the use of Alloy, a framework for modeling and analyzing structural constraints and behavior in software systems, to enumerate CRNs with declaratively specified properties. We showed how this framework can enumerate CRNs with a variety of structural constraints including biologically motivated catalytic networks and metabolic networks, and seesaw networks motivated by DNA nanotechnology. We also used the framework to explore analog function computation in rate-independent CRNs. We applied our approach in a case-study to find the smallest CRNs computing the *max*, *minmax*, *abs* and *ReLU* functions in a natural subclass of rate-independent CRNs where rate-independence follows from structural network properties.

There remain a number of open questions that motivate future research directions. An important area of optimization is improving the run-time of the Alloy enumeration. Can we optimize the isomorphic breaking process to eliminate all isomorphisms? For improved efficiency and ease of use, do we need to rely on a separate tool like Mathematica to determine whether a given CRN computes the desired function, or can the necessary functionality be performed in Alloy alone? Finally, it remains to be seen how easily the techniques developed in this paper could be applied to rate-dependent computation.

## References


1 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.

2 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

3 Murad Banaji. Counting chemical reaction networks with NAUTY. *arXiv preprint arXiv:1705.10820*, 2017.

4 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, 2011.

5 Gilles Bernot, Jean-Paul Comet, Adrien Richard, and Janine Guespin. Application of formal methods to biological regulatory networks: extending thomas' asynchronous logical approach with temporal logic. *Journal of theoretical biology*, 2004.

6 Luca Cardelli. Strand algebras for DNA computing. *Natural Computing*, 10(1):407–428, 2011.

7 Luca Cardelli. Morphisms of reaction networks that couple structure to function. *BMC systems biology*, 8(1):84, 2014.

8 Luca Cardelli, Milan Češka, Martin Fränzle, Marta Kwiatkowska, Luca Laurenti, Nicola Paoletti, and Max Whitby. Syntax-guided optimal synthesis for chemical reaction networks. In *CAV*, 2017.

9 Cameron Chalk, Niels Kornerup, Wyatt Reeves, and David Soloveichik. Composable rate-independent computation in continuous chemical reaction networks. In *CMSB*, pages 256–273. Springer, 2018.

10 Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13(4):517–534, 2014.

**11** Ho-Lin Chen, David Doty, and David Soloveichik. Rate-independent computation in continuous chemical reaction networks. In *Proceedings of the 5th conference on Innovations in theoretical computer science*, pages 313–326. ACM, 2014.

**12** Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature nanotechnology*, 8(10):755, 2013.

**13** Kevin M Cherry and Lulu Qian. Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. *Nature*, 559(7714):370, 2018.

**14** Ben Chugg, Anne Condon, and Hooman Hashemi. Output-oblivious stochastic chemical reaction networks. *arXiv preprint arXiv:1812.04401*, 2018.

**15** Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model Checking*. MIT Press, 2018.

**16** CRNs Exposed Github Page. URL: `https://github.com/marko-vasic/crnsExposed`.

**17** Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

**18** Anastasia C Deckard, Frank T Bergmann, and Herbert M Sauro. Enumeration and online library of mass-action reaction networks. *arXiv preprint arXiv:0901.3067*, 2009.

**19** Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In *ISSTA*, 2006.

**20** Niklas Een and Niklas Sorensson. An extensible SAT-solver. In *SAT03*, Santa Margherita Ligure, Italy, 2003.

**21** Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. DynAlloy: Upgrading Alloy with actions. In *ICSE*, 2005.

**22** Juan P. Galeotti, Nicolás Rosner, Carlos G. López Pombo, and Marcelo F. Frias. TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. *TSE*, 2013.

**23** Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

**24** Mirco Giacobbe, Călin C Guet, Ashutosh Gupta, Thomas A Henzinger, Tiago Paixão, and Tatjana Petrov. Model checking gene regulatory networks. In *TACAS*, 2015.

**25** Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011.

**26** Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *CAV*, pages 476–479. Springer, 1997.

**27** Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 2000.

**28** Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

**29** John Heath, Marta Kwiatkowska, Gethin Norman, David Parker, and Oksana Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 2008.

**30** Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

**31** De-An Huang, Jie-Hong R. Jiang, Ruei-Yang Huang, and Chi-Yun Cheng. Compiling program control flows into biochemical reactions. In *Proceedings of the International Conference on Computer-Aided Design*, pages 361–368, 2012.

**32** Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV*, 2017.

33 Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

34 Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *TACS*, 2001.

35 Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *International Conference on Software Engineering*, Limerick, Ireland, June 2000.

36 Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA*, 2000.

37 Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. Multi-representational security analysis. In *FSE*, 2016.

38 Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *ACM SIGPLAN Notices*, volume 37, pages 231–245. ACM, 2002.

39 Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Santa Margherita Ligure, Italy, May 2003.

40 Matthew R Lakin, David Parker, Luca Cardelli, Marta Kwiatkowska, and Andrew Phillips. Design and analysis of DNA strand displacement devices using probabilistic model checking. *Journal of the Royal Society Interface*, 2012.

41 Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

42 Tong Ihn Lee, Nicola J Rinaldi, François Robert, Duncan T Odom, Ziv Bar-Joseph, Georg K Gerber, Nancy M Hannett, Christopher T Harbison, Craig M Thompson, Itamar Simon, et al. Transcriptional regulatory networks in saccharomyces cerevisiae. *Science*, 298(5594):799–804, 2002.

43 Marcelo O Magnasco. Chemical kinetics is Turing universal. *Physical Review Letters*, 78(6):1190, 1997.

44 Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, pages 22–31, 2001.

45 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 2014.

46 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC)*, 2001.

47 Niall Murphy, Rasmus Petersen, Andrew Phillips, Boyan Yordanov, and Neil Dalchau. Synthesizing and tuning stochastic chemical reaction networks with specified behaviours. *Journal of The Royal Society Interface*, 15(145):20180283, 2018.

48 Jason Ptacek, Geeta Devgan, Gregory Michaud, Heng Zhu, Xiaowei Zhu, Joseph Fasolo, Hong Guo, Ghil Jona, Ashton Breitkreutz, Richelle Sopko, et al. Global analysis of protein phosphorylation in yeast. *Nature*, 438(7068):679, 2005.

49 Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

50 Lulu Qian and Erik Winfree. A simple DNA gate motif for synthesizing large-scale circuits. *Journal of the Royal Society Interface*, 8(62):1281–1297, 2011.

51 Sayed Ahmad Salehi, Keshab K. Parhi, and Marc D. Riedel. Chemical reaction networks for computing polynomials. *ACS Synthetic Biology*, 6(1):76–83, 2017.

52 Eric E Severson, David Haley, and David Doty. Composable computation in discrete chemical reaction networks. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 14–23, 2019.

53 Shalin Shah, Jasmine Wee, Tianqi Song, Luis Ceze, Karin Strauss, Yuan-Jyue Chen, and John Reif. Using strand displacing polymerase to program chemical reaction networks. *Journal of the American Chemical Society*, 2020.

54 Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.

55 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

56 Carlo Spaccasassi, Boyan Yordanov, Andrew Phillips, and Neil Dalchau. Fast enumeration of non-isomorphic chemical reaction networks. In *CMSB*, pages 224–247. Springer, 2019.

57 Niranjan Srinivas, James Parkin, Georg Seelig, Erik Winfree, and David Soloveichik. Enzyme-free nucleic acid dynamical systems. *Science*, 358(6369):eaal2052, 2017.

58 Marko Vasic, Cameron Chalk, Sarfraz Khurshid, and David Soloveichik. Deep Molecular Programming: A Natural Implementation of Binary-Weight ReLU Neural Networks. In *International Conference on Machine Learning*, 2020.

59 Marko Vasic, David Soloveichik, and Sarfraz Khurshid. CRN++: molecular programming language. In *International Conference on DNA Computing and Molecular Programming*, pages 1–18. Springer, 2018.

60 Vito Volterra. *Variazioni e fluttuazioni del numero d'individui in specie animali conviventi*. C. Ferrari, 1927.

61 Qinsi Wang, Paolo Zuliani, Soonho Kong, Sicun Gao, and Edmund M Clarke. Sreach: A probabilistic bounded delta-reachability analyzer for stochastic hybrid systems. In *CMSB*, 2015.

62 Thomas Wilhelm. The smallest chemical reaction system with bistability. *BMC systems biology*, 3(1):90, 2009.

63 Thomas Wilhelm and Reinhart Heinrich. Smallest chemical reaction system with hopf bifurcation. *Journal of mathematical chemistry*, 17(1):1–14, 1995.

64 David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature chemistry*, 3(2):103, 2011.


## A Proof of Rate Independence

In this section we develop an argument that the class of feed-forward, non-competitive CRNs as defined in the main text is rate-independent. For simplicity, we base our argument on the discrete CRN model, in which concentrations are integer molecular counts, reactions are discrete events (firings), and rate-independence corresponds to behaving correctly no matter what order the reactions occur in [10]. The continuous model is usually taken as an approximation of the discrete model.

Note that when we say that a species $S$ is consumed by a reaction, we mean that it appears with negative net stoichiometry in the reaction. So we would not say that a catalyst is consumed. We define produced similarly. We say configuration $d$ is reachable from $c$ if there is a sequence of reactions that can fire to get from $c$ to $d$.

In the main text, we define non-competitive as follows: if a species is consumed in a reaction then it cannot appear as a reactant somewhere else. Feed-forward is defined as follows: there exists a total ordering on the reactions such that no reaction consumes a species produced by a reaction later in the ordering. We also require that all reactions consume some species (boundedness condition).

Here we show that the feed-forward condition combined with boundedness implies that the CRN will always reach a static equilibrium. (A static equilibrium is one where no reaction can fire.) We then show that adding the non-competitive condition implies that the CRN always reaches the same static equilibrium independent of the order in which the reactions happen to occur.

*The CRN always reaches some static equilibrium*: If not then there is a set of reactions that can fire infinitely often. Choose the earliest (according to the ordering) reaction in this set. It must consume some $S$ by boundedness. But by feed-forwardness, $S$ can only be produced earlier in the ordering. Which means that the reactions that net produce $S$ can only fire finite many times (they are not in this set). This is a contradiction.

*The CRN always reaches the same static equilibrium*: Toward a contradiction, suppose two different static equilibria $c$ and $d$ are reachable. Let $p$ be the path to $c$ and $q$ be the path to $d$. Without loss of generality there are reactions that fire fewer times in $p$ than in $q$. Let $R$ be the reaction among these that comes earliest in the ordering. So compared to $q$, $p$ has at least as many firings of reactions earlier in the ordering than $R$. By non-competitiveness, no other reaction consumes the reactants of $R$. Let $S$ be a reactant of $R$. Consider two cases: (1) $S$ is consumed in $R$. By feed-forwardness, $S$ must be produced in a reaction earlier in the ordering than $R$. This means that the reactions producing $S$ fire at least as much in $p$ as in $q$. Since $R$ fired fewer times in $p$ than in $q$, there are some of $S$ left in $c$. (2) $S$ is not consumed in $R$ (it acts as a catalyst). By the argument below, since $R$ fires in $q$ at least once, $R$ fires in $p$ at least once. Thus $S$ is present in $c$. Combining (1) and (2), we have that $R$ can fire in $c$, which contradicts the assumption that $c$ is a static equilibrium.

*There are no reactions that can fire on the path toward one static equilibrium but not fire on the path to another*: Toward a contradiction, suppose two different static equilibria $c$ and $d$ are reachable. Let $p$ be the path to $c$ and $q$ be the path to $d$. Let $\Omega$ be the set of reactions that fire in $q$ but not in $p$. Let $R$ be the reaction in $\Omega$ that occurs first (in time) in $q$. Its reactants must be either inputs or produced outside of $\Omega$ since $R$ is the first reaction in $\Omega$ that fired in $q$. By non-competitiveness, the reactants of $R$ cannot be consumed in any reaction other than $R$. So it must be possible to fire $R$ at the end of $p$, which contradicts the assumption that $p$ is a static equilibrium.

## B Background: Alloy

The Alloy modeling language is a first-order logic with transitive closure [33]. The Alloy analyzer is a fully automatic tool for *scope-bounded* analysis of properties of Alloy models [35]. Given an Alloy model and a *scope*, i.e., a bound on the universe of discourse, the analyzer translates the Alloy model to a propositional satisfiability (SAT) formula and invokes an off-the-shelf SAT solver [20] to analyze the model.

An Alloy model consists of a set of paragraphs where each paragraph declares some typed sets or relations, defines some logical constraints, or defines a command that informs the analyzer of the analysis to perform. Each command defines a constraint solving problem. and each solution to the problem defines an Alloy *instance*, i.e., a valuation of the sets and relations declared in the model such that the constraints with respect to the command are satisfied. The analyzer supports instance enumeration using incremental SAT solvers [20, 46]. In addition, the analyzer supports *symmetry breaking* and adds symmetry breaking predicates [54] to the original formula, which allows the backend SAT solvers to more effectively prune their search, and when enumerating solutions, create fewer solutions [39]. The analyzer's default symmetry breaking does not guarantee removal of all isomorphisms but is quite effective in practice.

## C Autocatalytic Reactions

Similarly to catalytic reactions we model autocatalytic (Listing 8). Autocatalytic reactions add a requirement that in addition to existence of a catalyst species, the catalyst converts the other species into itself, for example: $X + Y \to Y + Y$.

**Listing 8** Autocatalytic reactions.

```
module autocatalytic
open elementary
pred Autocatalytic[] { Elementary[] and all r: Reaction | AutocatalyticReaction[r] }
pred AutocatalyticReaction[r: Reaction] {
    some elems[r.reactants] & elems[r.products]
    eq[#r.products, 2] and eq[#elems[r.products], 1] }
```

## D ReLU Minimality

In this section we argue that our enumeration in Table 1 is sufficient to ensure that 5 species are necessary for computing $ReLU$ no matter how many reactions are allowed.

Because with 4 species there are at most 2 different reactions possible (which we enumerate). Consider the $ReLU$ CRN with 4 species. This CRN must consist of 2 input species ($X^+$ and $X^-$) and 2 output species ($Y^+$ and $Y^-$), which we require to be distinct. Further, the output species have to appear only as products. Thus, only species $X^+$ and $X^-$ can appear as reactants. Due to the requirement that every reaction has to net consume some species (Listing 7), and that different reactions have to consume different species (non-competitiveness), it follows that the CRN can have at maximum 2 reactions, one net consuming $X^+$, and other $X^+$ species. Considering that our technique did not discover any $ReLU$ CRN with 2 reactions and 4 species, we conclude that there is no $ReLU$ computing CRN with 4 species.

## E Optimizing Analysis

In this section we explain how we optimize the analysis phase of search for *minmax* CRN.

The optimization is done by including tests. Instead of invoking *FindInstance* SMT solver for every combination of inputs and outputs, we construct a set of concrete test cases. If a test case fails we immediately discard that combination and move to the next one. This optimization improved analysis from 75s to 7.3s measured on the discovered *minmax* CRN. Furthermore from equality $|max(a,b)| + |min(a,b)| = min(|a|,|b|) + max(|a|,|b|)$, we first checked for CRNs that sattisfy this condition (using tests and FindInstance), and only run the check whether output species compute min and max on those. Checking for the above equality speeded up analysis becase the equality does not depend on the order of output species $y1$ and $y2$, thus reducing number of input output combinations that need to be tried. After implementing this additional optimization step analysis time went down to 0.75s measured on the discovered *minmax* CRN. The optimizations made it feasible to discover the *minmax* CRN.

## F Symmetry breaking

This section shows our Alloy model for symmetry breaking of CRNs (Listing 9).

The Alloy analyzer during its translation from Alloy to propositional formulas automatically adds to the propositional formulas *symmetry breaking* predicates, which reduce the number of isomorphic solutions [54]. However, this automatic support is not practical for breaking all isomorphisms since there is a delicate trade-off between the complexity of the predicates that are added and the time it takes for the back-end solvers to handle them.

We follow a more effective approach where additional constraints *in Alloy* are mechanically added directly to the Alloy model [39]. The key idea is to define a linear order on the atoms and require that any solution when scanned in a pre-defined manner contains the atoms in

conformance with the linear order. The approach breaks all symmetries for rooted, edge-labeled graphs. However, CRNs represent a more complex structure and the approach does not guarantee breaking all symmetries. Nonetheless, it removes many isomorphic solutions and provides us a practical tool for exploring CRNs.

Note that the symmetry breaking is focused on a case of elementary CRNs as those CRNs are our focus group (all of our inherited CRN models are subclass of elementary).

**Listing 9** Alloy modeling of CRN symmetry breaking.

```
module symmetry

open elementary

open util/ordering[Species] as Sordering
open util/ordering[Reaction] as Rordering

pred CheckFirstReaction {
  let first = Rordering/first,
      r1 = 0.(first.reactants), r2 = 1.(first.reactants),
      p1 = 0.(first.products), p2 = 1.(first.products)
  {
    r1 = Sordering/first
    r2 in r1 + r1.next
    p1 in r1 + r2 + (r1 + r2).next
    p2 in r1 + r2 + p1 + (r1 + r2 + p1).next
  }
}

pred CheckNonFirstReaction() {
  all r: Reaction - Rordering/first {
    let prevRxns = Rordering/prevs[r],
        prevSpecies = Int.(prevRxns.reactants + prevRxns.products),
        r1 = 0.(r.reactants), r2 = 1.(r.reactants),
        p1 = 0.(r.products), p2 = 1.(r.products)
    {
      r1 in prevSpecies + prevSpecies.next
      r2 in prevSpecies + r1 + (prevSpecies + r1).next
      p1 in prevSpecies + r1 + r2 + (prevSpecies + r1 + r2).next
      p2 in prevSpecies + r1 + r2 + p1 + (prevSpecies + r1 + r2 + p1).next
    }
  }
}

pred OrderReactionsBySize() {
  all disj r1, r2 : Reaction {
    Rordering/lt[r1, r2] implies {
      lt[#r1.reactants, #r2.reactants]
      or (eq[#r1.reactants, #r2.reactants]
           and lte[#r1.products, #r2.products])
    }
  }
}

pred ReactionsSameSize[r1, r2: Reaction] {
  eq[#r1.reactants, #r2.reactants]
    and eq[#r1.products, #r2.products]
}

pred CheckLexicographic() {
  all r: Reaction - Rordering/first {
    let p = r.prev,
    rr1 = 0.(r.reactants), rr2 = 1.(r.reactants), rp1 = 0.(r.products), rp2 = 1.(r.products),
    pr1 = 0.(p.reactants), pr2 = 1.(p.reactants), pp1 = 0.(p.products), pp2 = 1.(p.products)
    {
       ReactionsSameSize[r, p] implies {
        // DO only if sizes are the same assuming the size constraing.
        rr1 in pr1.*next
        rr1 = pr1 implies (no pr2 or rr2 in pr2.*next)
        (rr1 = pr1 and rr2 = pr2) implies (rp1 in pp1.*next)
        (rr1 = pr1 and rr2 = pr2 and rp1 = pp1) implies (no pp2 or rp2 in pp2.*next)
       }
    }
  }
```

```
  all r: Reaction {
    let r1 = 0.(r.reactants), r2 = 1.(r.reactants), p1 = 0.(r.products), p2 = 1.(r.products)
    {
      some r1 and some r2 implies Sordering/lte[r1, r2]
      some p1 and some p2 implies Sordering/lte[p1, p2]
    }
  }
}

pred SymmetryBreaking {
  Elementary
  CheckFirstReaction
  CheckNonFirstReaction
  OrderReactionsBySize
  CheckLexicographic
}
```

# Population-Induced Phase Transitions and the Verification of Chemical Reaction Networks

**James I. Lathrop**
Iowa State University, Ames, IA, USA
jil@iastate.edu

**Jack H. Lutz**
Iowa State University, Ames, IA, USA
lutz@iastate.edu

**Robyn R. Lutz**
Iowa State University, Ames, IA, USA
rlutz@iastate.edu

**Hugh D. Potter**
Iowa State University, Ames, IA, USA
hdpotter@iastate.edu

**Matthew R. Riley**
Iowa State University, Ames, IA, USA
mrriley@iastate.edu

## Abstract

We show that very simple molecular systems, modeled as chemical reaction networks, can have behaviors that exhibit dramatic phase transitions at certain population thresholds. Moreover, the magnitudes of these thresholds can thwart attempts to use simulation, model checking, or approximation by differential equations to formally verify the behaviors of such systems at realistic populations. We show how formal theorem provers can successfully verify some such systems at populations where other verification methods fail.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models

**Keywords and phrases** chemical reaction networks, molecular programming, phase transitions, population protocols, verification

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.5

**Funding** This research was supported in part by National Science Foundation grants 1545028, 1900716, and 1909688.

**Acknowledgements** The second and third authors thank Erik Winfree for his hospitality while they did part of this work during a 2020 sabbatical visit at Caltech. We thank Neil Lutz for technical assistance. We thank the reviewers for detailed suggestions that have improved our exposition, both here and in an expansion of this paper in preparation.

## 1 Introduction

Chemical reaction networks, mathematical abstractions similar to Petri nets, are used as a programming language to specify the dynamic behaviors of engineered molecular systems. Existing software can compile chemical reaction networks into DNA strand displacement systems that simulate them with growing generality and precision [52, 14, 6, 53]. Programming is a challenging discipline in any case, but this is especially true of molecular programming, because chemical reaction networks – in addition to being Turing universal [51, 18, 21] and hence subject to all the uncomputable aspects of sequential, imperative programs–are, like the systems that they specify, distributed, asynchronous, and probabilistic. Since many envisioned


applications of molecular programming will be safety critical [54, 55, 19, 33, 32, 50, 44], programmers thus seek to create chemical reaction networks that can be *verified* to correctly carry out their design intent.

One principle that is sometimes used in chemical reaction network design is the *small population heuristic* [31, 11, 20]. The idea here is to verify various stages of a design by model checking or software simulation to ferret out bugs in the design prior to laboratory experimentation or deployment. Since the number of states of a molecular system is typically much larger than its population (the number of molecules present), and since molecular systems typically have very large populations, this model checking or simulation can usually only be carried out on populations that are far smaller than those of the intended molecular systems. It is nevertheless reasonable to hope that, if a system is going to consist of a very large number of "devices" of various sorts, then any unforeseen errors in these devices' interactions will manifest themselves even with very small populations of each device. It is this reasonable hope that is the underlying premise of the small population heuristic. (Note that the small population heuristic can be regarded as a molecular version of the small scope hypothesis [24].)

The question that we address here is whether real molecular systems can thwart the small population heuristic. That is, can a real molecular system behave very differently at large populations than at small populations? If so, *how sensitive* can its behavior be to its population, and *how simple a mechanism* can achieve such sensitivity?

In order to ensure that we are only investigating population effects, we focus our attention on chemical reaction networks that are *population protocols* in the sense that their populations remain constant throughout their operations. If we have such a chemical reaction network, and if we vary its initial population *and nothing else*, then we are assured that any resulting variations of behavior are due solely to the differing populations.

In this paper we show that very simple chemical reaction networks can be very sensitive to their own populations. In fact, they can exhibit *population-induced phase transitions*, behaving one way below a threshold population and behaving very differently above that threshold. After reviewing chemical reaction networks in Section 2, we present in Section 3 a chemical reaction network $\mathbf{N}_1$, and we prove that $\mathbf{N}_1$ exhibits a population-induced phase transition in the following sense. There are two parameters, $m$ and $n$, in the construction. For this discussion, we may take $m = 34$ and $n = 67$, but the construction is general. There are $n + 1$ reactions among $n + 2$ species (molecule types) in $\mathbf{N}_1$. A species $Z_0$ is given an initial population $p$, and all other species counts are initially 0. Each reaction of $\mathbf{N}_1$ has two reactants and two products, so the total population of $\mathbf{N}_1$ is $p$ at all times. There are in $\mathbf{N}_1$ two distinguished species, $B$ and $R$. These "blue" and "red" species are abstract stand-ins for two different behaviors of $\mathbf{N}_1$. Our construction exploits the inherent nonlinearity of chemical kinetics to ensure that, if $p < 2^m$, then $\mathbf{N}_1$ terminates with essentially all its population blue, while if $p \geq 2^m$, then $\mathbf{N}_1$ terminates with essentially all its population red. Thus $\mathbf{N}_1$ exhibits a sharp phase transition at the population threshold $p = 2^m$.

Our construction is very simple. The chemical reaction network $\mathbf{N}_1$ changes its behavior at the threshold $p = 2^m$ by merely computing successive bits of $p$, starting at the least significant bit. This mechanism is so simple that it could be hidden, by accident or by malice, in a larger chemical reaction network. Moreover, for suitable values of $m$ (e.g., $m = 34$, so that the threshold $p = 2^m$ is roughly $1.7 \times 10^{10}$),

**(1)** any attempt to model-check or simulate $\mathbf{N}_1$ will perforce use a population much less than the threshold and conclude that $\mathbf{N}_1$ will always turn blue; while

**(2)** any realistic wet-lab molecular implementation of $\mathbf{N}_1$ will have a population greater than the threshold and thus turn red.

If the behaviors represented by blue and red here are a desired, "good" behavior of $\mathbf{N}_1$ (or of a network containing $\mathbf{N}_1$) and an undesired, "bad" behavior of this network, respectively, then the possibility of such a phase transition is a serious challenge to verifying the correct behavior of the chemical reaction network. Simply put, this is a context in which the small population heuristic can lead us astray.

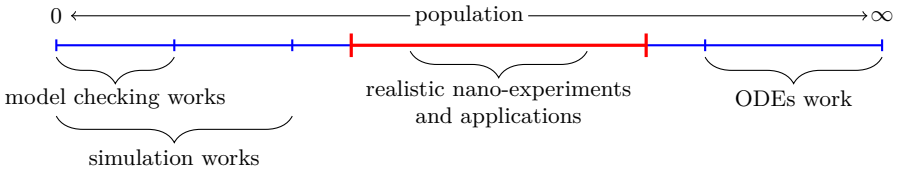



**Figure 1** Scales at which different verification methods (simulation, model checking, and ODE's) work. The gap in the middle shows the scale at which none of these methods will catch the "produce blue" behavior of the system design. This gap is problematic because it is the scale of realistic programmed molecular systems. We show in Section 5.4 how such systems can be verified using automated theorem proving.

There is a dual *large population heuristic* that is used even more often than the small population heuristic. A theorem of Kurtz [27, 2, 3] draws a connection between the behavior of a *stochastic* chemical reaction network (the type of chemical reaction network used in our work here and in most of molecular programming) at large populations and the behavior of a *deterministic* chemical reaction network, which is governed by a system of ordinary differential equations. Kurtz's theorem involves several preconditions and caveats, and it does not always transparently equate stochastic and deterministic behavior. When it does apply, however, we can use a mathematical software package to numerically solve the deterministic system and thereby understand the behavior of the stochastic chemical reaction network at sufficiently large populations.

In Section 4 we add a single reaction to the chemical reaction network $\mathbf{N}_1$, creating a chemical reaction network $\mathbf{N}_2$ that we prove (in Theorem 4.6) to exhibit two *coupled population-induced phase transitions* in the following sense. If $p < 2^m$ or $p \geq 2^n$, then $\mathbf{N}_2$ terminates with essentially all its population blue, while if $2^m \leq p < 2^n$, then $\mathbf{N}_2$ terminates with essentially all its population red. Thus $\mathbf{N}_2$ exhibits sharp phase transitions at the two population thresholds, $p = 2^m$ and $p = 2^n$. These phase transitions are *coupled* in that exceeding the second threshold returns the behavior of $\mathbf{N}_2$ to its behavior below the first threshold. For suitable values of $m$ and $n$ (e.g. $m = 34$ and $n = 67$ as above, so that the thresholds $p = 2^m$ and $p = 2^n$ are roughly $1.7 \times 10^{10}$ and $1.5 \times 10^{20}$), this implies (see Figure 1) that

**(1)** any attempt to model-check or simulate $\mathbf{N}_2$ will perforce use a population much less than the smaller threshold and conclude that $\mathbf{N}_2$ will always turn blue, and

**(2)** any realistic wet-lab molecular implementation of $\mathbf{N}_2$ will have a population between the two thresholds and thus turn red.

As we discuss later, when we analyze $\mathbf{N}_2$ with a numerical approach based on differential equations, we also do not observe a red outcome. The chemical reaction network $\mathbf{N}_2$ thus exemplifies a class of contexts in which the small population heuristic and the large population heuristic can both lead us astray.

We emphasize that the phase transitions in the chemical reaction networks $\mathbf{N}_1$ and $\mathbf{N}_2$ occur at thresholds in their *absolute populations*. In contrast, phase transitions in chemical reaction networks for approximate majority [4, 10, 17] occur at threshold *ratios between sub-populations*, and phase transitions in bacterial quorum sensing [36] occur at threshold *population densities*.

Section 5 discusses the consequences of our results for the verification of programmed molecular systems in some detail. Here we summarize these consequences briefly. Phase transitions are ubiquitous in natural and engineered systems [37, 45, 46, 47, 9, 43]. Our results are thus cautionary, but they should not be daunting. Fifteen years after Turing proved the undecidability of the halting problem, Rice [48, 49] proved his famous generalization stating that *every* nontrivial input/output property of programs is undecidable. Rice's theorem saves valuable time, but it has never prevented computer scientists from developing specific programs in disciplined ways that enable them to be verified. Similarly, Sections 3 and 4 give mathematical *proofs* that the chemical reaction networks $\mathbf{N}_1$ and $\mathbf{N}_2$ have the properties described above, and Section 5 describes how we have implemented such proofs in the Isabelle proof assistant [40, 41]. As molecular programming develops, simulators, model checkers, theorem provers, and other tools will evolve with it, as will disciplined scientific judgment about how and when to use such tools.

## 2 Chemical Reaction Networks

Chemical reaction networks (CRNs) are abstract models of molecular processes in well-mixed solutions. They are roughly equivalent to three models used in distributed computing, namely, Petri nets, population protocols, and vector addition systems [18]. This paper uses stochastic chemical reaction networks.

For our purposes, a (*stochastic*) *chemical reaction network* $\mathbf{N}$ consists of finitely many *reactions*, each of which has the form

$$A + B \to C + D, \tag{2.1}$$

where $A$, $B$, $C$, and $D$ (not necessarily distinct) are *species*, i.e., abstract types of molecules. Intuitively, if this reaction occurs in a solution at some time, then one $A$ and one $B$ disappear from the solution and are replaced by one $C$ and one $D$, these things happening instantaneously. A *state* of the chemical reaction network $\mathbf{N}$ with species $A_1, \ldots, A_n$ at a particular moment of time is the vector $(a_1, \ldots, a_s)$, where each $a_i$ is the nonnegative integer count of the molecules of species $A_i$ in solution at that moment. Note that we are using the so called "lower-case convention" for denoting species counts.

In the full stochastic chemical reaction network model, each reaction also has a positive real *rate constant*, and the random behavior of $\mathbf{N}$ obeys a continuous-time Markov chain derived from these rate constants. However, our results here are so robust that they hold for *any* assignment of rate constants, so we need not concern ourselves with rate constants or continuous-time Markov chains. In fact, for this paper, we can consider the reaction (2.1) to be the if-statement

$$\text{if } a > 0 \text{ and } b > 0 \text{ then } a, b, c, d := a - 1, b - 1, c + 1, d + 1 \tag{2.2}$$

(with the obvious modifications if A, B, C, and D are not distinct), where ":=" is parallel assignment. The reaction (2.1) is *enabled* in a state $q$ of $\mathbf{N}$ if $a > 0$ and $b > 0$ in $q$; otherwise, this reaction is *disabled* in $q$. A state $q$ of $\mathbf{N}$ is *terminal* if no reaction is enabled in $q$.

A *trajectory* of a chemical reaction network $\mathbf{N}$ is a sequence $\tau = (q_i \mid 0 \leq i < \ell)$ of states of $\mathbf{N}$, where $\ell \in \mathbb{Z}^+ \cup \{\infty\}$ is the *length* of $\tau$ and, for each $i \in \mathbb{N}$ with $i + 1 < \ell$, there is a reaction of $\mathbf{N}$ that is enabled in $q_i$ and whose effect, as defined by (2.2), is to change the state of $\mathbf{N}$ from $q_i$ to $q_{i+1}$. A trajectory $\tau = (q_i \mid 0 \leq i < \ell)$ is *terminal* if $\ell < \infty$ and $q_{\ell-1}$ is a terminal state of $\mathbf{N}$.

Assume for this paragraph that the context specifies an initial state $q_0$ of $\mathbf{N}$, as it does in this paper. A state $q$ of $\mathbf{N}$ is *reachable* if there is a finite trajectory $\tau = (q_i \mid 0 \leq i < \ell)$ of $\mathbf{N}$ with $q_{\ell-1} = q$. A *full trajectory* of $\mathbf{N}$ is a trajectory $\tau = (q_i \mid 0 \leq i < \ell)$ that is either terminal or infinite.

The fact that each reaction (2.1) has two *reactants* ($A$ and $B$) and two *products* ($C$ and $D$) means that $\mathbf{N}$ is a *population protocol* [5]. This condition implies that the total population of all species never changes in the course of a trajectory. If such a chemical reaction network has $s$ species and initial population $p$, its state space is thus the $(s-1)$-dimensional integer simplex

$$\Delta^{s-1}(p) = \left\{ (a_1, \ldots, a_s) \in \mathbb{N}^s \;\middle|\; \sum_{i=1}^{s} a_i = p \right\}. \tag{2.3}$$

Note that $|\Delta^{s-1}(p)| = \binom{p+s-1}{s-1}$. Of course, fewer than this many states may be reachable from a particular initial state of $\mathbf{N}$.

A full trajectory $\tau = (q_i \mid 0 \leq i < \ell)$ of a CRN $\mathbf{N}$ is (*strongly*) *fair* [30, 7] if it has the property that, for every state $q$ and reaction $\rho$ that is enabled in $q$,

$$(\exists^\infty i) q_i = q \implies (\exists^\infty j)[q_j = q \text{ and } \rho \text{ occurs at } j \text{ in } \tau], \tag{2.4}$$

where $(\exists^\infty i)$ means "there exist infinitely many $i$ such that." Note that every terminal trajectory of $\mathbf{N}$ is vacuously fair, because it does not satisfy the hypothesis of (2.4).

The stochastic kinetics of chemical reaction networks implies that, regardless of the rate constants of the reactions, for every population protocol $\mathbf{N}$ and every initial population $p$ of $\mathbf{N}$, there is a real number $\varepsilon > 0$ such that, for every state $q$ of $\mathbf{N}$ and reaction $\rho$ that is enabled in $q$, the probability that $\rho$ occurs in $q$ depends only on $q$ and is at least $\varepsilon$. This in turn implies that, with probability 1, $\mathbf{N}$ follows a fair trajectory. Hence, if $\mathbf{N}$ has a given behavior on all fair trajectories, then $\mathbf{N}$ has that behavior with probability 1.

We use the following two facts in Section 4. The first is an obvious consequence of the definition of fairness.

▶ **Observation 2.1.** *If $\tau = (q_i \mid 0 \leq i < \ell)$ is a fair trajectory of a population protocol $\mathbf{N}$, then, for every reaction $\rho$ of $\mathbf{N}$,*

$$(\exists^\infty i)[\rho \textit{ is enabled in } q_i] \implies (\exists^\infty j)[\rho \textit{ occurs at } j \textit{ in } \tau]. \tag{2.5}$$

A famous theorem of Harel [22, 26] implies that the general problem of deciding whether a chemical reaction network terminates on all fair trajectories is undecidable. Nevertheless, the following lemma gives a useful sufficient condition for termination of a population protocol on all fair trajectories. This lemma undoubtedly follows from a very old result on fairness, but we do not know a proper reference at the time of this writing. A proof appears in the Appendix.

▶ **Lemma 2.2** (fair termination lemma)**.** *If a population protocol with a specified initial state has a terminal trajectory from every reachable state, then all its fair trajectories are terminal.*

## 3 Single Phase Transition

This section presents the chemical reaction network $\mathbf{N}_1$ and proves that it exhibits a population-induced phase transition as described in the introduction.

Fix $m, n, p \in \mathbb{Z}^+$ with $n > m + 1$. Let $\mathbf{N}_1$ be a chemical reaction network consisting of the $n + 1$ *ζ-reactions*

$$\zeta_i \equiv Z_i + Z_i \to \begin{cases} Z_{i+1} + B & (0 \le i < m) \\ Z_{i+1} + R & (m \le i < n) \\ Z_i + R & (i = n) \end{cases}$$

and the *χ-reaction*

$$\chi \equiv B + R \to R + R.$$

All results here hold regardless of the rate constants of these $n + 2$ reactions.

We initialize $\mathbf{N}_1$ with $z_0 = p$ and all other counts 0.

Intuitively, $B$ is *blue*, $R$ is *red*, and the species $Z_i$ are all *colorless*.

▶ **Lemma 3.1.** $\mathbf{N}_1$ *terminates on all possible trajectories.*

▶ Notation. For $1 \le k \le n + 1$, let

$$S_k = \sum_{i=0}^{k-1} 2^i z_i,$$

noting that this quantity depends on the state of $\mathbf{N}_1$.

▶ **Lemma 3.2.** *Let $0 \le j \le n$ and $1 \le k \le n + 1$.*

1. *If $j \ne k - 1$, then the reaction $\zeta_j$ preserves the value of $S_k$.*
2. *If $j = k - 1$, then the reaction $\zeta_j$ reduces the value of $S_k$.*

▶ **Corollary 3.3.** *For every $1 \le k \le n + 1$, the inequality $S_k \le p$ is an invariant of $\mathbf{N}_1$.*

▶ **Corollary 3.4.** *If $1 \le k \le n$ and $z_k > 0$ in some reachable state of $\mathbf{N}_1$, then $p \ge 2^k$.*

In the following, for $d \in \mathbb{Z}^+$, we use both the mod-$d$ *congruence* (equivalence relation)

$$a \equiv b \bmod d,$$

which asserts of integers $a, b \in \mathbb{Z}$ that $b - a$ is divisible by $d$, and the **mod**-$d$ *operation*

$$b \bmod d$$

whose value, for $b \in \mathbb{Z}$, is the unique $r \in \mathbb{Z}$ such that $0 \le r < d$ and $r \equiv b \bmod d$.

▶ **Corollary 3.5.** *The congruence*

$$S_n \equiv p \bmod 2^n \tag{3.1}$$

*is an invariant of $\mathbf{N}_1$.*

▶ **Corollary 3.6.** *For every $1 \le k \le n$, the condition*

$$\Theta_k \;\equiv\; [z_k = \cdots = z_n = 0 \implies S_k = p]$$

*is an invariant of $\mathbf{N}_1$.*

▶ **Corollary 3.7.** *Let $(q_0, \ldots, q_t)$ be a trajectory of $\mathbf{N}_1$, where $q_t$ is a terminal state, and let $1 \le k \le n$. If $p \ge 2^k$, then there exists $1 \le s \le t$ such that $z_k > 0$ in $q_s$.*

▶ Notation. For each $r \in \{0, \ldots, 2^n - 1\}$, let $\lambda(r)$ be the number of 1s in the $n$-bit binary representation of $r$ (leading 0s allowed), and let

$$\varepsilon = \begin{cases} \lambda(p) & \text{if } p < 2^n \\ 1 + \lambda(p \bmod 2^n) & \text{if } p \geq 2^n. \end{cases}$$

Note that $\varepsilon$ is an integer depending on $n$ and $p$, and that $\varepsilon$ is negligible in the sense that $\varepsilon = o(p)$ as $p \to \infty$.

The boolean value of a condition $\varphi$ is $[\![\varphi]\!] = \textbf{if } \varphi \textbf{ then } 1 \textbf{ else } 0$.

▶ **Theorem 3.8.** $\mathbf{N}_1$ *terminates on all trajectories in the state* $(z_0, \ldots, z_n, b, r)$ *specified as follows.*

**(i)** $z_{n-1} \cdots z_0$ *is the* $n$*-bit binary expansion of* $p \bmod 2^n$.
**(ii)** $z_n = [\![p \geq 2^n]\!]$.
**(iii)** $b = (p - \varepsilon) \cdot [\![p < 2^m]\!]$
**(iv)** $r = (p - \varepsilon) \cdot [\![p \geq 2^m]\!]$.

**Proof.** Lemma 3.1 tells us that $\mathbf{N}_1$ terminates on all trajectories. Let $q = (z_0, \ldots, z_n, b, r)$ be a terminal state of $\mathbf{N}_1$, and note the following.

**(a)** For all $0 \leq i \leq n$, $\zeta_i$ is not enabled in $q$, so $z_i \in \{0, 1\}$.
**(b)** $\chi$ is not enabled in $q$, so $b = 0$ or $r = 0$.
**(c)** By (a), $S_n \leq \sum_{i=0}^{n-1} 2^i = 2^n - 1$, so Corollary 3.5 tells us that $S_n = p \bmod 2^n$, i.e., that (i) holds.
**(d)** If $p < 2^n$, then Corollary 3.4 tells us that $z_n = 0$. If $p \geq 2^n$, then Corollary 3.7 tells us that $z_n \geq 1$ somewhere along every trajectory leading to $q$. Since $z_n$ can never become 0 after becoming positive, this implies that $z_n = 1$ in $q$. Hence (ii) holds.
**(e)** By (c) and (d) we have $\sum_{i=0}^{n} z_i = \varepsilon$.
**(f)** Since $b + r + \sum_{i=0}^{n} z_i$ (the total population $p$) is an invariant of $\mathbf{N}_1$, (b) and (e) tell us that one of $b$ and $r$ is $p - \varepsilon$ and the other is 0.
**(g)** If $p < 2^m$, then Corollary 3.4 tells us that $z_m = \cdots = z_n = 0$ holds throughout every trajectory leading to $q$. This implies that none of the reactions $\zeta_m, \ldots, \zeta_n$ occurs along any trajectory leading to $q$, whence $r = 0$.
**(h)** If $p \geq 2^m$, then Corollary 3.7 tells us that $z_m > 0$ holds somewhere along every trajectory leading to $q$. This implies that the reaction $\zeta_{m-1}$ occurs, whence $r$ becomes positive, somewhere along every trajectory leading to $q$. Since $r$ can never become 0 after becoming positive, this implies that $r > 0$.
**(i)** By (f), (g), and (h), (iii) and (iv) hold. ◀

Since $\varepsilon$ is negligible with respect to $p$, Theorem 3.8 says that $\mathbf{N}_1$ terminates in an overwhelmingly blue state if $p < 2^m$ and in an overwhelmingly red state if $p \geq 2^m$. This is a very sharp phase transition at the population threshold $2^m$.

## 4 Coupled Phase Transitions

Let $m, n, p$, and $\mathbf{N_1}$ be as in Section 3, and let $\mathbf{N_2}$ be a CRN consisting of the $n+2$ reactions of $\mathbf{N_1}$ and the *ω-reaction*

$$\omega \equiv R + Z_n \to B + Z_n.$$

This section proves that $\mathbf{N_2}$ exhibits two coupled population-induced phase transitions as described in the introduction.

We use the same initialization for $\mathbf{N_2}$ as for $\mathbf{N_1}$. Again, all our results hold regardless of the rate constants of the $n+3$ reactions of $\mathbf{N_2}$.

Routine inspection verifies the following.

▶ **Observation 4.1.** *Lemma 3.2 and Corollaries 3.3-3.7 hold for* $\mathbf{N_2}$ *as well as for* $\mathbf{N_1}$.

If $p < 2^n$, then Corollary 3.4 tells us that $z_n$ never becomes positive in $\mathbf{N_2}$, so the $\omega$-reaction never occurs in $\mathbf{N_2}$. Thus, for $p < 2^n$, $\mathbf{N_2}$ behaves exactly like $\mathbf{N_1}$.

On the other hand, if $p \geq 2^n$, then the behavior of $\mathbf{N_2}$ is very different from that of $\mathbf{N_1}$. For example, in contrast with Lemma 3.1, we have the following.

▶ **Lemma 4.2.** *If* $p \geq 2^n$, *then not all trajectories of* $\mathbf{N_2}$ *terminate.*

It is easy to see that the infinite trajectory of $\mathbf{N}_2$ exhibited in the proof of Lemma 4.2 is not fair. In fact, we prove below that all fair paths of $\mathbf{N}_2$ terminate. First, however, we note that $\mathbf{N}_2$, like $\mathbf{N}_1$, has a unique terminal state.

Let $\varepsilon$ be as defined before Theorem 3.8.

▶ **Lemma 4.3.** *If* $p \geq 2^n$ *and* $\boldsymbol{N}_2$ *terminates, then it does so in the state* $(z_0, \ldots, z_n, b, r)$ *specified as follows.*

**(i)** $z_{n-1} \cdots z_0$ *is the* $n-$*bit binary expansion of* $p$ ***mod*** $2^n$.
**(ii)** $z_n = 1$.
**(iii)** $b = p - \varepsilon$.
**(iv)** $r = 0$.

▶ **Lemma 4.4.** *On any fair trajectory of* $\boldsymbol{N}_2$, *after finitely many steps, all* $\zeta$*-reactions are permanently disabled.*

▶ **Lemma 4.5.** *With any initialization, all fair trajectories of the chemical reaction network* $\boldsymbol{N}\chi\omega$, *consisting of just the reactions* $\chi$ *and* $\omega$, *are terminal.*

Recall the notation defined just before Theorem 3.8. The following result is our main theorem.

▶ **Theorem 4.6.** *Let* $(z_0, \ldots, z_n, b, r)$ *be the state of* $\boldsymbol{N}_2$ *specified as follows.*

**(i)** $z_{n-1} \cdots z_0$ *is the* $n$*-bit binary expansion of* $p$ ***mod*** $2^n$.
**(ii)** $z_n = [\![p \geq 2^n]\!]$.
**(iii)** $b = (p - \varepsilon) \cdot [\![p < 2^m \text{ or } p \geq 2^n]\!]$.
**(iv)** $r = (p - \varepsilon) \cdot [\![2^m \leq p < 2^n]\!]$.

*If* $p < 2^n$, *then* $\boldsymbol{N}_2$ *terminates in this state on all trajectories. If* $p \geq 2^n$, *then* $\boldsymbol{N}_2$ *terminates in this state on all fair trajectories.*

**Proof.** If $p < 2^n$, then Corollary 3.3 tells us that $z_n$ never becomes positive in $\mathbf{N}_2$, so $\omega$ is never enabled. Hence, in this case $\mathbf{N}_2$ behaves exactly like $\mathbf{N}_1$. Theorem 3.8 tells us that $\mathbf{N}_2$ terminates on all trajectories to the state satisfying (i) and (ii) above and, since $[\![p < 2^m]\!] = [\![p < 2^m \text{ or } p \geq 2^n]\!]$ and $[\![p \geq 2^m]\!] = [\![2^m \leq p < 2^n]\!]$, also satisfying (iii) and (iv) above.

If $p \geq 2^n$, then Lemmas 4.4 and 4.5 together tell us that $\mathbf{N}_2$ terminates on all fair trajectories. Since $[\![p \geq 2^n]\!] = 1$, $[\![p < 2^m \text{ or } p \geq 2^n]\!] = 1$, and $[\![2^m \leq p < 2^n]\!] = 0$, Lemma 4.3 tells us that termination must occur in the state satisfying (i)-(iv) above. ◀

Since $\varepsilon$ is again negligible with respect to $p$, Theorem 4.6 says that $\mathbf{N}_2$ terminates in an overwhelmingly blue state if $p < 2^m$ or $p \geq 2^n$ but in an overwhelmingly red state if $2^m \leq p < 2^n$. Hence $\mathbf{N}_2$ exhibits very sharp phase transitions at the population thresholds $2^m$ and $2^n$. As noted in the introduction and elaborated in Section 5 below, this has significant implications for the verification of chemical reaction networks.

# 5 Implications for Verification

The coupled phase transitions in the chemical reaction network $\mathbf{N_2}$ make it difficult to verify its behavior. In this section we describe the use and limitations of verifying the chemical reaction network using simulation, model checking and differential equations. None of these methods detected that the system turned red when the population is between $2^m = 2^{34} \approx 1.7 \times 10^{10}$ and $2^n = 2^{67} \approx 1.5 \times 10^{20}$. We then describe how the use of an interactive theorem prover enabled us to verify the chemical reaction network's behavior at both phase transitions, i.e., that it turned from blue to red at $2^m$ and from red to blue at $2^n$. The fact that theorem proving could verify behavior that was otherwise not verified for the chemical reaction network suggests that interactive theorem proving may have a useful role to play in future verification of a class of chemical reaction networks.

Recall that the chemical reaction networks $\mathbf{N_1}$ and $\mathbf{N_2}$ have fixed populations throughout any given execution, and that their initial states have $z_0$ as the entire population.

## 5.1 Simulation

The MATLAB SimBiology package is widely used to explore the behavior of a number of devices (molecules) executing concurrently [35]. Using SimBiology, simulations of the $\mathbf{N_2}$ chemical reaction network were performed on an Intel processor computer with a processor clock of 5.0 GHz and 64GB of RAM. Several simulations were performed with increasing populations $z_0$. With a population of $10^7$, the simulation performed as expected. However, with a population of $10^8$, the simulation failed and terminated with no output or error message. Thus, the stochastic simulation was unable to detect that the behavior of the $\mathbf{N_2}$ chemical reaction network could experience a phase transition.

## 5.2 Model Checking

The chemical reaction network $\mathbf{N_2}$ simulated in SimBiology and described above also was verified using the PRISM 4.6 probabilistic model checker [28]. Kwiatkowska and Thachuk, among others, have described the use of PRISM for the probabilistic verification of chemical reaction networks for biological systems [29].

To verify the chemical reaction network behavior we first converted the $\mathbf{N_2}$ model to SBML using the export function in SimBiology, and then converted the SBML model to PRISM using the sbml2prism conversion tool supplied with the PRISM software. PRISM was used to verify six key properties of the $\mathbf{N_2}$ chemical reaction network at multiple populations. For example, one of the properties stated that "$P >= 1[\mathrm{F\ G}\ r = 0]$", i.e., that with probability 1, the eventual state of the $R$ species has 0 molecules, and never changes from that. With a population of 100, PRISM generated the CTMC state model in 1.65 seconds using the same processor and memory as for the SimBiology simulations, and the verification of the six properties required less than 2 seconds of CPU time. For a population of 100 molecules, 97 are blue and 3 are colorless in the final state. PRISM also verified that in the final state $z_0 = z_1 = z_3 = z_4 = 0$ and $z_2 = z_5 = z_6 = 1$, so that $z_6z_5z_4z_3z_2z_1z_0$ is the binary expansion of one hundred.

However, we were unable to model check $\mathbf{N_2}$ with a population of 400 due to the rapid increase in states and limited memory. Thus, model checking confirmed the expected behavior of the $\mathbf{N_2}$ chemical reaction network for a population of 100 but could not detect the behavioral change to red when the population is greater than $2^{34}$.

Advanced methods to prune a model so that meaningful model checking can occur include symmetry reduction [23], statistical model checking [11], and automated partial exploration of the model [42]. Recent work by Cauchi, et al. using formal synthesis allowed verification of systems with 10 continuous variables [12]. However, even these methods would not be likely to help with the exceedingly large number of states when the number of molecules is scaled to a realistic value for experiments.

## 5.3 Differential Equations

We have seen how model checking and simulation fail to detect the "red" behavior in our chemical reaction network $\mathbf{N}_2$ due to the processing time and memory required for a large population. The red behavior also is not detected when $\mathbf{N}_2$ is approximated by deterministic semantics. In this model, a chemical reaction network is represented by a system of polynomial autonomous differential equations. Our purpose here is to investigate the usefulness of the large population heuristic in this context; we do not make any claims that our results respect the preconditions and caveats of Kurtz's theorem [27], which provides a mathematical link between deterministic and high-population stochastic systems.

In general, the system of differential equations induced by a chemical reaction network is difficult or impossible to solve exactly, and numerical methods are often used to approximate solutions. Here, we utilized MATLAB and the SimBiology package [35] to numerically integrate the system of differential equations for $\mathbf{N}_2$. We found that $\mathbf{N}_2$ reached and remained in a predominantly blue state for the duration of the simulation, again missing the red behavior.

We identify three potential causes for this failure. One potential cause is numerical failure; it may be that MATLAB's numerical integration was not robust enough to capture the relevant deterministic behavior, or that we did not let the simulation run long enough to converge. (We note that, at least in the stochastic case, we expect $\mathbf{N}_2$ to take an extreme amount of time to converge.) Another potential cause is that, as suggested by Kurtz's theorem, the deterministic system might correctly approximate high-population stochastic behavior, which falls above the second phase-transition threshold (and well above the range of a realistic wet-lab implementation of $\mathbf{N}_2$.) Finally, it may be that the stochastic and deterministic behaviors of $\mathbf{N}_2$ are not actually closely related, and the deterministic result does not imply anything conclusive about the underlying stochastic system. Regardless of the cause, however, we see that differential equation methods are not sufficient to capture the red behavior of $\mathbf{N}_2$.

## 5.4 Theorem Proving

The simulation, model checking, and differential equations approaches to chemical reaction network verification outlined above all make some simplifying assumptions: reduced state space or generalization to the continuum. In the case of our chemical reaction network, these assumptions lead to an incorrect verification result.

Interactive theorem proving, however, offers an exact approach that is guaranteed to apply at every scale. In the interactive theorem proving paradigm, users create a machine-checkable mathematical proof of verification properties in collaboration with a software system. Model checking also constructs a mathematical proof of correctness, but it relies more on a complete or semi-complete search of the state space in question. By contrast, the goal of interactive theorem proving is to construct a more traditional mathematical proof that is also machine-checkable. The result then applies to any population scale; a mathematical proof parameterized by population $N$ is valid at every possible value of $N$.

In a typical interactive theorem proving session, a user starts with a base of trusted facts generated from axioms and assumptions, and uses well-understood rules like modus ponens and double negation removal to construct new trusted facts and lemmas. As with a conventional mathematical proof, the user's goal is to add new trusted facts in a strategic way until reaching the goal of the proof.

We have verified our chemical reaction network with Isabelle/HOL [39, 40], a popular interactive theorem prover with several useful proof automation features. Instead of working at the level of rules like modus ponens, users can instruct Isabelle to execute more general proof methods that can apply sequences of basic rules without user direct input. For example, Isabelle can often prove the equivalence of predicate logic formulas with only one user-generated method invocation. Once invoked, such a method attempts to automatically construct a series of low-level logical rules whose application proves the equivalence. An Isabelle proof, then, consists of a directed acyclic graph of facts, connected by applications of these methods. The user's task is to choose a chain of intermediate goal facts in a way that allows Isabelle to connect them easily on the way to the overall goal.

Isabelle also provides the powerful Sledgehammer automation tool, which makes calls to external proof systems to automate aspects of proof creation. Sledgehammer takes a goal fact as input and attempts to generate a method invocation that proves it, operating at one level of abstraction above the proof methods invocations discussed above. Since it is often unclear which method to invoke (or which arguments to supply to it), this functionality can increase proof construction speed substantially.

We have used Isabelle to verify that our chemical reaction network has the desired behavior for all possible initializations. That is, if we initialize it with $N < 2^{34}$ or $N \geq 2^{67}$, the chemical reaction network terminates with majority blue, but if we initialize it with $2^{34} \leq N < 2^{67}$, it terminates with majority red. Theorem proving is able to verify behavior correctly in all regions, including the middle region that is inaccessible to model checking, simulation, and ODE methods. Figure 2 shows an image taken from the end of our Isabelle proof; it contains the three goal facts that we successfully verified, which summarize the behavior of the chemical reaction network.

Our Isabelle proof is loosely based on the proofs presented in Sections 3 and 4. Whereas those proofs define two chemical reaction networks $\mathbf{N_1}$ and $\mathbf{N_2}$, we use Isabelle's *locale* feature to associate assumptions about the population of $N$ with various parts of our proof. In the locale where $N < 2^{35}$, for example, we are able to prove that our chemical reaction network terminates with majority blue. Figure 2 shows how we enter these locales at the end of the proof to bring together our final results.

We refer to the three final locales as the lower blue region, the middle red region, and the upper blue region. For each region, our proof must show both termination and correctness; i.e., we must show that our chemical reaction network reaches a final state where no reactions are possible, and that any possible final state has the specified red or blue population.

As in Lemma 3.1, we show termination in the lower two regions via a "countdown" expression that is guaranteed to decrease with every reaction. See Figure 3 for our Isabelle definitions of termination and a general lemma we proved that allows us to use the countdown technique. In the upper blue region, it is impossible to prove termination without assuming that executions are fair. Our Isabelle proof includes Equation 2.4 as an unproven assumption; we are not interested in unfair trajectories, but since they exist we cannot prove that all trajectories are fair. For convenience, we also include Observation 2.1 as an an assumption. These two fairness assumptions allow us to prove that our chemical reaction network terminates in the upper blue region as well.

```
theory results
  imports
    zeta_termination
    blue_zeta_correctness
    red_zeta_correctness
    omega_termination
    omega_correctness
begin

context blue_zeta begin

lemma blue_zeta_result: "∃t. terminal (p t) ∧ b (p t) + 68 ≥ N"
proof -
  show ?thesis using blue_zeta_terminal_correct zeta_term path_term_def by auto
qed

end

context red_zeta begin

lemma red_zeta_result: "∃t. terminal (p t) ∧ r (p t) + 68 ≥ N"
proof -
  show ?thesis using red_zeta_terminal_correct zeta_term path_term_def by auto
qed

end

context final_omega begin

lemma omega_result: "∃t. terminal (p t) ∧ b (p t) + 68 ≥ N"
proof -
  show ?thesis using omega_term_state omega_terminal_correct by auto
qed

end
end
```

**Figure 2** The end of the Isabelle proof, which summarizes its results in three lemmas. The `context` statements bring our assumptions about the value of $N$ into context. The `using` statements bring in trusted facts from the rest of our proof and supply them as arguments to Isabelle's `auto` proof method. The identifier `p` refers to an arbitrary trajectory that is part of each context. Isabelle displays all statements with a white or light gray background to indicate that it has checked them completely, and they are valid.

Our correctness proofs rely heavily on the sum $S_{68} = \sum_{i=0}^{67} 2^i z_i$, using the notation of Section 3, which is an invariant in the lower two regions. In the upper blue region, it is an invariant until at least one $Z_{67}$ is produced. This invariant allows us to reason about the composition of terminal states. In the lower blue region, for example, we know that no red can ever be produced; the chemical reaction network can only produce its first red molecule alongside $Z$ species that would make the invariant too large. Following the proof of Theorem 3.8, then, we prove that any terminal state must be majority blue.

## 6 Conclusion

Taken together, the near-ubiquity of phase transitions in nature [47, 9], the sheer size of molecular populations, and the simplicity of the chemical reaction networks that we have shown to exhibit population-induced phase transitions, indicate that molecular programming will present us with many exceptions to the otherwise useful notion that most bugs can be demonstrated with small counterexamples. As we have seen, this presents a significant challenge to the verification of chemical reaction networks. Here we suggest some directions of current and future research that might help meet this challenge.

```
theory termin
  imports piptcrn
begin

definition terminal :: state ⇒ bool where terminal s1 = (¬(∃ s2. K s1 s2))
definition nonterm :: state ⇒ bool where nonterm s = (¬(terminal s))

definition path-term :: (nat ⇒ state) ⇒ bool where
  (path-term p) = (∃ t. (terminal (p t)))

definition state-term :: (state ⇒ bool) where
  (state-term s) = (∀ (p :: (nat ⇒ state)).
     ((∃ t. ((p t) = s))
      ⟶ (path-term p)))

lemma dec-imp-term:
    fixes f :: state ⇒ nat
    fixes p :: nat ⇒ state
    fixes c :: nat
    assumes evterm: ((f s) ≤ c) ⟶ (terminal s)
    assumes dec: ∀ i. ((¬(terminal (p i))) ⟶ (
      (f (p (i + 1)) < (f (p i)))))
    shows path-term p
proof −
  {
    fix n::nat
    have ((∃ t. ((f (p t)) ≤ n)) ⟶ (path-term p))
    proof (induction n)
      case 0
      then show ?case
        using dec gr-implies-not0 path-term-def by blast
    next
      case (Suc n)
      then show ?case
        by (metis dec le-SucE less-Suc-eq-le path-term-def)
    qed
  }
  then show ?thesis by blast
qed

end
```

**Figure 3** This Isabelle code defines a terminal state as a state with no outgoing reactions; $K$ is a relation that encodes which state transitions our reaction set allows. We also show a sample lemma that helps prove termination: if we identify a countdown expression $f$ and a constant $C$ such that all states with $f < C$ are terminal, then our system is guaranteed to terminate.

A great deal of creative work has produced a steady scaling up of model checking to larger and larger state spaces [16, 15, 1, 8, 34, 13]. Perhaps the most hopeful approach for dealing with population-induced phase changes, or with more general population-sensitive behaviors, is the model checking of parametrized systems [1].

Our results clearly demonstrate the advantage of including theorem proving (by humans and by software) in the verification toolbox for chemical reaction networks and other molecular programming languages. This in turn suggests that software proof assistants such as Isabelle [40, 39] be augmented with features to deal more directly with chemical reaction networks and with population-sensitive phenomena. It would also be useful to know how much of such work could be carried out with more fully automated theorem provers such as Vampire [25].

Some future programmed molecular applications will be safety-critical, such as in health diagnostics and therapeutics. It is likely that evidence that such systems behave as intended will be required for certification by regulators prior to deployment. Toward providing such evidence, Nemouchi et al. have recently shown how a descriptive language for safety cases can be incorporated into Isabelle in order to formalize argument-based safety assurance cases [38].

We conclude with a more focused, theoretical question. Our chemical reaction network $\mathbf{N}_1$ exhibits its phase transition on all trajectories, while $\mathbf{N}_2$ exhibits its coupled phase transitions only on all fair trajectories. Is there a chemical reaction network that achieves $\mathbf{N}_2$'s coupled phase transitions on *all* trajectories?

## References


1 Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 685–725. Springer, 2018. doi:10.1007/978-3-319-10575-8_21.

2 David F. Anderson and Thomas G. Kurtz. Continuous time Markov chain models for chemical reaction networks. In Heinz Koeppl, Gianluca Setti, Mario di Bernardo, and Douglas Densmore, editors, *Design and Analysis of Biomolecular Circuits*, pages 3–42. Springer, 2011.

3 David F. Anderson and Thomas G. Kurtz. *Stochastic Analysis of Biochemical Systems*. Springer, 2015.

4 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.

5 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

6 Stefan Badelt, Seung Woo Shin, Robert F. Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. A general-purpose CRN-to-DSD compiler with formal verification, optimization, and simulation capabilities. In *Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming*, Springer, pages 232–248, 2017.

7 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

8 Luca Bortolussi, Luca Cardelli, Marta Kwiatkowska, and Luca Laurenti. Central limit model checking. *ACM Trans. Comput. Log.*, 20(4):19:1–19:35, 2019. doi:10.1145/3331452.

9 Sarah Cannon, Sarah Miracle, and Dana Randall. Phase transitions in random dyadic tilings and rectangular dissections. *SIAM J. Discret. Math.*, 32(3):1966–1992, 2018. doi:10.1137/17M1157118.

10 Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific Reports*, 2, 2012.

11 Luca Cardelli, Marta Kwiatkowska, and Max Whitby. Chemical reaction network designs for asynchronous logic circuits. *Natural Computing*, 17(1):109–130, 2018. doi:10.1007/s11047-017-9665-7.

12 Nathalie Cauchi, Luca Laurenti, Morteza Lahijanian, Alessandro Abate, Marta Kwiatkowska, and Luca Cardelli. Efficiency through uncertainty: scalable formal synthesis for stochastic hybrid systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019.*, pages 240–251, 2019. doi:10.1145/3302504.3311805.

13 Milan Ceska, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Shepherding hordes of Markov chains. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 172–190. Springer, 2019.

14 Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature Nanotechnology*, 8(10):755–762, 2013.

15 Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Asp. Comput.*, 30(1):45–75, 2018. doi:10.1007/s00165-017-0432-4.

16 Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009. doi:10.1145/1592761.1592781.

17 Anne Condon, Monir Hajiaghayi, David G. Kirkpatrick, and Ján Manuch. Simplifying analyses of chemical reaction networks for approximate majority. In *Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming*, pages 188–209. Springer, 2017.

18 Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In Anne Condon, David Harel, Joost N. Kok, Arto Salomaa, and Erik Winfree, editors, *Algorithmic Bioprocesses*, Natural Computing Series, pages 543–584. Springer, 2009.

19 Shawn M. Douglas, Ido Bachelet, and George M. Church. A logic-gated nanorobot for targeted transport of molecular payloads. *Science*, 335(6070):831–834, 2012.

20 Samuel J. Ellis, Titus H. Klinge, James I. Lathrop, Jack H. Lutz, Robyn R. Lutz, Andrew S. Miner, and Hugh D. Potter. Runtime fault detection in programmed molecular systems. *ACM Trans. Softw. Eng. Methodol.*, 28(2):6:1–6:20, 2019. doi:10.1145/3295740.

21 François Fages, Guillaume Le Guludec, Olivier Bournez, and Amaury Pouly. Strong Turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In *Proceedings of the 15th International Conference on Computational Methods in Systems Biology*, pages 108–127. Springer, 2017.

22 David Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness. *J. ACM*, 33(1):224–248, 1986. doi:10.1145/4904.4993.

23 J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. In *Computational Methods in Systems Biology*, pages 32–47, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

24 Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019. doi:10.1145/3338843.

25 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35. Springer, 2013. doi:10.1007/978-3-642-39799-8_1.

26 Dexter Kozen. *Theory of Computation*. Texts in Computer Science. Springer, 2006. doi:10.1007/1-84628-477-5.

27 Thomas G. Kurtz. The relationship between stochastic and deterministic models for chemical reactions. *The Journal of Chemical Physics*, 57(7):2976–2978, 1972.

28 Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

29 Marta Kwiatkowska and Chris Thachuk. Probabilistic model checking for biology. *Software Systems Safety*, 36:165–189, 2014.

30 Marta Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989. doi:10.1016/0950-5849(89)90159-6.

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

32 Suping Li, Qiao Jiang, Shaoli Liu, Yinlong Zhang, Yanhua Tian, Chen Song, Jing Wang, Yiguo Zou, Gregory J Anderson, Jing-Yan Han, Yung Chang, Yan Liu, Chen Zhang, Liang Chen, Guangbiao Zhou, Guangjun Nie, Hao Yan, Baoquan Ding, and Yuliang Zhao. A DNA nanorobot functions as a cancer therapeutic in response to a molecular trigger in vivo. *Nature Biotechnology*, 36:258, 2018.

33 Xiaowei Liu, Yan Liu, and Hao Yan. Functionalized DNA nanostructures for nanomedicine. *Israel Journal of Chemistry*, 53(8):555–566, 2013.

34 Alessio Lomuscio and Edoardo Pirovano. A counter abstraction technique for the verification of probabilistic swarm systems. In *Proceedings of the 18th International Conference on*

*Autonomous Agents and MultiAgent Systems, AAMAS'19*, pages 161–169, 2019. URL: http://dl.acm.org/citation.cfm?id=3331689.

35 MATLAB. *version 9.7.0 (R2019b, Update 4)*. The MathWorks Inc., Natick, Massachusetts, 2019.

36 Melissa B. Miller and Bonnie L. Bassler. Quorum sensing in bacteria. *Annual Review of Microbiology*, 55(1):165–199, 2001. PMID: 11544353. doi:10.1146/annurev.micro.55.1.165.

37 Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

38 Yakoub Nemouchi, Simon Foster, Mario Gleirscher, and Tim Kelly. Isabelle/SACM: Computer-assisted assurance cases with integrated formal methods. In *Proceedings of the 15th International Conference on Integrated Formal MethodsIFM 2019*, pages 379–398. Springer, 2019. doi:10.1007/978-3-030-34968-4_21.

39 Tobias Nipkow and Gerwin Klein. *Concrete Semantics–With Isabelle/HOL*. Springer, 2014.

40 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 1 edition, 2002.

41 Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *Formal Asp. Comput.*, 31(6):675–698, 2019. doi:10.1007/s00165-019-00492-1.

42 Esteban Pavese, Víctor Braberman, and Sebastián Uchitel. Less is more: Estimating probabilistic rewards over partial system explorations. *ACM Transactions on Software Engineering and Methodology*, 25(2):16:1–16:47, 2016.

43 Gerald Pollack and Wei-Chun Chin, editors. *Phase Transitions in Cell Biology*. Springer, 2008.

44 Hamid Ramezani and Hendrik Dietz. Building machines with DNA molecules. *Nature Reviews Genetics*, 21(1):5–26, 2020.

45 Dana Randall. Phase transitions in sampling algorithms and the underlying random structures. In Haim Kaplan, editor, *Proceedings Scandinavian Symposium and Workshops on Algorithm Theory SWAT*, page 309. Springer, 2010. doi:10.1007/978-3-642-13731-0_29.

46 Dana Randall. Phase Transitions and Emergent Phenomena in Random Structures and Algorithms (Keynote Talk). In *31st International Symposium on Distributed Computing (DISC 2017)*, pages 3:1–3:2. Schloss Dagstuhl LZI, 2017. doi:10.4230/LIPIcs.DISC.2017.3.

47 Dana Randall. Statistical Physics and Algorithms (Invited Talk). In Christophe Paul and Markus Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020)*, pages 1:1–1:6. Schloss Dagstuhl LZI, 2020.

48 H. G. Rice. *Classes of Recursively Enumerable Sets and Their Decision Problems*. PhD thesis, Syracuse University, 1951.

49 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. doi:10.1090/s0002-9947-1953-0053041-6.

50 Apoorva Sarode, Akshaya Annapragada, Junling Guo, and Samir Mitragotri. Layered self-assemblies for controlled drug delivery: A translational overview. *Biomaterials*, 242:119929, 2020. doi:10.1016/j.biomaterials.2020.119929.

51 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, 2008.

52 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. In *Proceedings of the 14th International Meeting on DNA Computing*, pages 57–69. Springer, 2009.

53 Anupama J. Thubagere, Chris Thachuk, Joseph Berleant, Robert F. Johnson, Diana A. Ardelean, Kevin M. Cherry, and Lulu Qian. Compiler-aided systematic construction of large-scale DNA strand displacement circuits using unpurified components. *Nature Communications*, 8, 2017 .

54 John C. Wooley and Herbert S. Lin. *Catalyzing Inquiry at the Interface of Computing and Biology*. National Academies Press, 2005.

55 David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature Chemistry*, 3(2):103–113, 2011.

# A Proof of Fair Termination Lemma

▶ **Lemma A.1** (fair termination lemma). *If a population protocol with a specified initial state has a terminal trajectory from every reachable state, then all its fair trajectories are terminal.*

**Proof.** Let $\mathbf{N}$ be a population protocol with initial state $q_0$, and assume that $\mathbf{N}$ has a terminal trajectory from every reachable state. Let $\tau = (q_i \mid 0 \leq i < \infty)$ be an infinite trajectory of $\mathbf{N}$. It suffices to show that $\tau$ is not fair.

For each state $q$ of $\mathbf{N}$, let

$$I_q = \{i \in \mathbb{N} \mid q_i = q\}. \tag{A.1}$$

Since $\mathbf{N}$ is a population protocol, it has finitely many reachable states, so there is a state $q^*$ of $\mathbf{N}$ such that the set $I_{q^*}$ is infinite. This state $q^*$ is reachable, so our assumption tells us that there is a finite trajectory $\tau^* = (q^*_i \mid 0 \leq i < \ell)$ of $\mathbf{N}$ such that $q^*_0 = q^*$ and $q^*_{\ell-1}$ is terminal.

Now $I_{q^*_0} = I_{q^*}$ is infinite and $I_{q^*_{\ell-1}} = \emptyset$ (because $q^*_{\ell-1}$ is terminal, so it does not appear in the infinite trajectory $\tau$), so there exists $0 \leq k < \ell - 1$ such that $I_{q^*_k}$ is infinite and $I_{q^*_{k+1}}$ is finite. Let $q^{**} = q^*_k$, and let $\rho$ be the reaction that takes $q^*_k$ to $q^*_{k+1}$. Then $\rho$ is enabled in $q^{**}$ and there exist infinitely many $i$ such that $q_i = q^{**}$ (because $I_{q^{**}}$ is infinite), but there are only finitely many $j$ for which $q_j = q^*$ and $\rho$ occurs at $j$ in $\tau$ (because $I_{q^*_{k+1}}$ is finite). Hence $\tau$ is not fair. ◀

# ALCH: An Imperative Language for Chemical Reaction Network-Controlled Tile Assembly

**Titus H. Klinge**
Department of Mathematics and Computer Science, Drake University, Des Moines, IA, USA
titus.klinge@drake.edu

**James I. Lathrop**
Department of Computer Science, Iowa State University, Ames, IA, USA
jil@iastate.edu

**Sonia Moreno**
Department of Computer Science, Carleton College, Northfield, MN, USA
morenos@carleton.edu

**Hugh D. Potter**
Department of Computer Science, Iowa State University, Ames, IA, USA
hdpotter@iastate.edu

**Narun K. Raman**
Department of Computer Science, Carleton College, Northfield, MN, USA
ramann@carleton.edu

**Matthew R. Riley**
Department of Computer Science, Iowa State University, Ames, IA, USA
mrriley@iastate.edu

## Abstract

In 2015 Schiefer and Winfree introduced the chemical reaction network-controlled tile assembly model (CRN-TAM), a variant of the abstract tile assembly model (aTAM), where tile reactions are mediated via non-local chemical signals. In this paper, we introduce ALCH, an imperative programming language for specifying CRN-TAM programs. ALCH contains common features like Boolean variables, conditionals, and loops. It also supports CRN-TAM-specific features such as adding and removing tiles. A unique feature of the language is the *branch* statement, a nondeterministic control structure that allows us to query the current state of tile assemblies. We also developed a compiler that translates ALCH to the CRN-TAM, and a simulator that simulates and visualizes the self-assembly of a CRN-TAM program. Using this language, we show that the discrete Sierpinski triangle can be strictly self-assembled in the CRN-TAM. This solves an open problem that the CRN-TAM is capable of self-assembling infinite shapes at scale one that the aTAM cannot. ALCH allows us to present this construction at a high level, abstracting species and reactions into C-like code that is simpler to understand. Our construction utilizes two new CRN-TAM techniques that allow us to tackle this open problem. First, it employs the branching feature of ALCH to *probe* the previously placed tiles of the assembly and detect the presence and absence of tiles. Second, it uses scaffolding tiles to precisely control tile placement by occluding any undesired binding sites.

**2012 ACM Subject Classification** Theory of computation → Models of computation

**Keywords and phrases** Tile assembly, Chemical reaction network, Sierpinski triangle

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.6

**Supplementary Material** The ALCH compiler and the CRN-TAM simulator, together with examples and visual illustrations, are available at `http://web.cs.iastate.edu/~lamp`.

**Funding** This research was supported in part by National Science Foundation grants 1900716 and 1545028.

**Acknowledgements** We thank the three anonymous reviewers for their helpful comments and suggestions. We especially thank reviewer 3 for their detailed insights, comments, and suggestions.

# 1 Introduction

Molecular programming is a relatively new field that weaves together biology and computer science to specify the behavior of molecules at the nanoscale. Early research in the field was sparked in 1982 by Seeman's pioneering work employing DNA crossover tiles to self-assemble crystals at the nanoscale [13]. Seeman's work was later extended by Erik Winfree to include *cooperative* DNA tile self-assembly to construct more complex shapes and patterns [15]. Winfree formalized the abstract tile assembly model (aTAM) in his Ph.D. thesis, where he proved it is Turing complete [15]. As a result, the aTAM is considered a programming language for self-assembling two and three-dimensional nanoscale patterns and is still actively investigated today [8, 3, 10, 7].

Another model commonly used to study biomolecular computation is the *chemical reaction network* (*CRN*), which models the interactions of chemical species. The CRN model assumes the solution is well-mixed, and therefore computations are amorphous and do not rely on geometry or structure. Two common variants of the CRN model are *stochastic CRNs* and *deterministic CRNs*. Stochastic CRNs are modeled with discrete species counts, and their reactions are probabilistic. In contrast, deterministic CRNs model the species' state continuously with real-valued concentrations governed by a system of autonomous ordinary differential equations (ODEs). The law of mass action determines the rates of reactions in both models. For more information on these models, see [6, 5, 2].

In 2015, Schiefer and Winfree introduced the *chemical reaction network-controlled tile assembly model* (*CRN-TAM*) [11, 12]. Their model combines the amorphous properties of stochastic CRNs with the spatial self-assembly of complex structures afforded by the aTAM. More specifically, a chemical reaction network interacts with tiles from the aTAM model to exert non-local control over the self-assembly process.

Molecular programming provides a rich field for algorithmic study. However, it is often time-consuming and complex to generate algorithmic constructions at the level of chemical species, tiles, or reactions. Recently, Vasić, Soloveichik, and Khurshid introduced CRN++, a high-level language for implementing deterministic CRN programs [14]. The CRN++ language provides a toolset for manipulating concentrations as numerical variables, with some support for conditionals and loops. This simplifies the development of high-level deterministic CRNs by abstracting away many low-level details. Other such languages exist such as Liekens and Fernando's Chemical Bare Bones (CBB), a hypothetical chemical implementation of the simple but Turing complete Bare Bones programming language [9]. CBB implements increment, decrement, and loop instructions using a catalytic particle model in which a single multistate particle catalyzes reactions based on its state. However, these languages cannot be used for CRN-TAM programs, since they have no provision for tile self-assembly.

On the tile self-assembly side, we have seen several forms of abstraction. Becker presents a geometry-based system for generating shapes in the aTAM [1]. This system allows users to describe how information and assembly construction propagate along vectors defined in the physical space of the assembly. Users can then generate an aTAM system by designing a system of vectors and applying a well-defined procedure to convert it into tiles. Doty and Patitz provide a toolset at a lower level of abstraction, focusing on the connections between individual tiles and how information is shared across them [4]. Users can define variables to be transmitted from tile to tile via bond labels and transformation functions to "modify" those variables within a tile while specifying which sets of tiles can bond with which. The provided software then automatically generates an aTAM system. Both of these

tools focus on the parallel, semi-uncoordinated concept of tile self-assembly typical of aTAM constructions. In the CRN-TAM, on the other hand, the CRN component allows precise control over which tiles are added and when.

CRN-TAM constructions often rely on sequences of reactions and tile attachments, with sequential execution enforced by associating a chemical species with each reaction in the chain. For this reason, the CRN-TAM is a natural fit for a high-level imperative programming language. In this paper, we present the Algorithmic Language for Chemistry (ALCH), an imperative language for specifying CRN-TAM programs. ALCH targets the specific CRN-TAM design paradigm described above, where the CRN component mediates a strictly controlled sequence of tile actions We do not intend ALCH in its current form to be used for highly parallel aTAM-style constructions.

ALCH is reminiscent of other popular imperative languages, supporting loops and conditionals but omitting numerical computation and function calls. ALCH also contains many CRN-TAM specific statements that abstract away low-level details of the model's underlying semantics while maintaining that statements are executed in sequence. ALCH also includes a *branch* statement, a control structure that allows CRN-TAM programs to nondeterministically choose between a finite number of self-assembly paths. We are not aware of any shape that can be constructed in the CRN-TAM but not in ALCH, but we do not claim that ALCH is as general as the CRN-TAM. We have implemented an ALCH compiler that translates ALCH code into a proper CRN-TAM program and a simulator that visualizes the assembly process of a CRN-TAM program[1].

Using ALCH, we demonstrate that the CRN-TAM can construct infinite shapes that the aTAM cannot. For example, the discrete Sierpinski triangle is a well-known self-similar fractal that can be *weakly* self-assembled in the aTAM [15] but cannot be *strictly* self-assembled [8]. Weak self-assembly allows for "filler" tiles to be used to propagate information through an assembly, whereas strict self-assembly disallows this. We show that the non-local communication provided by the CRN-TAM is sufficient to overcome this limitation. Using ALCH, we construct a CRN-TAM program that strictly self-assembles the discrete Sierpinski triangle. Our construction relies on the ability to add and remove scaffolding tiles and self-assembles the fractal in a natural way, using only localized information contained in the current assembly. We achieve this by using ALCH's nondeterministic branch feature to probe previously placed tiles to inform which tiles are placed next. We also use the scaffolding tiles to occlude any spurious bonding sites, giving precise control over the placement of the next tile. The construction proceeds in a sequence of stages where each stage successfully self-assembles a subset of the discrete Sierpinski triangle. After the completion of a stage, all scaffolding tiles are removed, leaving only the Sierpinski triangle tiles. Thus, in the limit, only the Sierpinski triangle remains, since the scaffolding tiles are removed infinitely often. In fact, the ratio of scaffold tiles to Sierpinski triangle tiles approaches zero as the self-assembly process proceeds. The ALCH programming language and simulator simplifies the development process and the specification of the CRN-TAM program.

The rest of the paper is organized as follows. Section 2 gives an overview of the CRN-TAM model. Section 3 presents a detailed description of the ALCH programming language, including how each statement is compiled to the CRN-TAM. Section 4 gives an overview of the construction for the discrete Sierpinski triangle using the ALCH language, with examples to illustrate key concepts such as probing using nondeterministic branching. Finally, Section 5 discusses some conclusions from this work.

[1] The ALCH compiler and the CRN-TAM simulator, together with examples and visual illustrations, are available at `http://web.cs.iastate.edu/~lamp`.

## 2 Preliminaries

We now review the chemical reaction network-controlled tile assembly model (CRN-TAM), which combines the notions of the abstract tile-assembly model (aTAM) [15] and the stochastic chemical reaction network (sCRN) [2]. For a complete introduction to the model, see Schiefer and Winfree's original paper [11].

A *tile type* is a tuple $\boxed{t} = (N, E, S, W)$ consisting of four *bonds* for the north, east, south, and west sides of the tile, respectively. Each bond is a tuple $B = (\ell_B, s_B)$ where $\ell_B$ is the *label* and $s_B$ is the *binding strength* which is a non-negative integer. Given a finite set of tile types $T$, an *assembly* is a partial function $\alpha : \mathbb{Z}^2 \dashrightarrow T$ that encodes the positions of tiles in two-dimensional space. If $\alpha(i, j)$ is undefined, then we say that $(i, j)$ is *unoccupied* in the assembly $\alpha$. When two adjacent tiles in $\alpha$ have matching bond labels $\ell_N$ on their abutting sides, we say that they *interact* with a strength determined by their bond strengths $s_B$.

The literature is unclear about whether it is permissible to have bonds with the same label but asymmetric bond strengths; we have made the choice to allow it in this work. We adopt the prescription that adjacent bonds with the same label have interaction strength $s$, where $s$ is given by the minimum of the bond strengths. Note that this prescription is physically plausible; if we view a bond site as an exposed single DNA strand, a stronger bond corresponds to a longer exposed area. We can then choose the base pairs exposed by a weaker bond to be a subset of those exposed by a stronger bond. Our probe mechanism, discussed in a subsequent section, relies on such asymmetric bonds.

The *binding graph* of an assembly $\alpha$ is a two-dimensional lattice of vertices representing the tiles of $\alpha$ where two vertices are connected by an undirected edge with weight $s$ if their corresponding tiles in $\alpha$ interact with strength $s$. For $\tau \in \mathbb{N}$, we say that an assembly is $\tau$-stable if the minimum cut of its binding graph is at least $\tau$. We also denote assemblies using $\boxed{\boxed{\alpha}}$, and given a tile type $\boxed{t}$, use $\boxed{\boxed{t}}$ to denote the singleton assembly that consists of only a single tile of type $t$ placed at the origin. Note that the number of tiles of a given tile type $\boxed{t}$ available in solution is finite but unbounded. This is in contrast to the aTAM which assumes an unlimited supply of all tile types throughout the self-assembly process.

A *signal species* is an abstract molecule type. In contrast to tiles, signal species have no geometry and are used to facilitate non-local communication in the self-assembly process. Every tile $\boxed{t}$ has a unique *removal species* $t^*$, and given a finite set $T$ of tile types, we write $T^* = \{t^* \mid \boxed{t} \in T\}$ to denote the set of all tile removal species of $T$. Note that the definitions in Schiefer and Winfree's papers [11, 12] allow tile removal species to be shared or even omitted. However, it is convenient for the compiler to always generate tile removal species and for them to be unique.

A *CRN-TAM program* is a tuple $\mathcal{P} = (S, T, R, \tau, I)$ where $T$ is a finite set of tile types, $S$ is a finite set of signal species that satisfies $T^* \subseteq S$, $\tau \in \mathbb{N}$ is the *temperature*, $I : S \cup T \to \mathbb{N}$ is the *initial state* which specifies how many tiles and signal molecules are initially present, and $R$ is a finite set of reactions that are of the following six types.

**Signal** reactions are of the form $X_1 + X_2 \to Y_1 + Y_2$ where $X_1, X_2, Y_1, Y_2 \in S \cup \{\epsilon\}$. The $\epsilon$ symbol denotes the absence of a species, therefore $X + \epsilon \to Y_1 + Y_2$ is equivalent to $X \to Y_1 + Y_2$. Since these reactions only consist of signal species, their semantics are identical to those in the traditional sCRN model. The species on the left-hand-side are called *reactants* and are consumed by the reaction and the species on the right-hand-side are called *products* and are produced by the reaction.

**Deletion** reactions are of the form $X + \boxed{t} \to Y_1 + Y_2$ where $X, Y_1, Y_2 \in S \cup \{\epsilon\}$ and $\boxed{t} \in T$. These reactions consume a tile, treating it as if it were a signal species. Note, deletion reaction cannot consume tiles bound to the assembly.

**Creation** reactions are of the form $X_1 + X_2 \to \boxed{t} + Y$ where $X_1, X_2, Y \in S \cup \{\epsilon\}$ and $\boxed{t} \in T$. These reactions produce tiles, making them available to interact with assemblies.

**Relabelling** reactions are of the form $X + \boxed{t_1} \to Y + \boxed{t_2}$ where $X, Y \in S \cup \{\epsilon\}$ and $\boxed{t_1}, \boxed{t_2} \in T$.

**Activation** reactions are of the form $X + \boxed{t} \to \boxed{\boxed{t}} + t^*$ where $X \in S$, $\boxed{t} \in T$, and $t^*$ is the signal removal species for $\boxed{t}$. These reactions use tile $\boxed{t}$ to seed a new assembly with $\boxed{t}$ placed at the origin.

**Deactivation** reactions are of the form $\boxed{\boxed{t}} + t^* \to \boxed{t} + Y$ where $\boxed{t} \in T$, $t^*$ is the removal signal for $\boxed{t}$, and $Y \in S \cup \{\epsilon\}$. These reactions remove the tile $\boxed{t}$ from the singleton assembly $\boxed{\boxed{t}}$, thereby deactivating it.

In addition to the reactions above, for each $\boxed{t} \in T$, the following two reactions included in the set of reactions $R$.

**Addition** reactions of the form $\boxed{\alpha} + \boxed{t} \to \boxed{\beta} + t^*$ where $\boxed{\beta}$ and $\boxed{\alpha}$ are $\tau-$stable assemblies that differ by one copy of $\boxed{t} \in T$ and $t^* \in T^*$ is the removal signal for $\boxed{t}$.

**Removal** reactions of the form $\boxed{\beta} + t^* \to \boxed{\alpha} + \boxed{t}$ where again $\boxed{\beta}$ and $\boxed{\alpha}$ are $\tau-$stable assemblies that differ by one copy of $\boxed{t} \in T$ and $t^* \in T^*$ is the removal signal for $\boxed{t}$. These reactions can only remove $\boxed{t}$ from $\boxed{\beta}$ if there is an instance of $\boxed{t}$ that is bound at exactly $\tau$ strength.

A CRN-TAM program $\mathcal{P}$ is initialized with nonnegative counts of each tile and signal species type, according to $I$. In an execution of $\mathcal{P}$, the reactions above occur in a stochastic sequence. The species or assemblies on the left-hand side of a reaction are the *reactants* and those on the right are the *products*. A reaction is *enabled* if all of its reactants are present in solution. The subsequent reaction to execute is always chosen randomly from the set of all enabled reactions. The likelihood of choosing a particular reaction is proportional to the product of its reactant counts, as with regular stochastic CRNs. If an execution reaches a state where no reactions are enabled, we say that it has *terminated*. Some CRN-TAM programs, like the DST construction in this work, do not terminate and continue indefinitely. For more information on the kinetics of the CRN-TAM model, see [12].

The CRN-TAM distinguishes between free tiles in solution and tiles that are part of activated assemblies. Free tiles can bond to assemblies, but two free tiles cannot bond together. All tiles come into being as free tiles, including those in the initialization; immediately after initialization, then, only signal, creation, deletion, and relabeling reactions are possible. We refer to these reactions as the *CRN component* of the CRN-TAM program. The CRN component usually serves to coordinate activation, deactivation, addition, and removal reactions and guide tile assembly growth.

In most CRN-TAM constructions, the CRN component is engineered to execute at least one activation reaction, which creates a new tile assembly so tiles can be added. Tiles created with creation reactions (or present in solution from the start) can then bond via their addition reactions, and potentially later be removed via their removal reactions. As discussed above, a tile can bond at any site on an activated assembly where it would interact with strength at least $\tau$; tiles are subject to removal reactions when their interaction strength does not exceed $\tau$. Note that if tile $\boxed{t}$ has a removal signal $t^*$, then adding $\boxed{t}$ releases $t^*$, and removing $\boxed{t}$ requires and consumes $t^*$. This allows the CRN component to interact

more precisely with the addition and removal reactions. Some constructions also employ the deactivation reaction to eliminate existing (singleton) assemblies; unlike in the aTAM, the number of concurrent assemblies can increase or decrease over time. The constructions in this work, however, do not require more than one assembly.

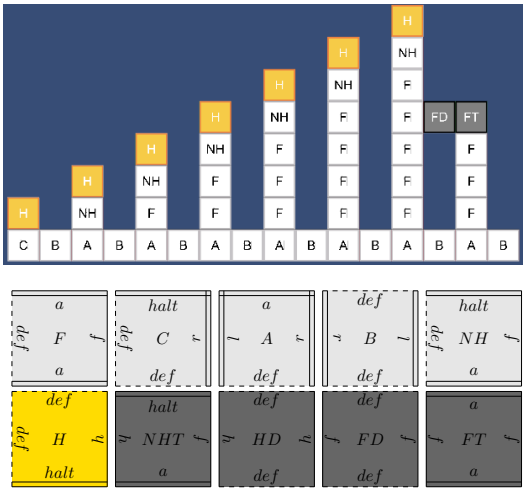
## 3 The ALCH Programming Language

We present an overview of the features of the ALCH language and its implementation. ALCH is an imperative language with provisions specific to the CRN-TAM model such as the **add**, **remove**, **activate**, and **deactivate** statements which all take a tile type as a parameter and execute the corresponding tile actions. ALCH provides high-level features such as conditions, loops, and variable declaration and assignment. To guarantee the proper sequential execution of the code, special *line number species* are used to track progress through the ALCH program. By ensuring that only a single line number species is present at any given time[2], the CRN-TAM program can transition from instruction to instruction without introducing any race conditions. At this time, ALCH only supports global variables and three datatypes: **bool**, **BondLabel**, and **TileSpecies**. Variables of type **bool** may be reassigned throughout the computation, but all **BondLabel** and **TileSpecies** variables are immutable and final. One unique feature of ALCH is the **branch** statement, which nondeterministically chooses and executes multiple independent code blocks of tile addition and removal statements until one block finishes execution. Effects from uncompleted blocks are reversed, so only the code from the completed block remains. The **branch** statement also returns a **bool** associated with the block that finished successfully. Using **branch**, it is possible to query the state of tile assemblies without permanently attaching tiles to them. Each block in a **branch** statement is implemented as a reversible random walk. As an optimization, blocks can be given different weights to make them more likely to be chosen at the nondeterministic branch point.

We developed a software compiler in C# that compiles ALCH programs into CRN-TAM programs. We also developed a simulator for the CRN-TAM that includes the following two extensions to the model which are used only for optimization purposes: (1) it supports reactions with arbitrary arity, relaxing the CRN-TAM requirement that reactions are at most bimolecular; (2) it allows any reaction to add, remove, or activate a tile as a side effect and removes the requirement for the specific per-tile add and remove actions. Note that the output of the ALCH compiler is strictly compliant with the original CRN-TAM as specified in [11]. We have not yet implemented tile deactivation in the simulator.

To demonstrate the expressiveness of ALCH, we will show that the CRN-TAM can strictly self-assemble an infinite shape at temperature 2 that the aTAM cannot. Consider an infinite staircase, visualized in Figure 1, where for each $k \in \mathbb{N}$, the $(2k)$th column is $2 + k$ tiles tall and the $(2k + 1)$th column is one tile tall. The gaps between steps (even-numbered columns) prevent an aTAM program from directly transferring information about the height of one step to the next. Consequently, all information about the height of steps must be passed along the base of the assembly; an infinite tileset is required. However, the CRN-TAM can build and remove probe tiles that allow the assembly to query the previous column. We take advantage of this and show that the CRN-TAM can self-assemble this infinite shape, as shown in Figure 1. Note that we omit the tile and bond declarations but include a graphical representation of the tile species used in the construction. We also omit the CRN species and reactions that ALCH outputs.

[2] See Subsection 3.3 for the one exception.




```
bool at_top;
activate C;
add H; add B; add A;
while (true) {
  at_top = branch {
    true()  { add NHT; add HD; }
    false() { add FT;  add FD; }
  };

  if (at_top) {
    remove HD; remove NHT;
    add NH; add H;
    add B; add A;
  } else {
    remove FD; remove FT;
    add F;
  }
}
```

**Figure 1** An ALCH simulation of the infinite staircase is shown in the upper left. ALCH code for the staircase is shown on the right-hand side. The definitions of the tile types are not shown but are provided visually with bond labels and strengths in the lower left. On the right-most column of the simulation, the $\boxed{FT}$ and $\boxed{FD}$ tiles probe the previous column to detect which tile should be placed. These probe tiles are temporary and are eventually removed. Chemical species and reactions of the staircase construction, as output by ALCH, are not shown. Note that the temperature $\tau$ of the CRN-TAM program is 2.

Intuitively, the self-assembly of the infinite staircase is implemented with a single infinite loop that repeatedly adds tiles to the assembly. Each execution of the loop begins by probing the previous column using the **branch** statement, which nondeterministically attempts to add the sequence of tiles $\boxed{FT}$ and $\boxed{FD}$ or the sequence of tiles $\boxed{NHT}$ and $\boxed{HD}$. If the latter succeeds, the variable at_top is set to **true**, and if the former succeeds, the variable is set to **false**. Notice that the **true**() branch will succeed if and only if the current column is the same height as the previous column because of the top tile $\boxed{H}$. The variable at_top is then used to either (a) finish the current column and initialize the next column or (b) add a single filler tile $\boxed{F}$ and continue with the current column. Using **branch** to query local structural information during the assembly is powerful; we employ a similar technique to show that the discrete Sierpinski triangle can be strictly self-assembled in the CRN-TAM.

We now define each of the language features of the ALCH programming language and explain how they are implemented in the ALCH compiler. We begin by discussing how variables are implemented and define some useful notation that we use to specify what reactions and species are created for each language construct.

The ALCH compiler processes all variable declarations at compile-time. All **BondLabel** and **TileSpecies** variables are added to a symbol table for later reference in **add**, **remove**, **activate**, and **deactivate** statements. Since **BondLabel** and **TileSpecies** variables are immutable and cannot be reassigned, this simple treatment is sufficient. **bool** variables are implemented using two chemical species that are created at compile-time, and we commonly refer to them as *Boolean flags*. A Boolean flag x represents two chemical species $(x, \overline{x})$, where at any given time one of $x$ and $\overline{x}$ has population 0 and the other has population 1. Unlike **BondLabel** and **TileSpecies** variables, **bool** variables are mutable and can be reassigned by switching which species has population 1.

Most ALCH statements are implemented with a set of reactions, and each of their corresponding reactions includes its *line number species* as a reactant. When two statements are executed in sequence, the first statement emits the corresponding line number species

of the second when it is finished. This allows the sequential execution of statements and avoids race conditions during the program execution. For statements that return a **bool**, the compiler creates a dedicated Boolean flag $(x, \overline{x})$ (or, in some cases, links an existing flag) for that line of code and guarantees that when the statement is executed, the associated flag contains the correct value.

When defining how each syntactical element of ALCH is implemented, it is convenient to use notation such as <block> to denote compound ALCH statements and expressions. For example, in the ALCH program in Figure 1, the **if** statement and surrounding code can be written abstractly as:

```
<block1>
if (<block2>) {
    <block3>
}
<block4>
```

Notice how each <block> represents a sequence of statements. Here <block1> must emit the appropriate line number species for the conditional, and similarly, the **if** statement must emit the appropriate line number species for <block4> when it is finished. Since most of these language constructs are implemented with chemical species and reactions, the following notation is convenient:

$$X_{\text{start}} \to \text{<block>} \to X_{\text{end}} \tag{1}$$

Intuitively this notation means that if the line number species $X_{\text{start}}$ is produced, then all the statements corresponding to <block> will be executed. The line number species $X_{\text{end}}$ will be produced afterward. It is important to note that <block> abstractly represents a *sequence* of ALCH instructions, which may themselves use many intermediate line number species. Since some statements return a Boolean flag, we also use $T_{\text{<block>}}$ and $F_{\text{<block>}}$ to denote the true and false species of the returned Boolean flag after <block> is executed.

## 3.1 Boolean Expressions and Variable Assignment

We now discuss how Boolean expressions such as (val1 && val2) || !val3 are evaluated as well as Boolean assignment statements such as **bool** a = <block>. We begin with the logical operations of negation, conjunction, and disjunction.

Given an abstract Boolean expression represented by <block>, we consider the implementation of the logical negation !<block>. Recall that, at compile-time, <block> is given a dual-rail Boolean flag $(x, \overline{x})$. To implement negation, we simply need to return the negated flag $(\overline{x}, x)$. We handle this at compile-time when we link the ! syntax element with the flag of its child element <block>. Intuitively, the compiler will "cross the wires" of <block>'s Boolean flag when it encounters !<block> so that its output flag is negated. Thus negation does not introduce any new species or reactions but rather modifies the output of <block> directly at compile-time so that $T_{\text{<block>}}$ and $F_{\text{!<block>}}$ are the same species and $F_{\text{<block>}}$ and $T_{\text{!<block>}}$ are the same species.

To process a conjunction of logical expressions, we evaluate each expression from left to right and immediately return a false Boolean flag if an expression evaluates to false. Only when all expressions have evaluated to true will a true Boolean flag be returned. Below is

how the conjunction statement <exp1> && <exp2> is implemented:

$$X_{\text{start}} \to \text{<exp1>} \to X_1 \tag{2}$$
$$X_1 + T_{\text{<exp1>}} \to X_2 + T_{\text{<exp1>}} \tag{3}$$
$$X_1 + F_{\text{<exp1>}} \to X_f + F_{\text{<exp1>}} \tag{4}$$
$$X_2 \to \text{<exp2>} \to X_3 \tag{5}$$
$$X_3 + T_{\text{<exp2>}} \to X_t + T_{\text{<exp2>}} \tag{6}$$
$$X_3 + F_{\text{<exp2>}} \to X_f + F_{\text{<exp2>}} \tag{7}$$

Notice how <exp1> is evaluated first, which emits the line number species $X_1$. The line number species together with the species $T_{\text{<exp1>}}$ and $F_{\text{<exp1>}}$ are used to determine whether the expression should immediately return false by producing the $X_f$ line number species or continue by producing $X_2$ to start evaluating <exp2>. This process continues until one expression evaluates to false, or all expressions are true, and the $X_t$ line number species is produced. A dedicated Boolean flag for the conditional is needed for output because the compiler cannot identify any preexisting child element that is guaranteed to hold the correct return value after execution. This Boolean flag is added to the CRN at compile-time, along with the following reactions to update the flag according to whichever $X_t$ or $X_f$ line number species is produced:

$$X_t + T_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{8}$$
$$X_t + F_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{9}$$
$$X_f + T_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{10}$$
$$X_f + F_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{11}$$

Here the species $T_{\text{result}}$ and $F_{\text{result}}$ correspond to the unique Boolean flag generated for this conjunction statement, and $X_{\text{end}}$ is the line number species that initiates the block immediately following the conjunction. We implement logical disjunction in a very similar way: the first time an expression returns true, we immediately return true; if all expressions return false, we return false.

We now describe how Boolean assignment statements such as a = <block> are implemented. To execute this command, we evaluate the right-hand side of the assignment. As discussed above, <block> has an associated Boolean return flag; when <block> finishes execution, this flag is guaranteed to hold the correct return value. We then use the flag species as catalysts to direct execution to the lines of code that set the variable a to true or to false accordingly. Below are the reactions that implement the assignment a = <block>:

$$X_{\text{start}} \to \text{<block>} \to X_1 \tag{12}$$
$$X_1 + T_{\text{<block>}} \to X_t + T_{\text{<block>}} \tag{13}$$
$$X_1 + F_{\text{<block>}} \to X_f + F_{\text{<block>}} \tag{14}$$

The line number species $X_t$ and $X_f$ encode the Boolean return value of <block>, and the following four reactions copy this result into the global Boolean flag for the variable a:

$$X_t + T_a \to X_{\text{end}} + T_a \tag{15}$$
$$X_t + F_a \to X_{\text{end}} + T_a \tag{16}$$
$$X_f + T_a \to X_{\text{end}} + F_a \tag{17}$$
$$X_f + F_a \to X_{\text{end}} + F_a \tag{18}$$

Here $T_a$ and $F_a$ are the species representing the global Boolean flag associated with the variable a. Since we do not know whether $a$ is true or false at compile-time, we must account for both possibilities. Note that we use the <block> Boolean flag species only as catalysts, so the dual-railed representation is preserved.

Since the CRN-TAM requires all reactions to be at most bimolecular, we can use at most one non-line-species product and one non-line-species reactant per reaction. To process information, we must often split computations across several reactions and pass information down in the line number species. Above, for example, the intermediate line number species $X_t$ and $X_f$ serve to temporarily store the return value so we can process it in the following reactions. This and similar patterns frequently occur throughout our implementation of ALCH.

## 3.2 Conditionals and Loops

ALCH also supports conditional execution with the conventional syntax as shown below:

```
if (<exp>) {
    <block>
}
```

The implementation below is similar to the previous constructions above.

$$X_{\text{start}} \to \text{<exp>} \to X_1 \tag{19}$$

$$X_1 + T_{\text{<exp>}} \to X_t + T_{\text{<exp>}} \tag{20}$$

$$X_1 + F_{\text{<exp>}} \to X_{\text{end}} + F_{\text{<exp>}} \tag{21}$$

$$X_t \to \text{<block>} \to X_{\text{end}} \tag{22}$$

We also support **else** blocks by modifying Reaction (21) to output an $X_f$ molecule and adding an additional reaction $X_f \to X_2$ where $X_2$ is the line number species for the **else** block. ALCH also supports **while** loops which are implemented in a similar fashion but alternates between the line number for <exp> and the internal <block>.

## 3.3 Tile Addition, Removal, Activation, and Deactivation

Recall that in the CRN-TAM, every tile species $\boxed{A}$ is associated with at most 1 tile removal signal $A^*$, and the following two sets of reactions.

$$\boxed{\boxed{\alpha}} + \boxed{A} \to \boxed{\boxed{\beta}} + A^* \tag{23}$$

$$\boxed{\boxed{\beta}} + A^* \to \boxed{\boxed{\alpha}} + \boxed{A} \tag{24}$$

Assemblies $\boxed{\boxed{\alpha}}$ and $\boxed{\boxed{\beta}}$ differ only by one instance of $\boxed{A}$, placed in $\boxed{\boxed{\beta}}$. We are given the option to have tiles with no removal signals in the CRN-TAM, but ALCH gives each tile type a unique removal signal. Therefore, we can add a tile by placing it in solution and relying on the first reaction above to attach it to the. We then wait to proceed until we can clean up the tile removal signal that the new tile releases when it bonds to an assembly. The implementation of **add** tileA is as follows where $X_{\text{start}}$ is the line number species of the **add** statement and $X_{\text{end}}$ is the line number species of statement that immediately follows.

$$X_{\text{start}} \to X_1 + \boxed{A} \tag{25}$$

$$X_1 + A^* \to X_{\text{end}} \tag{26}$$

The implementation of **remove** tileA is similar, but it relies on the existence of Reaction (24) discussed earlier:

$$X_{\text{start}} \to X_1 + A^* \tag{27}$$

$$X_1 + \boxed{A} \to X_{\text{end}} \tag{28}$$

Assembly activation is more difficult. The CRN-TAM allows only activation reactions of the form: $X + \boxed{A} \to \boxed{\boxed{A}} + A^*$. There are two difficulties here. First, it is challenging to guarantee that $\boxed{A}$ is activated as a new assembly instead of being added to a preexisting assembly. In order for an activation reaction for $\boxed{A}$ to proceed, we must already have $\boxed{A}$ in solution; if $\boxed{A}$ is in solution, we cannot prevent it from bonding to an existing compatible site. Instead of guaranteeing this explicitly, we rely on users of ALCH to prevent these situations. The second difficulty is that tile activation reactions cannot output a line number species, so we have no easy way of passing execution to the next reaction in our desired sequence. We handle this issue by producing the desired line number species in advance, as shown in the implementation of **activate** tileA below.

$$X_{\text{start}} \to X_1 + X_3 \tag{29}$$

$$X_1 \to X_2 + \boxed{A} \tag{30}$$

$$X_2 + \boxed{A} \to \boxed{\boxed{\alpha}} + A^* \tag{31}$$

$$X_3 + A^* \to X_{\text{end}} \tag{32}$$

Although the line number species $X_3$ is present initially, the last reaction cannot execute until the end, when $A^*$ is also present.

We straightforwardly implement tile deactivation, subject to similar constraints. Instead of temporarily having two line number species in solution, we temporarily have none as we wait for the deactivation reaction to return one.

## 3.4 Nondeterministic Branch Construct

We allow nondeterminism in our language through the **branch** construct. A branch statement contains multiple branch paths; a branch path is a sequence of tile addition and removal instructions collectively associated with a Boolean value. At the start of a branch statement, a program nondeterministically chooses one of the branch paths and begins executing it. Broadly speaking, **branch** returns the Boolean value of the path that ultimately finishes successfully. Each path contains only reversible commands, so if one path is impossible to complete, execution will ultimately reverse out of it and proceed down a different path. Since we require branch paths to be reversible, we allow only **add** and **remove** commands inside **branch** paths. It is possible to support additional commands by making other language constructs reversible, but for our purposes here, **add** and **remove** statements are sufficient.

It is important to note that our notion of reversibility is not complete. For example, suppose we execute **add** tileA inside a branch path. If this statement is reversed, the system will attempt to remove the tile $\boxed{A}$. However, if there are multiple instances of $\boxed{A}$ bonded to the assembly, it is not guaranteed to remove the same tile added earlier in the branch. Additionally, if we add a tile at a strength greater than $\tau$, we will not be able to remove it when attempting to reverse the addition. Any ALCH programmer should exercise caution when using the **branch** statement to avoid such side effects.

```
result = branch {
    true(2) {
        add tileA;
        remove tileB;
    }
    false(1) {
        remove tileC;
    }
};
```

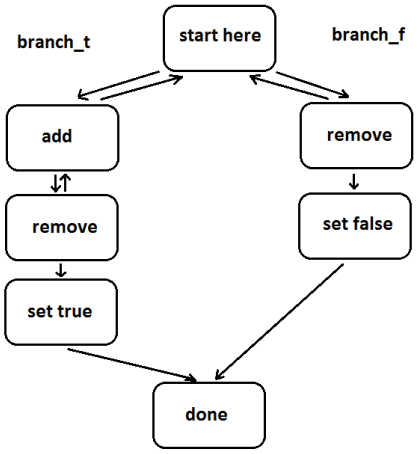



**Figure 2** Possible execution paths through a branch statement. Instructions associated with **true** and instructions associated with **false** are executed nondeterministically via a random walk. The **branch** statement terminates when one path runs to completion, and it returns the corresponding Boolean flag. The integers inside the parentheses of the **true** and **false** branches correspond to *weights* that bias the random walk.

The **branch** statement is implemented with a single branch point that can lead to any one of the branch paths, as shown in Figure 2. From that branch point, we execute only one branch path at a time. Since each branch path is reversible, if execution proceeds down a branch that is incapable of completing, it will eventually return to the branch point via random walk. When a branch finishes execution, we return the Boolean flag that corresponds with the path that completed.

Consider the following **branch** statement where <trueblock> and <falseblock> are arbitrary sequences of **add** and **remove** statements.

```
branch {
    true()  { <trueblock>  }
    false() { <falseblock> }
}
```

The above **branch** statement is implemented in ALCH with the chemical reactions:

$$X_{\text{start}} \leftrightarrow \text{<trueblock>} \rightarrow X_t \tag{33}$$

$$X_{\text{start}} \leftrightarrow \text{<falseblock>} \rightarrow X_f \tag{34}$$

A few things should be noted about the above implementation. First, both the <trueblock> and <falseblock> use the same line number species $X_{\text{start}}$. Second, those reactions are reversible, as indicated by the bidirectional arrows. Third, once one of the blocks finishes, it is completed with an irreversible reaction that terminates the **branch** statement. Fourth, the **add** and **remove** commands outside of **branch** are not reversible; inside branch paths, we modify each add and remove command to make them reversible. The reversible implementation for the **add** statement is shown below.

$$X_1 \leftrightarrow \boxed{A} \tag{35}$$

$$A^* \leftrightarrow X_2 \tag{36}$$

A reversible **remove** statement is implemented in a similar way but is not shown.

The last thing to note about the **branch** statement is that it returns a Boolean flag. Therefore a dedicated flag must be created at compile-time and be appropriately set after the execution is completed. Therefore the following reactions are also needed to set this Boolean flag.

$$X_t + T_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{37}$$
$$X_t + F_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{38}$$
$$X_f + T_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{39}$$
$$X_f + F_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{40}$$

## 4 Strict Self-Assembly of the Discrete Sierpinski Triangle

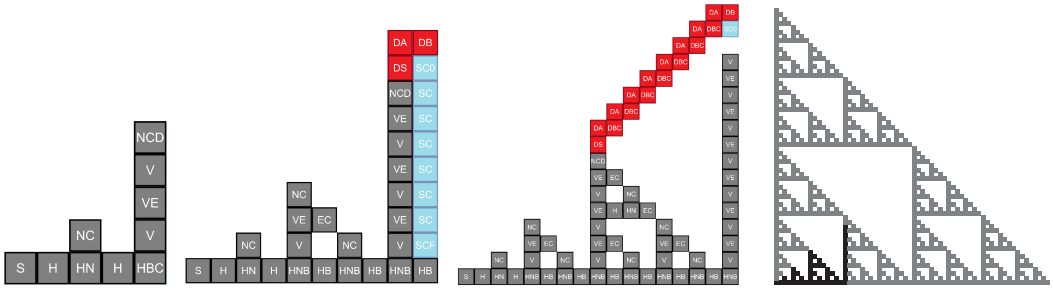
We now present the CRN-TAM construction that strictly self-assembles the discrete Sierpinski triangle (DST) using ALCH. Our discussion here is complete but brief; see Appendix A for a more detailed description of our algorithm. To see the complete specification of the construction in ALCH, along with a video visualization of the self-assembly, see `http://web.cs.iastate.edu/~lamp/`.

We begin with an overview of tile types and a brief description of their purpose and then describe the DST construction algorithm in detail. Since the DST is symmetric about the line $f(x) = x$, we refer to the two symmetric halves as the lower symmetric triangle (LST) and the upper symmetric triangle (UST). We first discuss the techniques to strictly self-assemble the LST, which can be easily modified to construct the UST in parallel. In our construction, it is useful to distinguish between three types of tiles: (1) structural tiles, (2) scaffold tiles, and (3) probe tiles. Structural tiles are permanent and form the DST itself. Scaffold tiles are used to construct temporary auxiliary structures to facilitate the DST construction. Probe tiles are rapidly added and removed to query existing information of previously placed structural tiles. To avoid unwanted crosstalk between the symmetric halves, we duplicate the set of structure tiles into a symmetric group with bonds that are incompatible with the LST tiles. We also differentiate the tile types of even and odd columns to prevent a partially constructed column from interfering with the construction.

We now discuss the construction for the strict self-assembly of the DST. The first step in our construction unpacks the initial structure shown in Figure 3a with hard-coded tile activation and addition statements. This is easily accomplished by adding tiles in a specific order that avoids ambiguity in placement. After the initial structure tiles are placed, we then construct the LST column by column, adding structure tiles one-at-a-time, completing each column before proceeding to the next. We also use a variable to track whether we are currently constructing an even or odd column. The process of adding one structure tile at a time is akin to a dot-matrix printer, placing dots of ink one line at a time.

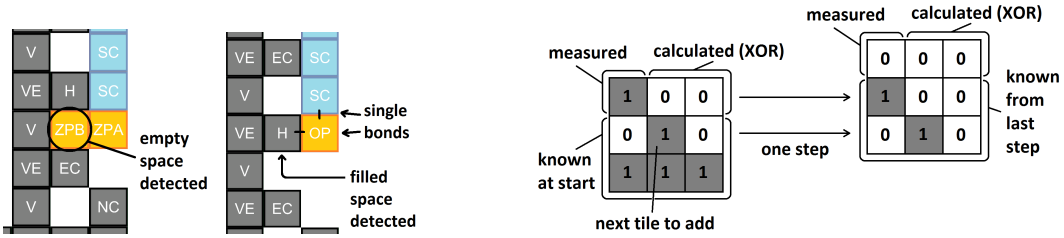
### 4.1 Scaffold Construction

We construct two types of scaffolds. The diagonal scaffold, shown in red in Figure 3, runs along the diagonal of the DST and provides an anchor for the vertical scaffold, which is shown in cyan. The vertical scaffold covers up potential bond sites that we do not wish to bond to, as illustrated in Figure 3b. The diagonal scaffold is straightforward to construct; before constructing each column, we extend it out by two more tiles. For the vertical scaffold, we must extend it only as far as the base of the DST. We extend the DST base row out by



**(a)** Seed. **(b)** Vertical scaffold. **(c)** Diagonal scaffold. **(d)** Sierpinski triangle.

**Figure 3** (a) The initial hard-coded structure upon which we build the lower half of the DST. (In the final program that constructs the whole DST, this structure has a symmetric upper half.) (b) Demonstrates how our construction extends a vertical scaffold down to occlude all the potential tile bond sites on column currently being constructed. (c) Shows the diagonal scaffold before erasing itself and starting a new diagonal scaffold. (d) Shows a section of the Sierpinski triangle that includes the lower and upper symmetric halves; the part corresponding to (c) is highlighted.




**(a)** Illustrates how the probe detects empty spaces in the Sierpinski triangle; both paths are attempted in parallel.

**(b)** Illustrates how the next $3 \times 3$ window around the probe is updated using the previous window and the tile detected by the probe.

**Figure 4** Visualization of the probe querying nearby tiles and updating the $3 \times 3$ window.

one space to denote the bottom of the vertical scaffold. We begin the vertical scaffold with $\boxed{SC_0}$ and construct most of it from vertically double-bonded $\boxed{SC}$ tiles. We use $\boxed{SC_0}$ so that we know when we are done when removing the scaffold.

The special final tile $\boxed{SC_f}$ has a single bond on its north and south edges; it cannot attach until it can bond cooperatively with the base tile below it and the scaffold tile above it. When our system succeeds at placing $\boxed{SC_f}$, it knows to continue to the next phase. We allow the assembly to remove $\boxed{SC}$ as well, in case $\boxed{SC}$ bonds at the bottom instead of $\boxed{SC_f}$; scaffold construction proceeds as a random walk, which we bias with reaction rates.

Since the diagonal scaffold is not part of the DST, we must periodically clean it up. Some columns in the LST are entirely solid up to the diagonal; when we encounter one of these, we destroy the existing diagonal and begin a new diagonal starting from the top of the solid column. As with $\boxed{SC_0}$, we start with a special diagonal tile so that we can remove the diagonal in a loop and know when to stop.

## 4.2 Adding Structure Tiles with the Probe

When beginning to place tiles on a new column $i$, the vertical scaffold must be completely initialized as in Figure 3b. We must know which tile, if any to add to the DST at each vertical position: T-joint, straight connector, etc. To that end, after constructing the vertical

scaffold, we initialize a $3 \times 3$ Boolean grid, centered on $(i, 1)$, of Boolean flag variables. This grid stores whether those tile positions are occupied in the full DST; note that if we know the $3 \times 3$ grid around a position, we know which tile, if any, goes there. The lower six squares are entirely determined by whether $i$ is even or odd; the lowest row of the LST is solid, and the second-lowest alternates every space between filled and empty. To determine the upper-left space, we use the "probe" to measure whether $(i-1, 2)$ is filled or empty in column $i-1$, which we have already constructed. We do this by nondeterministically attempting to build two structures in parallel, as shown in Figure 4a, and can deduce the value of $(i-1, 2)$ based on which one succeeds. If the upper left space $(i-1, 2)$ is empty, then it is possible to place a tile there; using double-bonded probe tiles, we build south from the scaffold and then west into the potential empty space. If this construction succeeds, we know that the space is empty. We exploit cooperative bonding to determine if $(i-1, 2)$ is filled. Structure tiles connect to each other with double bonds; each structure tile, however, has at least a single bond on its east edge. Our probe tile, then, has a single bond on its north and west edges. It can bond cooperatively with the scaffold and space $(i-1, 2)$ only if $(i-1, 2)$ is filled. We use ALCH's `branch` structure to nondeterministically try both paths until one succeeds, at which point our program knows the upper-left space of the $3 \times 3$ grid. We can then calculate the upper-center and upper-right spaces using the XOR characterization of the DST.

With the grid filled in, our program can put the correct tile into solution (or skip forward if no tile is required). All incorrect bond sites in column $i$ are covered by the vertical scaffold, so our tile is guaranteed to bond at the correct location. We must then "slide" the $3 \times 3$ grid one space north (updating the Boolean flags accordingly) to process the next tile site, as illustrated in Figure 4b. The lowest six spaces of the new grid overlap with the old grid, so we already know them. As during initialization, we can calculate the upper-left space using the probe method and the remaining two using XOR. We proceed in this fashion up the entire column until it is completed. Note that when adding tiles in the middle of column $i$, we must make sure they do not bond into column $i+1$ using bond sites on the part of column $i$ that we have already constructed. We use even and odd bond types to prevent this; the tiles we add for column $i$ are incompatible with the bond sites in column $j$.

## 4.3 Constructing the Upper Symmetric Triangle

We have discussed how to construct the lower symmetric triangle (LST); it is straightforward to extend this method to the upper symmetric triangle (UST). Since the DST is symmetric, we need not track any additional information. We generate a symmetric scaffold corresponding to the vertical scaffold discussed above. (Since the diagonal scaffold is off-center, we skip the symmetric version of $\boxed{SC_0}$.) When we add a structure tile to the LST, we add its symmetric version as well. We must also make a straightforward modification to our method for finishing off the solid columns (rows in the UST) that signal diagonal scaffold cleanup; see the appendix for details.

# 5 Conclusion

In this paper, we define ALCH, a programming language for the CRN-TAM, and use it to exhibit a strict self-assembly of the discrete Sierpinski triangle (DST). Our use of ALCH allows us to conceptualize our construction at the level of imperative tile commands and familiar control structures like conditionals and while loops. Furthermore, since it is impossible to strictly self-assemble the DST in the aTAM, our construction serves as a proof that the CRN-TAM can strictly self-assemble infinite shapes that the aTAM cannot.

We have utilized two new techniques in our DST construction. First, we have used a *probe* mechanism to measure which tiles have been placed, allowing us to derive information from the already-constructed system. The probe technique showcases ALCH's nondeterministic branch structure, exploring multiple potential executions to find one that can complete. It also enables us to query the parts of the DST we have already constructed. Second, we have used a temporary scaffold to *occlude* undesirable tile bonding sites and precisely control where new tiles are added. Both of these techniques leverage the CRN-TAM's ability to remove tiles and create temporary structures.

We considered an alternate strategy to construct the DST using a CRN-TAM Turing machine implementation to control scaffold construction and tile placement. This entailed maintaining a secondary representation of the partially-constructed DST in the Turing machine tape, updating and querying it as the construction proceeds. The Turing machine would likely require unbounded storage to retain the last-constructed column even if it does not store the whole DST. On the other hand, our CRN-TAM construction acts as a "transformer," converting a stream of local data into a stream of tile placements without retaining unbounded information. The only part of the DST that we store in a computational form is the local $3 \times 3$ grid. We update it using the probe mechanism, thereby converting measurements of the existing DST into a bounded representation of the local DST area.

Our second technique, occluding bond sites with a temporary scaffold, is very general; we can apply it to any construction where we have a frontier of potential bond sites and must bond at a precise one. We expect this technique to be useful in constructing a wide variety of infinite shapes in the CRN-TAM. Our DST construction does not require a Turing machine, but the full power of CRN-TAM universality is available to use in combination with occlusion scaffolds. We speculate that it is possible to construct every connected recursively enumerable subset of $\mathbb{Z}^2$ using variants of this technique.

For the current version of ALCH, we have focused on a very sequential programming model. However, the CRN-TAM, allows for potentially massive parallelism via large chemical populations; it would be interesting to explore additional ALCH features that leverage this capability. For example, the aTAM tileset design toolkit by Doty and Patitz [4] provides an abstraction for highly-parallel tile assembly. Incorporating a similar tool into ALCH could enable powerful constructions that combine chemical parallelism with the coordination capabilities of ALCH's imperative framework. More broadly, we speculate that ideas from classical concurrent programming are relevant to ALCH as well.

We hope that the tools and techniques presented here will catalyze research into the CRN-TAM and similar hybrid models.

## References


1 Florent Becker. Pictures worth a thousand tiles, a geometrical programming language for self-assembly. *Theoretical Computer Science*, 410(16):1495–1515, 2009. Theory and Applications of Tiling. doi:10.1016/j.tcs.2008.12.011.

2 Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In *Algorithmic Bioprocesses*, Natural Computing Series, pages 543–584. Springer, 2009. doi:10.1007/978-3-540-88869-7_27.

3 David Doty, Jack H Lutz, Matthew J Patitz, Robert T Schweller, Scott M Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *Proceedings of the 53rd Symposium on Foundations of Computer Science*, pages 302–310. IEEE, 2012. doi:10.1109/FOCS.2012.76.

4 David Doty and Matthew J. Patitz. A domain-specific language for programming in the tile assembly model. In *Proceedings of the 17th International Conference on DNA Computing*

*and Molecular Programming*, pages 25–34. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-10604-0_3.

5 Irving Robert Epstein and John Anthony Pojman. *An Introduction to Nonlinear Chemical Dynamics: Oscillations, Waves, Patterns, and Chaos*. Oxford University Press, 1998. doi:10.1021/ed077p450.1.

6 Martin Feinberg. *Foundations of chemical reaction network theory*. Springer, 2019. doi:10.1007/978-3-030-03858-8.

7 David Furcy, Scott M. Summers, and Christian Wendlandt. New bounds on the tile complexity of thin rectangles at temperature-1. In *Proceedings of the 25rd International Conference on DNA Computing and Molecular Programming*, pages 100–119. Springer International Publishing, 2019. doi:10.1007/978-3-030-26807-7_6.

8 James I. Lathrop, Jack H. Lutz, and Scott M. Summers. Strict self-assembly of discrete Sierpinski triangles. *Theoretical Computer Science*, 410(4):384–405, 2009. doi:10.1016/j.tcs.2008.09.062.

9 Anthony M. L. Liekens and Chrisantha T. Fernando. Turing complete catalytic particle computers. In *Advances in Artificial Life*, pages 1202–1211. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74913-4_120.

10 Pierre-Étienne Meunier and Damien Woods. The non-cooperative tile assembly model is not intrinsically universal or capable of bounded Turing machine simulation. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 328–341. ACM, 2017. doi:10.1145/3055399.3055446.

11 Nicholas Schiefer and Erik Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming*, pages 34–54. Springer International Publishing, 2015. doi:10.1007/978-3-319-21999-8_3.

12 Nicholas Schiefer and Erik Winfree. Time complexity of computation and construction in the chemical reaction network-controlled tile assembly model. In *Proceedings of the 22nd International Conference on DNA Computing and Molecular Programming*, pages 165–182. Springer International Publishing, 2016. doi:10.1007/978-3-319-43994-5_11.

13 Nadrian C. Seeman. Nucleic acid junctions and lattices. *Journal of Theoretical Biology*, 99(2):237–247, 1982. doi:10.1016/0022-5193(82)90002-9.

14 Marko Vasić, David Soloveichik, and Sarfraz Khurshid. CRN++: Molecular programming language. *Natural Computing*, pages 1–17, 2020. doi:10.1007/s11047-019-09775-1.

15 Erik Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998. URL: https://resolver.caltech.edu/CaltechETD:etd-05192003-110022.

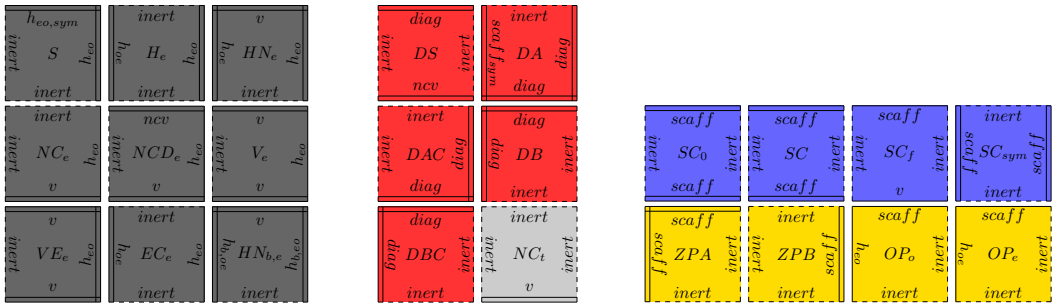
# A Strict DST Construction: Details

We now present a more detailed look into our DST construction in ALCH. We begin with an overview of tile types and a brief description of their purpose; we then describe the DST construction algorithm in detail.

The DST is symmetric about the line $y = x$. We refer to the two symmetric halves as the lower symmetric triangle (LST) and the upper symmetric triangle (UST). We will focus on the LST construction algorithm, as it can be easily modified to construct the UST at the same time.

## A.1 Tile Types

We distinguish three types of tiles. Structure tiles form the DST itself. Scaffold tiles form semi-permanent auxiliary scaffolds that enable us to build the DST, and probe tiles are added and removed quickly to probe the existing structure for useful information. One commonality between all three tile types is the *inert* bond label, which we use always at strength 0.




**(a)** These are the structure tiles that form the odd columns of the LST; we omit the even column tiles and the tiles for the entire UST, which are very similar.

**(b)** These tiles form the scaffolding that runs along regions of the southwest-to-northeast diagonal.

**(c)** The blue tiles form the vertical scaffolding that obscures bond sites to facilitate adding tiles at specific locations. The yellow tiles form the probes that determine whether a position in the previous column is filled or empty.

**Figure 5** Tiles types used in the DST construction.

### A.1.1 Structure Tiles

Structure tiles use several bond labels for the LST.

- $v$ is a strength 2 vertical bond that joins structure tiles in completed regions of the DST.
- $h_{eo}$ and $h_{oe}$ are likewise structural horizontal bonds. We must disambiguate between even and odd columns; $h_{eo}$ joins an even-column tile on the left with an odd-column tile on the right, and $h_{oe}$ is the reverse.
- $h_{b,eo}$ and $h_{b,oe}$ are variants that mark the lowest ("base") row in the DST.
- $ncv$ interfaces structure tiles with one type of scaffold tiles.

To avoid unwanted crosstalk between the symmetric halves, we duplicate the set of structure tiles into a symmetric group with bonds that are incompatible with the LST tiles. Likewise, we use separate bond labels and tile types to avoid crosstalk between even and odd columns. This produces four similar categories of structure tile: even LST, odd LST, even UST, and odd UST. We present a list of even LST tiles in Figure 5a. Note that most structure tiles have an $h_{eo}$ bond of strength at least one on their eastern edges so that probe tiles can attach cooperatively.

Tile $\boxed{S}$ is the seed tile that we activate to form the southwest corner of the DST. Tile $\boxed{NCD_e}$ interfaces between the structure and the scaffold, and tile $\boxed{HN_{b,e}}$ is a variant tile type that occurs specifically on the lowest row. All the other structure tile types in Figure 5a fill in the DST structure in a straightforward way.

### A.1.2 Scaffold Tiles

We use two types of scaffolds. The vertical scaffold extends along the eastern face where the next column is to be added; we extend and retract it to expose structural tile addition sites. The diagonal scaffold extends along parts of the southwest-to-northeast diagonal and provides an attachment point for the vertical scaffold. We require two additional bond labels: $scaff$ for the vertical scaffold and $diag$ for the diagonal scaffold.

See Figure 5b for a list of diagonal scaffold tiles. Tile $\boxed{DS}$ interfaces with the structure tiles and begins the diagonal scaffold; tiles $\boxed{DA}$ and $\boxed{DB}$ form the body of the diagonal. Since $\boxed{DA}$ and $\boxed{DB}$ contain bond sites to begin the vertical scaffolds, when we finish with a column we must replace them with the capped variants $\boxed{DAC}$ and $\boxed{DBC}$ so future vertical scaffolds don't spuriously bond there. We use $\boxed{NC_t}$ as a temporary variant of $\boxed{NC_e}$ that is useful for cleaning up the scaffold.

We present a list of vertical scaffold tiles in Figure 5c. Tiles $\boxed{SC}$ and $\boxed{SC_{sym}}$ form the body of the vertical and symmetric horizontal scaffolds. We use $\boxed{SC_0}$ and $\boxed{SC_f}$ at the beginning and end of the LST vertical scaffold so that we can identify when we are done adding and removing it; since we know this information from the LST, we don't require corresponding symmetric tile species.

### A.1.3 Probe Tiles

When constructing a new column, we use a probe mechanism to determine whether specific rows in the last constructed column contain structure tiles; this allows us to use $XOR$ to reconstruct the DST with constant information stored in chemical species counts. We have separate probe mechanisms to detect "zeros" (empty positions) and "ones" (filled positions). See Figure 5c for a list of probe tiles.

## A.2 Initialization

We now begin our discussion of the DST construction algorithm. First, we prepare the structure shown in Figure 3a, using a straightforward series of tile additions that do not result in ambiguity. We also initialize to odd the flag that tracks whether we are in an even or an odd column.

We face two challenges when constructing the rest of the triangle:

- We must add each tile in the correct location, instead of any of a potentially unbounded number of incorrect locations.
- At each position, we must determine which tile to add, if any; i.e., we must know whether to add nothing, $\boxed{EC_e}$, $\boxed{HN_e}$, etc.

We solve the first problem by tracking the tile positions around the tile position in question. To solve the second problem, we extend a scaffold of tiles to occlude all unintended bond sites.

To begin, we add the diagonal scaffold tile $\boxed{DS}$ above $\boxed{NCD_e}$; this will be the start of our occluding scaffold. Immediately after $\boxed{DS}$ is added, we enter a loop construct in our algorithm. We will refer to this loop as the outer loop; each outer loop iteration constructs another column of the LST.

## A.3 Outer Loop

### A.3.1 Initialization: building the scaffold

Inside the loop, we must first build out the scaffold. We add $\boxed{DA}$ and $\boxed{DB}$ to $\boxed{DS}$, and we extend the base row with $\boxed{H_{b,o}}$ (or $\boxed{HN_{b,e}}$ in an even row). Since we have a tile set specifically for constructing the base layer, we don't need to worry about adding $\boxed{H_{b,o}}$ in the wrong row.

Now, we construct the vertical scaffold down from $\boxed{DB}$ to produce a structure like the one shown in Fig 3b. We add $\boxed{SC_0}$ first so that when we remove it again we will know we have reached the top; as discussed below, $\boxed{SC_f}$ is a mechanism to detect the bottom row.

We then add $\boxed{SC}$ until we reach the bottom row. Since we have added $\boxed{H_{b,o}}$ extending out, we cannot add $\boxed{SC}$ at row 0 or lower. We must detect when we reach the bottom, however, so we can stop attempting to add $\boxed{SC}$ and continue with the rest of the program.

Whenever we attempt to add $\boxed{SC}$, we also attempt to add $\boxed{SC_f}$ in parallel using the branch structure. Recall that $\boxed{H_{b,o}}$ always has a bond site on its north edge; since $\boxed{SC_f}$ has single bonds on its north and south edges and must bond at strength 2, it can only bond in row 1 between $\boxed{H_{b,o}}$ to the south and $\boxed{SC}$ to the north.

It may be that $\boxed{SC}$ bonds in row 1 instead of $\boxed{SC_f}$; we always add $\boxed{SC}$ reversibly so that if this happens the program can proceed (and can *only* proceed) by removing $\boxed{SC}$. In this way we have as many chances as we need to add $\boxed{SC_f}$ and continue with the program. We attempt to add $\boxed{SC}$ in one branch and $\boxed{SC_f}$ in another; the return value tells us whether we have finished adding the scaffold.

### A.3.2 Guaranteeing correct added tile position

We can now remove the vertical scaffold row by row, exposing only one tile addition site at a time. There are two types of addition sites: north and east edges of preexisting tiles. We claim that when we add a new structure tile, at most one potential bond site is exposed, so the tile is added unambiguously.

Recall that we have separate bond types for even and odd columns; an odd-column structure tile cannot bond to the east side of another odd-column structure tile, and likewise with even columns. If we are building column $i$, then, we don't need to worry about unintended bonding in column $i+1$. In column $i$ itself, the region above our intended bond site is covered by vertical scaffold tiles and is therefore not a concern. In the region below our intended bond site, all viable bond sites have already been taken up. We can therefore guarantee that we can always add the next DST structure tile unambiguously.

### A.3.3 Choosing the correct tile

Now that we can guarantee that tiles are added at the correct position, we must determine which tile to add and whether or not to add one at all.

We store a $3 \times 3$ "window" of boolean flags around the tile position where we will potentially add a tile, as shown in Fig. 4b. Each flag is true or false based on whether the corresponding position in the DST is full or empty. Note that if we possess this information, it is easy to determine whether we must add the center tile, and, if so, which tile we must add.

If we are constructing column $i$, we have added the first tile $\boxed{H_{b,o}}$ or $\boxed{HN_{b,e}}$ at position $(i, 0)$. We will therefore initialize the $3 \times 3$ grid centered on $(i, 1)$, which is the next potential tile position to fill. The bottom row is always filled in all three positions by the base row, so we can initialize the lower three flags to true. The second row in the DST always alternates between full and empty, so we need to set either the center flag or the center-left and center-right flags to true depending on whether we are building an odd or an even column. Since we track this information, we can easily initialize the middle row.

We do not immediately have enough information to initialize the upper tiles. Recall, however, that the DST can be characterized as a cellular automaton based on the XOR relation $\oplus$:

$$DST[x, y] \leftrightarrow DST[x-1, y] \oplus DST[x, y-1]. \tag{41}$$

Therefore if we could somehow measure the upper-left tile, we could calculate the upper-center and upper-right tiles.

We can measure the upper-left tile $(i-1,2)$ using the branch construct. We require two series of tile additions: one that is only possible if $(i-1,2)$ is empty, and one that is only possible if it contains a tile.

If $(i-1,2)$ is empty, we can add a "zero probe" tile into that location; we therefore attempt to add such a tile, first building $\boxed{ZPA}$ down from the scaffold and then attempting to build $\boxed{ZPB}$ at $(i-1,2)$. See Fig. 4a for an illustration.

Recall that all structural tiles have an east bond site of strength at least one. We therefore attempt in parallel to add a "one probe" $\boxed{OP}$ with strength-one north and west bond sites. If there is a structural tile in $(i-1,2)$, then the one probe can bond cooperatively with it and the vertical scaffold, as shown in Fig. 4a.

We perform these attempts in parallel using the branch construct. It is possible that $\boxed{ZPA}$ will bond when $(i-1,2)$ is full. Since we add $\boxed{ZPA}$ reversibly, this is not a problem; the program can only proceed by removing $\boxed{ZPA}$, and $\boxed{OP}$ then has another chance to bond. It is clear, then, that only the correct branch can fully complete. When it does, the branch statement returns the correct value of $(i-1,2)$.

Once we know $(i-1,2)$, we can calculate the upper-center tile value in our $3\times 3$ window using the XOR characterization of the DST. We can then similarly calculate the upper-right tile; that completes the grid, and we can add the appropriate tile into the exposed bond site or skip it if no tile is required.

We must then adjust the grid so it is centered on $(i,2)$ instead of $(i,1)$. Note that the lower two rows of the new grid must be the same as the upper two rows of the old grid, which we have already calculated and stored. We therefore need to calculate only the top row. We can measure $(i-1,3)$ in the same way we measured $(i-1,2)$; this allows us to calculate the new top row the same way we calculated the old top row. We can then continue adding or skipping tiles and sliding the grid upwards iteratively as we construct the column; one sliding iteration is shown in Fig. 4b.

At some point we will attempt to remove $\boxed{SC}$ and instead remove $\boxed{SC_0}$; we detect this with the branch construct and terminate the column loop. This also signals the end of the outer loop. We remove $\boxed{DB}$ and replace it with $\boxed{DBC}$ so that new $\boxed{SC}$ and $\boxed{SC_0}$ tiles can't bond there and restart the outer loop.

## A.4 Cleaning Up the Diagonal

As we build additional columns, we extend the diagonal scaffold along the diagonal of the DST; since it is not part of the DST, we must periodically remove it to "clean up" our construction.

At every horizontal coordinate that is a power of 2, the DST contains a column of filled cells that extends all the way from the baseline to the diagonal, as shown in Fig 3c. We can detect this in our program by inspecting the state of the $3\times 3$ grid when we remove $\boxed{SC_0}$; if we have just completed a column of filled cells, we clean up the diagonal in the region to the left of the completed column.

We began the diagonal scaffold with a special tile $\boxed{DS}$, just as we begin the vertical scaffolds with $\boxed{SC_0}$. We can therefore remove $\boxed{DA}$ and $\boxed{DBC}$ repeatedly until we can instead remove $\boxed{DS}$ with the branch construct. When we remove $\boxed{DS}$, we have cleaned up the scaffold.

Recall that there is a specific tile $\boxed{NCD_e}$ with a north bond site that allows the diagonal scaffold to connect. On the old column that supported the diagonal scaffold, we must replace $\boxed{NCD_e}$ with $\boxed{NC_e}$; otherwise when we attempt to add the diagonal scaffold, it might bond

on the old column instead of the new one. We must also ensure that $\boxed{NCD_e}$ is at the top of the new column. To facilitate this swap without risking an unintended tile placement, we place a new tile species $\boxed{NC_t}$ as a temporary cap on the new column. At the end of this process, all tile positions to the left of the new column correctly match the LST, with no excess scaffold. Since we repeat this process iteratively at farther and farther positions, we are strongly constructing the LST.

## A.5 Constructing the Upper Symmetric Triangle

We have shown a construction of the lower symmetric triangle (LST) of the DST. We can construct the upper symmetric triangle (UST) at the same time using a very similar mechanism. There is no need to calculate the $3 \times 3$ grid for the UST, as we already know its symmetric version for the LST.

We duplicate the tileset that we used to construct the LST so that there is no tile placement ambiguity between symmetric halves. With a few exceptions, discussed below, whenever we add or remove a vertical scaffold or structure tile in the LST, we also add or remove the symmetric tile in the UST. Also, since the horizontal scaffold bonds onto $\boxed{DA}$, we must replace $\boxed{DA}$ with $\boxed{DAC}$ at the end of the outer loop.

The diagonal scaffold is not entirely symmetric across the diagonal axis, so we must make several adjustments. First, since the diagonal scaffold occupies the spaces where $\boxed{SC_0}$ would go in the UST, we do not add a symmetric $\boxed{SC_0}$; we attach the horizontal scaffold directly onto the diagonal scaffold. We rely on the $\boxed{SC_0}$ tile in the LST to inform horizontal scaffold removal. Second, every power-of-two row in the UST intersects with the diagonal at one grid point; we fill in that grid point manually every time we clean up a section of the diagonal scaffold.

With these modifications, our ALCH program strongly constructs the full DST in the CRN-TAM.

# Implementing Non-Equilibrium Networks with Active Circuits of Duplex Catalysts

**Antti Lankinen**
Department of Bioengineering, Imperial College London, UK
antti.lankinen15@imperial.ac.uk

**Ismael Mullor Ruiz**
Department of Bioengineering and Imperial College Centre for Synthetic Biology, Imperial College London, UK
i.mullor-ruiz16@imperial.ac.uk

**Thomas E. Ouldridge**
Department of Bioengineering and Imperial College Centre for Synthetic Biology, Imperial College London, UK
t.ouldridge@imperial.ac.uk

## Abstract

DNA strand displacement (DSD) reactions have been used to construct chemical reaction networks in which species act catalytically at the level of the overall stoichiometry of reactions. These effective catalytic reactions are typically realised through one or more of the following: many-stranded gate complexes to coordinate the catalysis, indirect interaction between the catalyst and its substrate, and the recovery of a distinct "catalyst" strand from the one that triggered the reaction. These facts make emulation of the out-of-equilibrium catalytic circuitry of living cells more difficult. Here, we propose a new framework for constructing catalytic DSD networks: Active Circuits of Duplex Catalysts (ACDC). ACDC components are all double-stranded complexes, with reactions occurring through 4-way strand exchange. Catalysts directly bind to their substrates, and the "identity" strand of the catalyst recovered at the end of a reaction is the same molecule as the one that initiated it. We analyse the capability of the framework to implement catalytic circuits analogous to phosphorylation networks in living cells. We also propose two methods of systematically introducing mismatches within DNA strands to avoid leak reactions and introduce driving through net base pair formation. We then combine these results into a compiler to automate the process of designing DNA strands that realise any catalytic network allowed by our framework.

**2012 ACM Subject Classification** Hardware → Biology-related information processing

**Keywords and phrases** DNA strand displacement, Catalysis, Information-processing networks

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.7

**Supplementary Material** A compiler to generate optimal sequences for each strand in any allowed catalytic network is available at `https://zenodo.org/record/3948343`.

## 1 Introduction

DNA is an attractive engineering material due to the high specificity of Watson-Crick base pairing and well-characterised thermodynamics of DNA hybridisation [13, 40], which give DNA the most predictable and programmable interactions of any natural or synthetic molecule [43]. DNA computing involves exploiting these properties to assemble computational devices made of DNA. The computational circuits are typically realised using DNA strand displacement (DSD) reactions, in which sections of DNA strands called *domains* with partial or full complementarity hybridise, displacing one or more previously hybridised strands in the process [55]. DSD is initiated by the binding of short complementary sequences called *toeholds*. It is helpful to divide DSD reactions into a few common reaction steps, including:

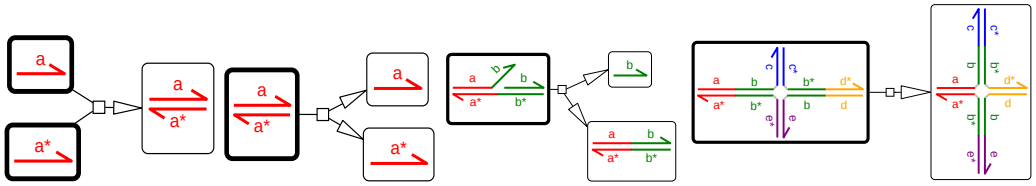

binding, unbinding, and three- or four-way strand displacement and branch migration, shown in Figure 1. DSD is an attractive scheme for computation as it can be used as a medium in which to realise chemical reaction networks (CRNs) [44], which provide an abstraction of systems exhibiting mass-action chemical kinetics and have been shown to be Turing complete [27]. DSD is then Turing complete as well [34,52]. DSD has been used to construct, for example, logic circuits [35,42], artificial neural networks [9,17,38], dynamical systems [46], catalytic networks [8,36,56], and other computational devices [1,53]. To facilitate testing and realisation of DSD systems, domain-level design tools [23,45] as well as domain-to-sequence translation [54] software have been introduced.

While DNA nanotechnology is concerned with using DNA as a non-biological material, a key goal of DNA nanotechnology is the imitation and augmentation of cellular systems. It is therefore worth considering how these natural systems typically perform computation and information processing. One ubiquitous biological paradigm for signal propagation and processing is the catalytic activation network, as exemplified by kinases [20, 28, 29]. Kinases are catalysts that modify substrates by phosphorylation and consume ATP in the process. These substrates can be, for example, transcription factors, but can also be kinases themselves that are either activated or deactivated by phosphorylation. The opposite function, dephosphorylation, is performed by phosphatases [4]. The emergent catalytic network then performs information propagation or computation by converting species, kinases and phosphatases, between their active and passive states. Kinase cascades are featured in many key biological functions, such as cellular growth, adhesion, and differentiation [28, 51] and long-term potentiation [47].

Most catalytic networks - including many with a simple topology and a constant steady state, such as a a single kinase and phosphatase species competing to activate/deactivate a substrate - operate out of equilibrium and consume fuel even in their steady state. This fuel-consuming, non-equilibrium behaviour is vital in allowing them to perform functions such as signal splitting, amplification, time integration and insulation [5, 12, 18, 30, 31]. Moreover, since the key molecular species are recovered rather than consumed by reactions, catalytic networks can operate continuously, responding to stimuli as they change over time - unlike many architectures for DSD-based computation and information processing that operate by allowing the key components to be consumed [1, 9, 38]. This ability to operate continuously is invaluable in autonomous environments such as living cells.

In this work, we propose a minimal mechanism for implementing reaction networks of molecules that exist in catalytically active and inactive states, a simple abstraction of natural kinase networks. In these catalytic activation networks, we implement arbitrary activation reactions of the form $A^{\text{on}} + B^{\text{off}} + \sum_i F_i \rightarrow A^{\text{on}} + B^{\text{on}} + \sum_i W_i,\ i \in \{1, 2, 3...\}$. Here, the catalyst A in its active state $A^{\text{on}}$ drives B between its inactive and active states ($B^{\text{off}}$, $B^{\text{on}}$) by the conversion of one or more fuel molecules $\{F_i\}$ into waste $\{W_i\}$. Equivalent deactivation reactions in which an active catalyst deactivates a substrate are also considered.

The rest of this paper is organised as follows. In Section 2, we propose and motivate the concept of a direct bimolecular catalytic reaction and consider the necessary conditions for DSD species that are able to perform such reactions. Section 3 introduces a novel DSD framework to implement these reactions, and its computational properties are analysed in Section 4. Based on these findings, we propose a systematic method of introducing mismatched base pairs within species in our framework to improve its function in Section 5. We combine our findings and propositions into a software to automate the sequence-level design of any CRN that is realisable within our framework, and detail this software in Section 6. In Section 7, we discuss our framework, findings, and future work. We conclude the paper in Section 8.




**(a)** Bind. **(b)** Unbind. **(c)** Displace (3-way). **(d)** Branch migrate (4-way).

**Figure 1** Basic reaction steps in the DSD formalism, as represented by Visual DSD [23]. Each domain is represented by a letter and a colour. "*" denotes the Watson-Crick complement. The barbed end of a strand indicates the 3' end.

## 2 Direct Action of Molecular Catalysts

Catalytic processes are those in which a species is both a reactant and a product. Such processes cannot result from an individual elementary reaction of binding, unbinding or unimolecular state change; catalysis is therefore only defined at the level of the overall stoichiometry of a series of elementary reactions. As a corollary, the same overall stoichiometry can result from many different combinations of elementary steps.

In kinase cascades, functional changes in substrates are a result of direct binding of the catalyst to the substrate. Moreover, the essential products of the reaction (the activated substrate and recovered catalyst) are the same molecules that initially bound to each other - albeit with some modification of certain residues, or turnover of small molecules such as ATP or ADP to which they are bound. Motivated by these facts, we propose the following definition for a direct bimolecular catalytic activation reaction.

▶ **Definition 1** (Direct bimolecular catalytic activation). *Consider the (non-elementary) process*

$$A^{\text{on}} + B^{\text{off}} + \sum_i F_i \rightarrow A^{\text{on}} + B^{\text{on}} + \sum_i W_i, \ i \in \{1, 2, 3...\},$$

*where* A *in active state* $A^{\text{on}}$ *catalyses the conversion of* B *from inactive* $B^{\text{off}}$ *to active* $B^{\text{on}}$*, using one or more ancillary fuels* $\{F_i\}$ *and producing waste* $\{W_i\}$*. The overall reaction is a direct bimolecular catalytic activation reaction if and only if:*

1. *The reaction is initialised with the interaction of* A *in state* $A^{\text{on}}$ *and* B *in state* $B^{\text{off}}$*.*
2. A *and* B *molecules have molecular cores that are retained in the products, rather than the input molecules being consumed and distinct outputs released.*

*Deactivation reactions have an equivalent form, but convert* B *from* $B^{\text{on}}$ *to* $B^{\text{off}}$*. If the same overall reaction stoichiometry is implemented by any set of reactions and species that violate conditions 1 and 2, we label the process a* pseudocatalytic bimolecular activation reaction.

Direct bimolecular catalytic (de)activation reactions have some important functional properties. The first is that, if the first step of the reaction requires the presence of A and B, nothing can happen unless both molecules are present. In pseudocatalytic implementations, as we discuss below, it is possible to produce activated $B^{\text{on}}$ or sequester A even if no B molecules in state $B^{\text{off}}$ are present, violating the logic of activation-based networks. The second is that the persistence of a molecular core of both the substrate and the catalyst allows either or both to be localised on a surface or scaffold, as is observed for some kinase cascades in living cells [14, 41, 50] and is often proposed for DNA-based systems [6, 7, 37, 39, 48].

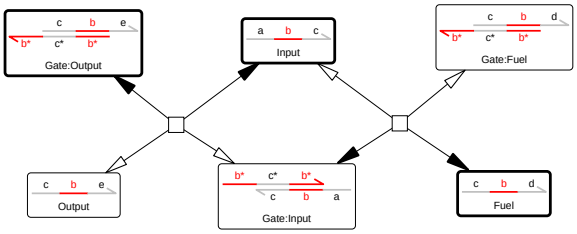


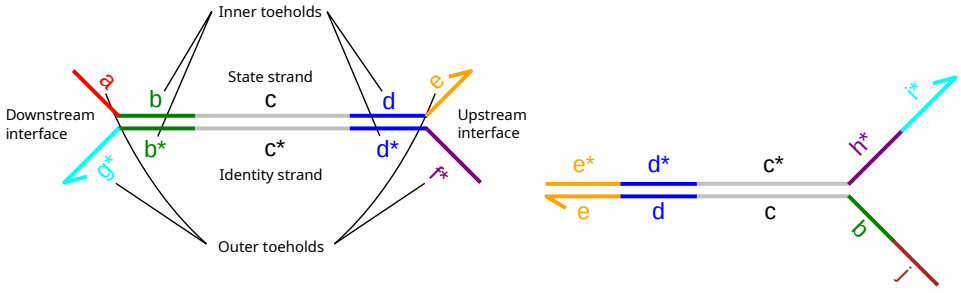

**Figure 2** Catalytic reaction using a seesaw gate [19, 36]. Reactants are shown in bold boxes; the input acts pseudocatalytically to "convert" the fuel into an output, with ancillary gate complexes consumed and produced. Each compound reaction is illustrated by a small square, and consists of sequential bind, displace, and unbind reactions. All reactions are reversible; open arrows indicate reactions proceeding forwards, and closed arrows by reactions proceeding backwards.

A number of DNA computing frameworks have been developed to implement reactions of the stoichiometry of Definition 1. The simplest, illustrated in Figure 2 (a), involves a two-step seesaw gate [19, 36]. An input molecule (representing A in state $A^{\text{on}}$ in Definition 1) binds to a gate-output complex ($F$), releasing the output (B in state $B^{\text{on}}$). The input is then displaced by a molecule conventionally described as the fuel, but fulfilling the role of B in state $B^{\text{off}}$ from Definition 1 in the context of catalysis, recovering A in state $A^{\text{on}}$ and generating a waste duplex ($W$). Although the A molecule recovered at the end of the process is the same one that initiated the process, the strands representing $B^{\text{off}}$ and $B^{\text{on}}$ molecules are distinct and the reaction is not initiated by the binding of A and B; it is therefore pseudocatalytic.

This pseudocatalysis can have important consequences. If a small quantity of input the strand representing $A^{\text{on}}$ is added to a solution containing the gate-output complex $F$ but no strand representing $B^{\text{off}}$, a large fraction of the $A^{\text{on}}$ strand will be sequestered and a corresponding amount of the $B^{\text{on}}$ strand produced. This sequestration of A and production of activated B from nothing violates the logic of ideal catalytic activation networks.

More complex strategies to implement reactions of the stoichiometry of Definition 1 using DSD exist [8, 34]. These approaches rely on the catalyst and substrate ($A^{\text{on}}$ and $B^{\text{off}}$ from Definition 1) interacting with a gate, rather than binding to each other, and the recovered catalyst and product are separate strands - the reactions are therefore pseudocatalytic. In certain limits, these strategies can approximate a mass-action dependence of reaction rates on the concentrations of $A^{\text{on}}$ and $B^{\text{off}}$ [8, 33], providing a better approximation to the logic of ideal catalytic activation circuits than the simple seesaw motif. The price, however, is the need to construct large multi-stranded gate complexes to facilitate the reaction; the complexity of these motifs is a major barrier to implementing such systems in autonomous setting such as living cells. Moreover, localising catalysts and substrates to a scaffold or surface remains challenging when the molecules themselves are not recovered.

We now consider how to design minimal DSD-based units that implement direct bimolecular catalytic (de)activation in catalytic activation networks. If the core of the substrate species B must be retained in both $B^{\text{off}}$ and $B^{\text{on}}$, $B^{\text{off}}$ and $B^{\text{on}}$ cannot simply be two strands with a slightly different sequence. Instead, $B^{\text{off}}$ and $B^{\text{on}}$ must either be distinct complexes of strands, in which at least one strand is common, have different secondary structure within a single strand, or both. To avoid complexities in balancing the thermodynamics of hairpin loop formation with bimolecular association, and suppressing the kinetics of unimolecular rearrangement, we do not pursue the possibility of engineering metastable secondary structure




**(a)** Major species. **(b)** Ancillary species.

**Figure 3** (a) Structure of major species in the ACDC system (substrates or catalysts), illustrating upstream and downstream interfaces, and inner and outer toeholds. The long central domain forms a stable binding duplex. (b) Structure of ancillary species (fuel, waste or substrate-catalyst complex).

within a strand. At least one of the states $B^{\text{off}}$ and $B^{\text{on}}$ of B must therefore consist of at least two strands. Moreover, since each activation state of each species must be a viable substrate in an arbitrary catalytic (de)activation network, the simplest approach that allows for a generic catalytic mechanism is to implement all substrate/catalyst species as two-stranded complexes.

## 3 ACDC: A Duplex-Based Catalytic DSD Framework

We introduce the Active Circuits of Duplex Catalysts (ACDC) scheme to implement catalytic activation networks through direct bimolecular catalytic (de)activation. Each reaction has three inputs: a substrate, a catalyst, and a single fuel complex. The outputs are a modified substrate, the recovered catalyst and a waste complex. The domain-level structures of these species are shown in Figure 3.

Substrates and catalysts – hereafter referred to as *major species* – are structurally identical. Each consists of two strands, each of which has one central long domain ($\sim$ 20 nucleotides (nt)) and two toeholds ($\sim$ 5 nt) on each side of the long domain. In major species, these strands are called the *identity strand* and the *state strand*. The identity strand is the preserved molecular core; the state strand specifies the activation state of a major species at a particular time (specifically, through the domain at its 5′ end - labelled “a” in Fig. 3).

The two strands in a major species are bound by three central domains; the *outer toeholds* at either end of the strands are *available* (unbound). Major species thus contain two *interfaces* at either end of the molecule, both displaying two available toeholds, one on each constituent strand. The *inner toeholds*, which are bound in major species, are described as *hidden*. We call the interface at the 5’ end of the state strand and the 3’ end of the identity strand the *downstream* interface and the interface with the 3’ end of the state strand and 5’ end of the identity strand the *upstream* interface.

All other two-stranded species in ACDC, including fuel and waste species, are described as *ancillary species*. They have a distinct structure from major species (Figure 3). Ancillary species also consist of two strands of five domains, but are bound by the central long domain and two shorter flanking toeholds (one outer toehold and one inner toehold) on one side. They therefore possess just one interface of available toeholds, but this interface presents two contiguous available toeholds on each strand.

The catalytic reaction of a single ACDC unit proceeds as shown in Figure 4. The downstream interface of the catalyst A in state $A^{\text{on}}$ and upstream interface of the substrate B in state $B^{\text{off}}$ bind together through recognition of all four available toeholds in the relevant interfaces. The resultant complex undergoes a 4-way branch migration, with the base pairs between the state and identity strand of the substrate and catalyst being exchanged for base pairs between the two state strands and the two identity strands. After the exchange of a hidden toehold and the central binding domain, the 4-stranded complex is held together by only two inner toeholds on either side of a 4-way junction. Dissociation by spontaneous detachment of these toeholds creates two ancillary product species, a waste $W_{AB\to B^{\text{on}}}$ and an intermediate complex $AB$. The sequence of these three reactions is called the *2r-4* reaction [21].

The fuel $F_{AB\to B^{\text{on}}}$ is identical to the waste, except for a single toehold. This toehold corresponds to the outer toehold of the state strand of B from the downstream interface. $F_{AB\to B^{\text{on}}}$ and $AB$ can undergo another *2r-4* reaction, producing B in state $B^{\text{on}}$ (equivalent to $B^{\text{off}}$, but with a single domain changed in the downstream interface) and recovering the catalyst. With the downstream interface of substrate B changed from that of $B^{\text{off}}$ into that of $B^{\text{on}}$, the substrate has been activated and could act as a catalyst to another reaction, provided that an appropriate downstream substrate and fuel were present. An equivalent catalytic process could trigger another reaction converting B from $B^{\text{on}}$ to $B^{\text{off}}$, *deactivating* B, analogous to dephosphorylation by a phosphatase. Note that the domain structures of ancillary species participating in an ACDC catalytic unit are unambiguously specified by the major species involved, since the individual strands are the same.

The ACDC mechanism borrows significantly from Qian and Winfree's design for surface-bound reaction networks [37]. In particular, that proposed framework also includes double-stranded species with identity and state strands, and exploits 4-way strand exchange reactions. However, the mechanistic details are more complex; species do not directly bind to each other, and interactions are mediated by multi-stranded gate complexes.

The basic ACDC unit in Figure 4 satisfies the conditions of Definition 1 for direct bimolecular catalytic activation, since the reaction is initiated by the binding of A in state $A^{\text{on}}$ and B in state $B^{\text{off}}$, and the identity strands in the major species are retained throughout. In this case, a single fuel molecule is consumed and a single waste produced by a single catalytic conversion. ACDC relies on the experimentally-verified mechanism of toehold-mediated 4-way branch migration [10, 22, 25, 49]. The number of base pairs and complexes is unchanged by each *2r-4* reaction, and therefore a bias for clockwise activation cycles (as opposed to anticlockwise deactivation) would require a large excess of fuel complexes $F_{AB\to B^{\text{on}}}$ relative to waste $W_{AB\to B^{\text{on}}}$. In addition, for a single catalytic cycle to operate as intended, the following assumptions must hold:

▶ **Assumption 2** (Stability of complexes). *It is assumed that strands bound together by long domains are stable and will not spontaneously dissociate. It is also assumed that if two strands are bound by a pair of complementary domains, any adjacent pairs of complementary domains that could bind to form a contiguous duplex are not available.*

▶ **Assumption 3** (Detachment of products). *It is assumed that 4-stranded complexes bound together by two pairs of toehold domains either side of a junction can dissociate into duplexes.*

▶ **Assumption 4** (Need for two complementary toeholds to trigger branch migration). *It is assumed that if a 4-stranded complex is formed by the binding of a single pair of toehold domains, it will dissociate into product duplexes, rather than undergo branch migration.*

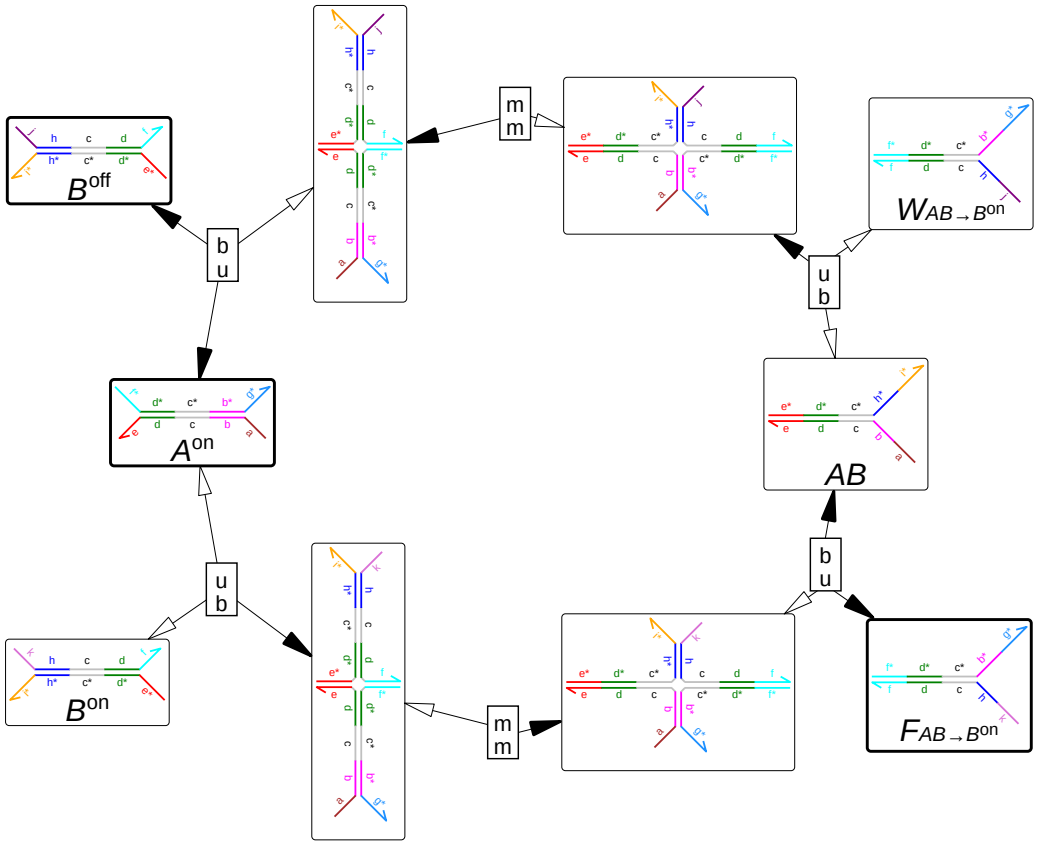



**Figure 4** A basic ACDC reaction unit $A^{\text{on}} + B^{\text{off}} + F_{AB \to B^{\text{on}}} \to A^{\text{on}} + B^{\text{on}} + W_{AB \to B^{\text{on}}}$, as represented by Visual DSD [23]. Inputs to the reaction are shown in bold, and each small box corresponding to a reaction step is labelled with b/u (bind/unbind) or m (migrate). Imbalances in the concentration of fuel and waste drive the reaction clockwise (the direction indicated by open arrows).

Assumption 2 ensures that the system keeps its duplex-based structure, and that toeholds are well hidden in complexes when required. Assumption 3 is necessary to avoid all species being sequestered into 4-stranded complexes. Note that the assumption is not that detachment must happen extremely quickly, since such 4-stranded complexes need to be metastable enough to initiate branch migration with reasonable frequency. It is equivalent to the need for single toeholds to detach in 3-way toehold exchange reactions [36]. In practice, toehold length and conditions such as temperature could be tuned to optimize the relative propensity for branch migration and detachment. Given a reasonable balance between branch migration and detachment, Assumption 4 – which enables the switching of B from $B^{\text{off}}$ and $B^{\text{on}}$ to have a downstream effect – is also likely to be satisfied.

## 4 Domain-based constraints in ACDC Networks

Larger catalytic activation networks can be constructed from the basic ACDC units of Figure 4, since the substrate B in its activated state $B^{\text{on}}$ can itself act as a catalyst. To describe these networks, let us now formalise the notation so that roman letters A, B, C etc. represent the nodes of the catalytic network, and italic symbols $A^{\text{on}}, B^{\text{off}}, F_{AB \to B^{\text{on}}}, AB$ etc.

represent the actual double-stranded molecular species in solution, carrying both identity and state information where appropriate. In this formalism, let A $\rightarrow$ B be a shorthand for the reaction $A^{\text{on}} + B^{\text{off}} + F_{AB \rightarrow B^{\text{on}}} \rightarrow A^{\text{on}} + B^{\text{on}} + W_{AB \rightarrow B^{\text{on}}}$ and C $\dashv$ B a shorthand for the reaction $C^{\text{on}} + B^{\text{on}} + F_{CB \rightarrow B^{\text{off}}} \rightarrow C^{\text{on}} + B^{\text{off}} + W_{CB \rightarrow B^{\text{off}}}$. Then, any potential catalytic activation network can be represented as a weighted directed graph, where nodes represent catalyst/substrate in the network and edges represent activation (edge weight 1) or deactivation (edge weight -1). Is it possible to realise any such graph using ACDC?

▶ **Assumption 5** (Toehold orthogonality). *We assume that there are sufficiently many toehold domain sequences that cross-talk between non-complementary domains is negligible.*

Since ACDC components share a long central domain, specificity is entirely driven through toehold recognition. As noted by Johnson, [21], there is a finite number of orthogonal short toehold domains that limits the size of the connected network that can be constructed. We assume that the network of interest does not violate this limit. We instead ask the realisability question at the level of domains.

▶ **Definition 6** (Realisability). *A catalytic activation network is realisable using the ACDC framework if a domain structure for the major species, which implies the domain structure of the ancillary species, can be specified such that:*

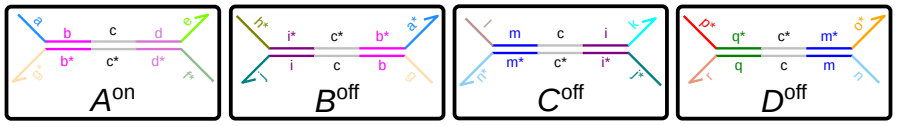
1. *All network edges* A $\rightarrow$ B (A $\dashv$ B) *are realised through basic ACDC units as illustrated in Figure 4. These units implement the overall reaction*

$$A^{\text{on}} + B^{\text{off}}(B^{\text{on}}) + F_{AB \rightarrow B^{\text{on}}(B^{\text{off}})} \rightarrow A^{\text{on}} + B^{\text{on}}(B^{\text{off}}) + W_{AB \rightarrow B^{\text{on}}(B^{\text{off}})},$$

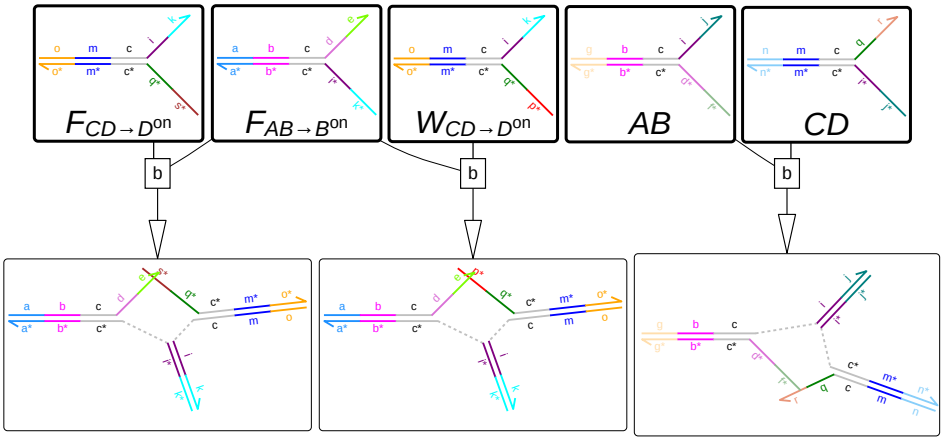
   *where the bracketed terms apply to deactivation reactions.*
2. *Other than the pairs of species that undergo reactions implied by condition 1, no pairs of species exist for which: (a) it is possible to exchange a pair of strands between the species and retain three contiguous bound domains in both the resultant complexes; and (b) the two species are able to bind via two available pairs of complementary toeholds. If (a) and (b) are both satisfied, the pair of species could undergo 2r-4 reactions as illustrated in Figure 4.*
3. *No two strands can form an uninterrupted duplex of four bound domains or more.*
4. *No two species (including all wastes, fuels and catalyst-substrate complexes) possess two available toehold pairs that could form a contiguous complementary duplex as shown in Figure 5(b).*

Condition 2 rules out reactions that respect the architecture of ACDC, but which involve reactants that are not intended to interact. Condition 3 rules out strand exchange reactions that allow an increase in the number of bound domains, which would sequester additional toeholds and violate the ACDC architecture (it is assumed that strand exchange reactions that would reduce the number of bound domains can be neglected). Condition 4 rules out the formation of 4-stranded complexes that can only dissociate by disrupting an uninterrupted two-toehold duplex. Contiguous duplexes of this kind are potentially stable, even if they cannot undergo strand exchange, and would potentially sequester components.

▶ **Lemma 7** (Realisability with activation implies realisability with deactivation). *If a catalytic activation network with purely activation reactions is realisable using the basic ACDC formalism, it is also realisable using the basic ACDC formalism if any subset of those reactions are converted to deactivation.*




**(a)** Major species.




**(b)** Ancillary species and unwanted reactions.

**Figure 5** Major species and a subset of ancillary species from an implementation of A → B → C → D using the ACDC formalism. Three unwanted reactions, as identified in Lemma 10, occur between the shown ancillary species.

**Proof.** A deactivation reaction is simply an activation reaction with the role of the fuel and waste reversed. Therefore a domain structure specification that realises a given network with activation reactions also realises all networks of the same structure. ◀

## 4.1 Realisability of Motifs in the ACDC formalism

Since there are infinitely many networks, we restrict our analysis to a set of motifs (generalised versions of the minimal examples depicted in Figure 6), establishing whether these motifs can be realised in isolation. The *split*, *integrate*, *cascade*, *self-activation*, *bidirectional edge*, *feedback loop* (FBL), and *feedforward loop* (FFL) are chosen because of their importance in biology and synthetic biology [2, 15, 16]. The proofs of theorems not explicitly given in this section are provided in Appendix B.

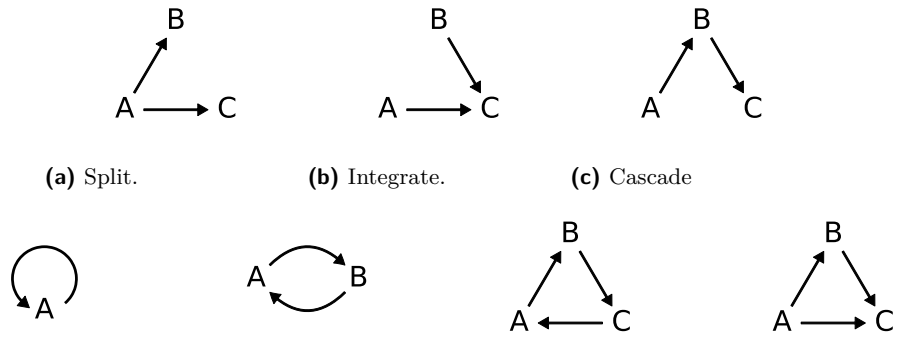
### 4.1.1 Motifs Without Loops

Theorems 8 and 9 establish that arbitrarily complex split and integrate motifs, constructed using ACDC in accordance with Definition 1, are realisable as per Definition 6.

▶ **Theorem 8** (Split motifs are realisable). *Consider the $N$ reactions*

$$\mathrm{A} \to \mathrm{B}_1 \quad \mathrm{A} \to \mathrm{B}_2 \quad \dots \quad \mathrm{A} \to \mathrm{B}_N,$$

*in which all $\mathrm{B}_i$ are distinct from* A*. This network is realisable for any $N \geq 1$.*




**(a)** Split. **(b)** Integrate. **(c)** Cascade

**(d)** Auto-activation loop. **(e)** Bidirectional edge. **(f)** Feedback loop. **(g)** Feedforward loop.

**Figure 6** Minimal example motifs of interest in a catalytic activation network.

▶ **Theorem 9** (Integrate motifs are realisable). *Consider the $N$ reactions*

$$\mathrm{A}_1 \to \mathrm{B} \quad \mathrm{A}_2 \to \mathrm{B} \quad \dots \quad \mathrm{A}_N \to \mathrm{B},$$

*in which all $\mathrm{A}_i$ are distinct from* B*. This network is realisable for any $N \geq 1$.*

Although all networks consist of simply combining split and integrate motifs for each node, proving that all split and integrate motifs are realisable in isolation does not prove that any network assembled from them is realisable. We therefore explore other simple motifs. For example, consider the *cascade* motif (a 3-component example is illustrated in Figure 6).

▶ **Lemma 10** (The ancillary species of a catalyst's upstream reactions and substrate's downstream reactions cause leak reactions). *Consider a reaction $\mathrm{B} \to \mathrm{C}$, and further assume that $\mathrm{A} \to \mathrm{B}$ and $\mathrm{C} \to \mathrm{D}$ for a species* A *and a species* D*. Then $AB$ and $CD$, and $F_{AB \to B^{\mathrm{on}}}$ and $F_{CD \to D^{\mathrm{on}}}/W_{CD \to D^{\mathrm{on}}}$ possess two available toehold pairs that could form a contiguous complementary duplex. No other violations of realisability occur.*

This failure is illustrated in Figure 5. The essence of the problem is that both the inner and outer toehold domains from the downstream end of $B^{\mathrm{on}}$ are available in $AB$ and $F_{AB \to B^{\mathrm{on}}}$, and the inner and outer toehold domains from the upstream end of $C$ are available in $CD$, $F_{CD \to D^{\mathrm{on}}}$ and $W_{CD \to D^{\mathrm{on}}}$. Since the downstream end of $B^{\mathrm{on}}$ is complementary to the upstream end of $C^{\mathrm{off}}$, the result is that the species can bind to each other strongly.

▶ **Theorem 11** (Cascades with at most 3 components are realisable; longer cascades are not realisable). *Consider the set of $N$ reactions $\mathrm{A}_1 \to \mathrm{A}_2$, $\mathrm{A}_2 \to \mathrm{A}_3 \dots \mathrm{A}_{N-1} \to \mathrm{A}_N$, in which all $\mathrm{A}_i$ are distinct. This network is realisable if and only if $N \leq 3$.*

**Proof.** A direct consequence of Lemma 10 and Definition 6. ◀

▶ **Theorem 12** (Long cascades are non-realisable due to a particular type of leak reaction only). *Consider the set of reactions $\mathrm{A}_1 \to \mathrm{A}_2$, $\mathrm{A}_2 \to \mathrm{A}_3 \dots \mathrm{A}_{N-1} \to \mathrm{A}_N$ for $N > 3$, in which all $\mathrm{A}_i$ are distinct. This network would be realisable if reactions between ancillary species $A_i A_{i+1}$ and $A_{i+2} A_{i+3}$, and $F_{A_i A_{i+1} \to A^{\mathrm{on}}_{i+1}}$ and $F_{A_{i+2} A_{i+3} \to A^{\mathrm{on}}_{i+3}}/W_{A_{i+2} A_{i+3} \to A^{\mathrm{on}}_{i+3}}$, were absent.*

The result of Theorem 11 is discouraging, since cascades are a major feature of kinase networks [20, 29]. Nonetheless, we will continue the analysis of remaining motifs, and present a potential solution in Section 5.

### 4.1.2 Motifs With Loops

A network possesses a loop if it is possible to traverse a path that begins and ends at the same node without using the same edge twice. For the purposes of this classification, a given (directed) edge can be traversed in either direction. Loops are common components of natural networks, providing the possibility of oscillation, bistability and filtering [2, 11].

▶ **Theorem 13** (Loops of odd length are not realisable). *Consider a system of reactions* $\mathrm{A}_1 \leftrightarrow \mathrm{A}_2 \leftrightarrow \mathrm{A}_3 \ldots \mathrm{A}_{N-1} \leftrightarrow \mathrm{A}_1$*, where* $\leftrightarrow$ *indicates a catalytic activation in either direction. This network, a* directionless loop*, is not realizable if* $N$ *is odd, unless the long central domain is self-complementary.*

**Proof.** ACDC circuits require that the long central domain alternates between a sequence and its complement in the identity strands of catalysts and their substrates. If $N$ is odd, then the sequence must be self-complementary for this alternation to happen. ◀

Introducing a self-complementary central domain is a strategy that risks a competition between duplexes and single-stranded hairpins. We do not consider it further.

▶ **Theorem 14** (Self interactions and bidirectional edges are not realisable). *Consider a system of reactions* $\mathrm{A}_1 \rightarrow \mathrm{A}_2 \rightarrow \mathrm{A}_3 \ldots \mathrm{A}_{N-1} \rightarrow \mathrm{A}_1$*. This network is not realisable if* $N \leq 2$*.*

The ACDC system is not inherently suited to auto-activation or bidirectional interactions. These motifs require complementarity between both the downstream and upstream toeholds of either a single species, or two species. Strands in the system therefore violate condition 3 of Definition 6 and will tend to hybridise to form fully complementary duplexes.

An isolated feedback loop is a network of size $N$ with a single directed path around the network. A simple example of length 3 is shown in Fig. 6(f).

▶ **Theorem 15** (Feedback loops are not realisable). *Consider the feedback loop* $\mathrm{A}_1 \rightarrow \mathrm{A}_2 \rightarrow \mathrm{A}_3 \ldots \mathrm{A}_{N-1} \rightarrow \mathrm{A}_1$*. Such a system is not realisable for any* $N$*.*

**Proof.** A direct consequence of Theorems 11, 13, and 14. ◀

As a consequence of Theorems 13 and 14, any realisable feedback loop must have $N \geq 4$. However, a feedback loop of this length faces the same issues as a cascade: formation of stable, undesired products between ancillary species. As with cascades, the problem is essentially local, due to interactions between ancillary species in reaction $n$ and reaction $n + 2$.

▶ **Theorem 16** (Long feedback loops with an even number of units are non-realisable due to a particular type of leak reaction only). *Consider the feedback loop*

$$\mathrm{A}_1 \rightarrow \mathrm{A}_2 \quad \mathrm{A}_2 \rightarrow \mathrm{A}_3 \quad \ldots \quad \mathrm{A}_{N-1} \rightarrow \mathrm{A}_N \quad \mathrm{A}_N \rightarrow \mathrm{A}_1$$

*For* $N$ *even,* $N \geq 4$*, this network would be realisable if reactions between ancillary species* $A_iA_{i+1}$ *and* $A_{i+2}A_{i+3}$*, and* $F_{A_iA_{i+1}\rightarrow A_{i+1}^{\mathrm{on}}}$ *and* $F_{A_{i+2}A_{i+3}\rightarrow A_{i+3}^{\mathrm{on}}}/W_{A_{i+2}A_{i+3}\rightarrow A_{i+3}^{\mathrm{on}}}$*, were absent. Here, the index* $j$ *in* $\mathrm{A}_j$ *should be interpreted modularly:* $\mathrm{A}_j = \mathrm{A}_{j-N}$ *for* $j > N$*.*

An isolated feedforward loop is a network of size $N$ with two directed paths from one node $i$ to another node $j$. Every other node appears exactly once in one of these paths. An example with path lengths of 1 and 2 is shown in Figure 6.

▶ **Theorem 17** (The relative lengths of paths are constrained in feedforward loops). *Consider the generalised feedforward loop*

$$\begin{array}{llll} \mathrm{A} \to \mathrm{B}_1 & \mathrm{B}_1 \to \mathrm{B}_2 & \dots & \mathrm{B}_{N-1} \to \mathrm{B}_N \quad \mathrm{B}_N \to \mathrm{D} \\ \mathrm{A} \to \mathrm{C}_1 & \mathrm{C}_1 \to \mathrm{C}_2 & \dots & \mathrm{C}_{M-1} \to \mathrm{C}_M \quad \mathrm{C}_M \to \mathrm{D} \end{array}$$

*For such a network to be realisable, it is necessary that $N \geq 1$, $M \geq 1$, and $N - M$ is even.*

**Proof.** The claim about $N - M$ having to be even follows from Theorem 13.

Assume for contradiction that a FFL with $N = 0$ and $M \geq 2$ and even is realisable. Since A activates $\mathrm{C}_1$, and both A and $\mathrm{C}_M$ activate D, it must be that $\mathrm{C}_M$ can also perform a branch migration with $\mathrm{C}_1$, which is an unwanted reaction violating condition 2 of Definition 6. ◀

Since each path in a feedforward loop is a cascade, Theorems 11 and 17 imply that only feedforward loops with a single intermediate in each branch are realisable.

▶ **Theorem 18** (Realisability of feedforward loops). *Consider the generalised feedforward loop*

$$\begin{array}{llll} \mathrm{A} \to \mathrm{B}_1 & \mathrm{B}_1 \to \mathrm{B}_2 & \dots & \mathrm{B}_{N-1} \to \mathrm{B}_N \quad \mathrm{B}_N \to \mathrm{D} \\ \mathrm{A} \to \mathrm{C}_1 & \mathrm{C}_1 \to \mathrm{C}_2 & \dots & \mathrm{C}_{M-1} \to \mathrm{C}_M \quad \mathrm{C}_M \to \mathrm{D} \end{array}$$

*Such a system is realisable if and only if $N = 1$ and $M = 1$.*

**Proof.** As a consequence of Theorems 8, 9, 11, and 17, all other FFLs are not realisable. The realisability of the FFL with N=1 and M=1 can be verified by inspection. ◀

Typically, feedforward loops use branches of different lengths to achieve a complex response to a signal over time [2,11]. Such networks are not realisable. Indeed, our analysis of various motifs has revealed that the majority are not realisable. Broadly speaking, there are a number of small motifs (e.g. auto-activation, bi-directional reactions, feedforward loops with no intermediates in one branch) that cannot be achieved because the major species themselves interact directly. In addition, loops of odd total length are not realisable due to the nature of complementary base pairs. However, most motifs are ruled out because of a single type of interaction, between the ancillary species in one reaction and the ancillary species in another reaction that occurs two steps downstream. In Section 5, we propose a strategy to overcome this last problem, massively increasing the scope of the ACDC framework.

## 5 Overcoming the Cascade Leak Reaction and Introducing Hidden Thermodynamic Drive

The most severe limitation of the ACDC system detailed in Section 3 is expressed by Theorem 11. Long cascades, and loops incorporating cascades, are non-realisable due to interactions between ancillary species of a given reaction, and ancillary species of a reaction separated by two catalytic steps (Theorem 12).

▶ **Assumption 19** (Mismatches destabilise complexes held together by two contiguous toehold domains). *We assume that a single mismatched C-C or G-G base pair, positioned adjacent to the interface of two toehold domains, is sufficiently destabilizing that an unwanted complex formed only by the binding of these toehold domains no longer precludes realisability.*

The basic design of the ACDC motif assumes that toehold binding is relatively weak; two toehold domains on either side of a junction must be able to dissociate by Assumption 2. Individual C-C or G-G mismatches are known to be highly destabilising [40], and should similarly allow for two contiguous domains to detach. Given Assumption 19, the challenge is then to systematically introduce mismatches so that all interactions between ancillary species identified in Theorem 12 are compromised by a mismatch, without compromising intended circuit activity. Our full scheme is visualised in Figure 7.

▶ **Definition 20** (Mismatches proposed to destabilize unintended complexes)**.** *We propose the following mismatches.*

1. *We propose that the upstream interface of every major species is made distinct for active and inactive states. Specifically, we introduce a G base at the inner edge of the outer toehold domain of the state strand of the inactive species, and a C base in the same position for the active species. Catalysts that (de)activate that species possess a C(G) in the complementary position of their downstream interface.*
2. *We introduce a C-C mismatch at the outer edge of the inner toehold domain at the downstream interface of each major species. This mismatch is eliminated in the formation of waste complexes, and retained in the substrate-catalyst complexes.*

▶ **Assumption 21** (Mismatches cannot cause leak reactions)**.** *We assume that the sequence constraints introduced by mismatch inclusion do not violate Assumption 5, and that the destabilisation of duplexes does not violate Assumption 2.*

In practice, mismatches will likely result in some increase in the rate of interactions between otherwise hidden toeholds; we assume that these rates remain negligible.

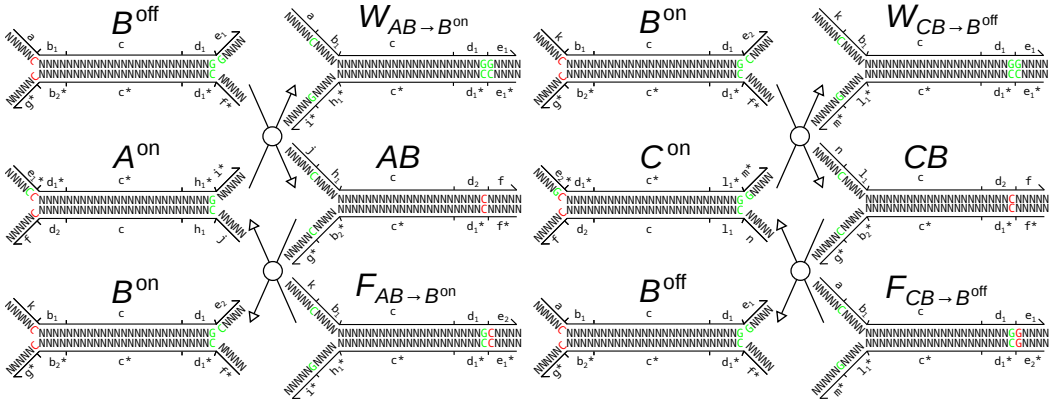
▶ **Theorem 22** (Mismatches successfully destabilize unintended complexes)**.** *The scheme proposed in Definition 20 satisfies the following:*

1. *All motifs that are realisable in the mismatch-free ACDC design remain realisable in the mismatch-based scheme.*
2. *Cascades of arbitrary length $N$ with at most the first and last reactions deactivating are realisable;*
3. *Feedback loops with $N$ even and $N \geq 6$ in which all reactions are activating are realisable;*
4. *Feedforward loops with $N \geq 1$, $M \geq 1$, $N - M$ even, in which at most the first and last reactions are deactivating in each branch, are realisable.*

The proof of Theorem 22 is given in Appendix B.

Note that the introduction of mismatches proposed in Definition 20 invalidates Lemma 7, since the downstream domains of activating and deactivating catalysts are now distinct. Indeed, the described strategy only eliminates unwanted sequestration in cascades in which the intermediate steps are activating. Nonetheless, it makes complex networks in which - for example - deactivating catalysts are always active realisable. Networks of this kind are common in biology [20, 29].

The first type of mismatch in Definition 20 ensures that there is always a C-C mismatch between the upstream toeholds of the state strand of $A^{\mathrm{on}}_{i+2}$ and the downstream toeholds of the state strand of $A^{\mathrm{on}}_{i+1}$ in the cascade $\mathrm{A}_i \to / \dashv \mathrm{A}_{i+1} \to \mathrm{A}_{i+2} \to / \dashv \mathrm{A}_{i+3}$, weakening the unwanted binding between the fuel and waste species identified in Theorem 12. Here $\to / \dashv$ indicates activation or deactivation. The second type of mismatch in Definition 20 ensures that the upstream toeholds of the identity strand of $\mathrm{A}_{i+2}$ are no longer fully complementary to the downstream toeholds of $\mathrm{A}_{i+1}$ in the cascade $\mathrm{A}_i \to / \dashv \mathrm{A}_{i+1} \to / \dashv \mathrm{A}_{i+2} \to / \dashv \mathrm{A}_{i+3}$, weakening the unwanted binding between ancillary species $A_i A_{i+1}$ and $A_{i+2} A_{i+3}$.




**(a)** A → B. **(b)** C ⊣ B.

**Figure 7** Illustration of the proposed mismatch schemes for reactions A → B and C ⊣ B, assuming toeholds of length 5 nucleotides and central domains of length 17 nucleotides. Specific mismatched bases are highlighted in red, and the same bases are highlighted in green when not part of a mismatch. The domains are separated with ticks on each species, and upstream interfaces of the major species are shown on the right of each diagram.

Having proposed these mismatches, it is important to determine that they would not compromise the intended reactions. The first type of mismatch in Definition 20 is not present in any complex that must form during the operation of the network; only in the initially-prepared fuel and if a (de)activating catalyst binds to an (in)active substrate. It therefore presents no issues for intended reactions.

The second type of mismatch in Definition 20 is more subtle. When a catalyst $A^{\text{on}}$ interacts with its substrate $B^{\text{off}}$, a mismatch at the very end of the catalyst duplex is converted into a mismatch within the stem of of the catalyst-substrate complex $AB$. Since mismatches are known to be more destabilizing in duplex interiors [32, 40], this conversion represents a local barrier to branch migration. The thermodynamic favourability of the full *2r-4* reaction $A^{\text{on}} + B^{\text{off}} \to AB + W_{AB \to B^{\text{on}}}$ (or the equivalent step in a deactivation reaction) is marginal, as the mismatch at the downstream end of $B^{\text{off}}$ counters this barrier. We assume that the local barriers introduced would not prohibit the intended reactions - indeed, conventional 3-way strand displacement is able to proceed through unmitigated C-C mismatch formation, albeit with a significant effect on kinetics [26]. In this case, any penalty is likely to be far weaker.

The second step of the catalytic turnover, $AB + F_{AB \to B^{\text{on}}} \to A^{\text{on}} + B^{\text{on}}$ (or the equivalent in a deactivation) is thermodynamically favourable (two internal mismatches are converted into exterior mismatches) and without local barriers, although one of the toeholds is effectively shortened to 4 base pairs. The overall catalytic (de)activation cycle effectively eliminates a single C-C (G-G) mismatch initially present in the fuel. The reaction as a whole is therefore driven forwards by the free energy of base-pairing via "hidden thermodynamic driving" [19]; products are more stable than reactants without consumption of initially available toeholds. In this sense, the mismatches proposed in Definition 20 will improve the efficacy of the ACDC motif, as the concentration excess of fuel relative to waste required to drive the reaction in the desired direction would be reduced.

## 6 A Compiler for ACDC Networks

To construct an ACDC network that implements a given graph, three things need to be done: (1) verification that the network is realisable; (2) enumerating all domains on all species given the graph topology; and (3) compile sequences for each domain and thus for each strand present in the system. We have created an ACDC compiler with this functionality [24]. While compilers for DSD systems that could be potentially be extended to accommodate our framework exist [3, 46], we decided to make our own since our framework has unique requirements about verifying the feasibility of a given CRN and introducing mismatches within domains.

The first part is done, at least at the level of each cascade and loop present, by analysing the properties of a given graph. For every pair of nodes $i, j$, all directed simple paths are computed. We search for paths of length $N \geq 3$ that containing edge weights of -1 anywhere other than at the first or last edge; these cascades are not rendered realisable by our mismatch scheme, per Theorem 22. Moreover, if there exists more than 1 path between the nodes, then either a FFL (at least two paths from $i$ to $j$ or from $j$ to $i$) or a FBL (at least one path from $i$ to $j$ and from $j$ to $i$) exists in the graph. Furthermore, if there exists more than 1 path between the nodes after transforming the graph to an undirected form, there exists a "directionless loop" (Theorem 13) in the graph. The realisability of the loop(s) can be verified from the lengths of the paths according to Theorems 13 and 22.

If a given graph is found to be realisable, then domains are assigned for each strand of each species, such that all complementarities and mismatches required by the topology are satisfied. This ask can be achieved by local analysis of the network topology.

Finally, a NUPACK [54] script is generated to generate optimal sequences for each strand. The required mismatches are hard-coded into the domain definitions in the script. The software is available at `https://zenodo.org/record/3948343`.

## 7 Discussion

We have introduced the ACDC scheme for constructing DNA-based networks that perform direct catalysis, analysed its shortcomings, and subsequently proposed practical improvements. As of now, we have focused only on the realisability of ACDC implementations for some graphs, not their dynamical behaviour. Three natural directions for further theoretical investigation are: (1) proving the realisability of arbitrary networks; (2) implementing additional hidden thermodynamic driving so that both *2r-4* substeps of a catalytic reaction are thermodynamically downhill; and (3) automated design of ACDC networks to perform some desired transfer function between input concentrations $x_i(t),\ i = 1..N$ and output concentrations $y_j(t),\ j = 1..M$. With regard to the first, we conjecture that all violations of realisability in arbitrary networks are attributable to the causes identified in Section 4.

Equally important, however, is experimentally testing the ACDC motif. Whilst 4-way branch migration has been used in several contexts [10, 22, 25, 49], the toehold exchange mechanism proposed here is relatively untested. It is also important to establish that the mismatches function as intended, limiting sequestration reactions and providing strong overall thermodynamic driving without causing excessive local barriers that frustrate the necessary reactions. A final consideration is the possibility of leak reactions involving non-complementary toeholds that we have assumed to be negligible. It remains to be established that unintended reactions will occur at a negligible rate, particularly in the context of species containing mismatches. This research is ongoing within the group.

A key property of ACDC is the two recognition interfaces within each species and the inherent symmetry in the species that follows. While this symmetry is a design feature that allows both substrate-like and catalyst-like behaviour for a single species, it also has a drawback that domains that are essential for some reaction to occur are also present in reactions where they only act as identity placeholders (downstream interface of a catalyst and an upstream interface of a substrate) that do not interact with any other domain. Consider the reaction in Figure 4; the identity of the "placeholder domains" $a, b, g, h, i, j, k$ that aren't involved in the initial binding and migration reactions could be swapped to arbitrary domains that aren't complementary with $d, e, f$ or each other in only one species and the reaction could still occur (assuming the correct fuel species is generated based on the substrate and catalyst). However, this may not be possible if A and B are part of some larger computational network where the placeholder domain identities are important. Another drawback of the symmetry is the limitation of loop lengths to even numbers, characterised in Theorem 13. An obvious potential mitigation to this problem is to make the central domain its own complement, although this choice risks the formation of self-complementary hairpins.

The weaknesses of the ACDC motif invite the exploration of other possible designs of catalytic activation networks that operate via direct bimolecular catalysis. It is an open question as to whether the shortcomings of ACDC can be mitigated without a substantial increase in complexity or abandoning the mechanism of direct catalytic action.

## 8 Conclusion

We have established the concept of a direct catalytic reaction and discussed why previous work on catalytic DNA computing does not fulfil this definition. We have then proposed a framework, ACDC, for implementing non-equilibrium catalytic (de)activation networks using direct catalytic activation, analogous to systems seen in living cells. ACDC is simple in the sense that all species contain only two strands - an important consideration in the context of implementing DSD circuitry in a broad range of contexts.

We have analysed the framework's expressiveness by exploring the implementation of seven network motifs with ACDC. The basic design is highly limited by the inherent symmetry of components, prohibiting long cascades and most feedforward and feedback loops. However, we propose that systematic placement of mismatches can obviate these difficulties in many contexts. Moreover, we argue that these initially-present mismatches can contribute a "hidden thermodynamic driving" [19] to the ACDC motifs, increasing the robustness of the design to subtleties in DNA thermodynamics and reducing the concentration imbalances of fuels required to drive the reactions forward. We present a compiler for the sequence design of ACDC-based networks that implements these findings [24].

## References


1 Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, November 1994. doi:10.1126/science.7973651.

2 Uri Alon. *An introduction to systems biology: design principles of biological circuits, Second Edition*. CRC Press LLC, Boca Raton, UNITED STATES, 2019.

3 Stefan Badelt, Seung Woo Shin, Robert F. Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. A general-purpose CRN-to-DSD compiler with formal verification, optimization, and simulation capabilities. In *DNA computing and molecular programming*, Lecture notes in computer science, pages 232–248. Springer, Cham, September 2017. doi:10.1007/978-3-319-66799-7_15.

**4** David Barford, Amit K. Das, and Marie-Pierre Egloff. The structure and mechanism of protein phosphatases: insights into catalysis and regulation. *Annual Review of Biophysics and Biomolecular Structure*, 27(1):133–164, June 1998. Publisher: Annual Reviews. doi:10.1146/annurev.biophys.27.1.133.

**5** John P. Barton and Eduardo D. Sontag. The energy costs of insulators in biochemical networks. *Biophysical Journal*, 104(6):1380–1390, March 2013. doi:10.1016/j.bpj.2013.01.056.

**6** Hieu Bui, Shalin Shah, Reem Mokhtar, Tianqi Song, Sudhanshu Garg, and John Reif. Localized DNA hybridization chain reactions on DNA origami. *ACS Nano*, 12(2):1146–1155, February 2018. Publisher: American Chemical Society. doi:10.1021/acsnano.7b06699.

**7** Gourab Chatterjee, Neil Dalchau, Richard A. Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature Nanotechnology*, 12(9):920–927, September 2017. Number: 9 Publisher: Nature Publishing Group. doi:10.1038/nnano.2017.127.

**8** Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature Nanotechnology*, 8(10):755–762, October 2013. doi:10.1038/nnano.2013.189.

**9** Kevin M. Cherry and Lulu Qian. Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. *Nature*, 559(7714):370–376, July 2018. doi:10.1038/s41586-018-0289-6.

**10** Nadine L. Dabby. *Synthetic molecular machines for active self-assembly: prototype algorithms, designs, and experimental study*. PhD thesis, California Institute of Technology, Pasadena, California, 2013. URL: https://pdfs.semanticscholar.org/e668/440cdb786ea7c2d0d6ae306c5aefef1208f6.pdf.

**11** Wiet de Ronde and Pieter Rein ten Wolde. Multiplexing oscillatory biochemical signals. *Physical Biology*, 11(2):026004, April 2014. doi:10.1088/1478-3975/11/2/026004.

**12** Abhishek Deshpande and Thomas E. Ouldridge. High rates of fuel consumption are not required by insulating motifs to suppress retroactivity in biochemical circuits. *Engineering Biology*, 1(2):86–99, December 2017. Publisher: IET Digital Library. doi:10.1049/enb.2017.0017.

**13** Robert M. Dirks, Justin S. Bois, Joseph M. Schaeffer, Erik Winfree, and Niles A. Pierce. Thermodynamic analysis of interacting nucleic acid strands. *SIAM Review*, 49(1):65–88, January 2007. Publisher: Society for Industrial and Applied Mathematics. doi:10.1137/060651100.

**14** Elaine A. Elion. Ste5: a meeting place for MAP kinases and their associates. *Trends in Cell Biology*, 5(8):322–327, August 1995. doi:10.1016/S0962-8924(00)89055-8.

**15** Michael B. Elowitz and Stanislas Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335–338, January 2000. Number: 6767 Publisher: Nature Publishing Group. doi:10.1038/35002125.

**16** Timothy S. Gardner, Charles R. Cantor, and James J. Collins. Construction of a genetic toggle switch in Escherichia coli. *Nature*, 403(6767):339–342, January 2000. Number: 6767 Publisher: Nature Publishing Group. doi:10.1038/35002131.

**17** Anthony J. Genot, Teruo Fujii, and Yannick Rondelez. Scaling down DNA circuits with competitive neural networks. *Journal of The Royal Society Interface*, 10(85):20130212, August 2013. Publisher: Royal Society. doi:10.1098/rsif.2013.0212.

**18** Christopher C. Govern and Pieter Rein ten Wolde. Energy dissipation and noise correlations in biochemical sensing. *Physical Review Letters*, 113(25):258102, December 2014. Publisher: American Physical Society. doi:10.1103/PhysRevLett.113.258102.

**19** Natalie E. C. Haley, Thomas E. Ouldridge, Ismael Mullor Ruiz, Alessandro Geraldini, Ard A. Louis, Jonathan Bath, and Andrew J. Turberfield. Design of hidden thermodynamic driving for non-equilibrium systems via mismatch elimination during DNA strand displacement. *Nature Communications*, 11(1):2562, May 2020. Number: 1 Publisher: Nature Publishing Group. doi:10.1038/s41467-020-16353-y.

20 Ira Herskowitz. MAP kinase pathways in yeast: for mating and more. *Cell*, 80(2):187–197, January 1995. doi:10.1016/0092-8674(95)90402-6.

21 Robert F. Johnson. Impossibility of sufficiently simple chemical reaction network implementations in DNA strand displacement. In Ian McQuillan and Shinnosuke Seki, editors, *Unconventional computation and natural computation*, Lecture notes in computer science, pages 136–149. Springer International Publishing, 2019. doi:10.1007/978-3-030-19311-9_12.

22 Shohei Kotani and William L. Hughes. Multi-arm junctions for dynamic DNA nanotechnology. *Journal of the American Chemical Society*, 139(18):6363–6368, May 2017. doi:10.1021/jacs.7b00530.

23 Matthew R. Lakin, Simon Youssef, Filippo Polo, Stephen Emmott, and Andrew Phillips. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3213, November 2011. doi:10.1093/bioinformatics/btr543.

24 Antti Lankinen. ACDC compiler, July 2020. URL: https://zenodo.org/record/3948343.

25 Tong Lin, Jun Yan, Luvena L. Ong, Joanna Robaszewski, Hoang D. Lu, Yongli Mi, Peng Yin, and Bryan Wei. Hierarchical assembly of DNA nanostructures based on four-way toehold-mediated strand displacement. *Nano Letters*, 18(8):4791–4795, August 2018. Publisher: American Chemical Society. doi:10.1021/acs.nanolett.8b01355.

26 Robert R. F. Machinek, Thomas E. Ouldridge, Natalie E. C. Haley, Jonathan Bath, and Andrew J. Turberfield. Programmable energy landscapes for kinetic control of DNA strand displacement. *Nature Communications*, 5(1):1–9, November 2014. Number: 1 Publisher: Nature Publishing Group. doi:10.1038/ncomms6324.

27 Marcelo O. Magnasco. Chemical kinetics is Turing universal. *Physical Review Letters*, 78(6):1190–1193, February 1997. doi:10.1103/PhysRevLett.78.1190.

28 G. Manning, D. B. Whyte, R. Martinez, T. Hunter, and S. Sudarsanam. The protein kinase complement of the human genome. *Science*, 298(5600):1912–1934, December 2002. Publisher: American Association for the Advancement of Science Section: Review. doi:10.1126/science.1075762.

29 Christopher J. Marshall. MAP kinase kinase kinase, MAP kinase kinase and MAP kinase. *Current Opinion in Genetics & Development*, 4(1):82–89, February 1994. doi:10.1016/0959-437X(94)90095-7.

30 Pankaj Mehta, Alex H. Lang, and David J. Schwab. Landauer in the agea of synthetic biology: energy consumption and information processing in biochemical networks. *Journal of Statistical Physics*, 162(5):1153–1166, March 2016. doi:10.1007/s10955-015-1431-6.

31 Thomas E. Ouldridge, Christopher C. Govern, and Pieter Rein ten Wolde. Thermodynamics of computational copying in biochemical systems. *Physical Review X*, 7(2):021004, April 2017. Publisher: American Physical Society. doi:10.1103/PhysRevX.7.021004.

32 Thomas E. Ouldridge, Ard A. Louis, and Jonathan P. K. Doye. Structural, mechanical, and thermodynamic properties of a coarse-grained DNA model. *The Journal of Chemical Physics*, 134(8):085101, February 2011. Publisher: American Institute of Physics. doi:10.1063/1.3552946.

33 Tomislav Plesa. Stochastic approximation of high-molecular by bi-molecular reactions. *arXiv:1811.02766 [math, q-bio]*, November 2018. arXiv: 1811.02766. URL: http://arxiv.org/abs/1811.02766.

34 Lulu Qian, David Soloveichik, and Erik Winfree. Efficient Turing-universal computation with DNA polymers. In Yasubumi Sakakibara and Yongli Mi, editors, *DNA computing and molecular programming*, Lecture notes in computer science, pages 123–140, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-18305-8_12.

35 Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, June 2011. doi:10.1126/science.1200520.

**36** Lulu Qian and Erik Winfree. A simple DNA gate motif for synthesizing large-scale circuits. *Journal of the Royal Society Interface*, 8(62):1281–1297, September 2011. doi:10.1098/rsif.2010.0729.

**37** Lulu Qian and Erik Winfree. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In Satoshi Murata and Satoshi Kobayashi, editors, *DNA computing and molecular programming*, Lecture notes in computer science, pages 114–131, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-11295-4_8.

**38** Lulu Qian, Erik Winfree, and Jehoshua Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475(7356):368–372, July 2011. doi:10.1038/nature10262.

**39** Ismael Mullor Ruiz, Jean-Michel Arbona, Amitkumar Lad, Oscar Mendoza, Jean-Pierre Aimé, and Juan Elezgaray. Connecting localized DNA strand displacement reactions. *Nanoscale*, 7(30):12970–12978, July 2015. Publisher: The Royal Society of Chemistry. doi:10.1039/C5NR02434J.

**40** John SantaLucia and Donald Hicks. The thermodynamics of DNA structural motifs. *Annual Review of Biophysics and Biomolecular Structure*, 33(1):415–440, 2004. _eprint: https://doi.org/10.1146/annurev.biophys.32.110601.141800. doi:10.1146/annurev.biophys.32.110601.141800.

**41** Hans J. Schaeffer, Andrew D. Catling, Scott T. Eblen, Lara S. Collier, Anke Krauss, and Michael J. Weber. MP1: a MEK binding partner that enhances enzymatic activation of the MAP kinase cascade. *Science*, 281(5383):1668–1671, September 1998. Publisher: American Association for the Advancement of Science Section: Report. doi:10.1126/science.281.5383.1668.

**42** Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–1588, December 2006. Publisher: American Association for the Advancement of Science Section: Report. doi:10.1126/science.1132493.

**43** Nadrian C. Seeman and Hanadi F. Sleiman. DNA nanotechnology. *Nature Reviews Materials*, 3(1):1–23, November 2017. doi:10.1038/natrevmats.2017.68.

**44** David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, March 2010. doi:10.1073/pnas.0909380107.

**45** Carlo Spaccasassi, Matthew R. Lakin, and Andrew Phillips. A logic programming language for computational nucleic acid devices. *ACS synthetic biology*, 8(7):1530–1547, July 2019. doi:10.1021/acssynbio.8b00229.

**46** Niranjan Srinivas, James Parkin, Georg Seelig, Erik Winfree, and David Soloveichik. Enzyme-free nucleic acid dynamical systems. *Science*, 358(6369), December 2017. doi:10.1126/science.aal2052.

**47** J. David Sweatt. The neuronal MAP kinase cascade: a biochemical signal integration system subserving synaptic plasticity and memory. *Journal of Neurochemistry*, 76(1):1–10, 2001. doi:10.1046/j.1471-4159.2001.00054.x.

**48** Mario Teichmann, Enzo Kopperger, and Friedrich C. Simmel. Robustness of localized DNA strand displacement cascades. *ACS Nano*, 8(8):8487–8496, August 2014. Publisher: American Chemical Society. doi:10.1021/nn503073p.

**49** Suvir Venkataraman, Robert M. Dirks, Paul W. K. Rothemund, Erik Winfree, and Niles A. Pierce. An autonomous polymerization motor powered by DNA hybridization. *Nature Nanotechnology*, 2(8):490–494, August 2007. Number: 8 Publisher: Nature Publishing Group. doi:10.1038/nnano.2007.225.

**50** Alan J. Whitmarsh, Julie Cavanagh, Cathy Tournier, Jun Yasuda, and Roger J. Davis. A mammalian scaffold complex that selectively mediates MAP kinase activation. *Science*, 281(5383):1671–1674, September 1998. Publisher: American Association for the Advancement of Science Section: Report. doi:10.1126/science.281.5383.1671.

**51** Christian Widmann, Spencer Gibson, Matthew B. Jarpe, and Gary L. Johnson. Mitogen-activated protein kinase: conservation of a three-kinase module from yeast to human. *Physiolo-*

*gical Reviews*, 79(1):143–180, January 1999. Publisher: American Physiological Society. doi:10.1152/physrev.1999.79.1.143.

52 Wataru Yahiro and Masami Hagiya. Implementation of Turing machine using DNA strand displacement. In Carlos Martín-Vide, Takaaki Mizuki, and Miguel A. Vega-Rodríguez, editors, *Theory and Practice of Natural Computing*, Lecture notes in computer science, pages 161–172, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-49001-4_13.

53 Peng Yin, Harry M. T. Choi, Colby R. Calvert, and Niles A. Pierce. Programming biomolecular self-assembly pathways. *Nature*, 451(7176):318–322, January 2008. Number: 7176 Publisher: Nature Publishing Group. doi:10.1038/nature06451.

54 Joseph N. Zadeh, Conrad D. Steenberg, Justin S. Bois, Brian R. Wolfe, Marshall B. Pierce, Asif R. Khan, Robert M. Dirks, and Niles A. Pierce. NUPACK: Analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32(1):170–173, 2011. doi:10.1002/jcc.21596.

55 David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature Chemistry*, 3(2):103–113, February 2011. doi:10.1038/nchem.957.

56 David Yu Zhang, Andrew J. Turberfield, Bernard Yurke, and Erik Winfree. Engineering entropy-driven reactions and networks catalyzed by DNA. *Science*, 318(5853):1121–1125, November 2007. Publisher: American Association for the Advancement of Science Section: Report. doi:10.1126/science.1148532.

## A Notation For ACDC Species and Reactions

$[a\ b]$ denotes a strand consisting of domains $a$ and $b$. Logical not is denoted by $\neg$ and logical and by $\wedge$.

▶ **Definition 23.** *(ACDC major species structure). Each major species in an ACDC network consists of two strands, each of which have one long domain and four toehold domains. The two strands are called state strand and identity strand based on the fact that one strand decodes the state of the species and other the identity. A major species $X$ has the following domains (note the use of H for "inner" to avoid confusion with "identity"):*

- $SH5(X)$*: the inner toehold domain on the 5' side (downstream end) of the state strand.*
- $SO5(X)$*: the outer toehold domain on the 5' side (downstream end) of the state strand.*
- $SH3(X)$*: the inner toehold domain on the 3' side (upstream end) of the state strand.*
- $SO3(X)$*: the outer toehold domain on the 3' side (upstream end) of the state strand.*
- $IH5(X)$*: the inner toehold domain on the 5' side (upstream end) of the identity strand.*
- $IO5(X)$*: the outer toehold domain on the 5' side (upstream end) of the identity strand.*
- $IH3(X)$*: the inner toehold domain on the 3' side (downstream end) of the identity strand.*
- $IO3(X)$*: the outer toehold domain on the 3' side (downstream end) of the identity strand.*
- $SL(X)$*: the long domain on the state strand.*
- $IL(X)$*: the long domain on the identity strand.*

▶ **Definition 24.** *(Subset and logical operations for ACDC species). The following operations will be useful in the analysis of ACDC networks:*

- *Complementarity* $\diamond$ *:* $x \diamond y$ *is true for sequences* $x, y$ *iff* $x = y^*$ *(and* $x^* = y$*).*
- *Complementarity with mismatch* $\square$ *:* $x \square y$ *is true for sequences* $x, y$ *iff* $x = y^*$ *(and* $x^* = y$*) except for a single centrally-placed C-C or G-G mismatch.* $x \square y$ *is distinct from* $\neg x \diamond y$*, for which it is assumed that interactions between* $x$ *and* $y$ *are negligible.*
- $5'$ *(downstream end) state toehold sequence* $S5(X) := [SO5(X)\ SH5(X)]$.
- $3'$ *(upstream end) state toehold sequence* $S3(X) := [SH3(X)\ SO3(X)]$.
- $5'$ *(upstream end) identity toehold sequence* $I5(X) := [IO5(X)\ IH5(X)]$.
- $3'$ *(downstream end) identity toehold sequence* $I3(X) := [IH3(X)\ IO3(X)]$.

▶ **Definition 25.** *(Major species). A major species $X$ must satisfy*

$$\neg\big(SO5(X)\diamond IO3(X)\big)\wedge\big(SH5(X)\diamond IH3(X)\big)\wedge \\ \big(SL(X)\diamond IL(X)\big)\wedge \\ \big(SH3(X)\diamond IH5(X)\big)\wedge\neg\big(SO3(X)\diamond IO5(X)\big).$$

▶ **Definition 26.** *(Domain complementarities in an ACDC reaction without mismatches). An ACDC reaction* $\mathrm{A}\rightarrow\mathrm{B}$ *or* $\mathrm{A}\dashv\mathrm{B}$ *implies*

$$\begin{aligned} &S5(A^{\mathrm{on}})\diamond S3(B^{\mathrm{off}})=S3(B^{\mathrm{on}}) &&\wedge\\ &IL(A^{\mathrm{on}})=IL(A^{\mathrm{off}})\diamond IL(B^{\mathrm{off}})=IL(B^{\mathrm{on}}) &&\wedge\\ &I3(A^{\mathrm{on}})=I3(A^{\mathrm{off}})\diamond I5(B^{\mathrm{off}})=I5(B^{\mathrm{on}}). \end{aligned}$$

*Domains not constrained by these requirements are non-complementary. We emphasize that the domains of ancillary species involved in* $\mathrm{A}\rightarrow\mathrm{B}$ *are determined unambiguously by the domains of the relevant major species.*

▶ **Definition 27.** *(Domain complementarities in ACDC reactions with mismatches). An ACDC reaction* $\mathrm{A}\rightarrow/\dashv\mathrm{B}$ *with mismatches placed as per Definition 20 implies*

$$\begin{aligned} &S5(A^{\mathrm{on}})\diamond S3(B^{\mathrm{off}})\ /\ S5(A^{\mathrm{on}})\square S3(B^{\mathrm{off}}) &&\wedge\\ &S5(A^{\mathrm{on}})\square S3(B^{\mathrm{on}})\ /\ S5(A^{\mathrm{on}})\diamond S3(B^{\mathrm{on}}) &&\wedge\\ &IL(A^{\mathrm{on}})=IL(A^{\mathrm{off}})\diamond IL(B^{\mathrm{off}})=IL(B^{\mathrm{on}}) &&\wedge\\ &I3(A^{\mathrm{on}})=I3(A^{\mathrm{off}})\square I5(B^{\mathrm{off}})=I5(B^{\mathrm{on}}). \end{aligned}$$

*Domains not constrained by these requirements are non-complementary.*

## B Proofs of Theorems and Lemmas 8 - 22

▶ **Theorem 8** (Split motifs are realisable)**.** *Consider the $N$ reactions* $\mathrm{A}\rightarrow\mathrm{B}_1,\mathrm{A}\rightarrow\mathrm{B}_2,\dots\mathrm{A}\rightarrow\mathrm{B}_N$*, in which all* $\mathrm{B}_i$ *are distinct from* A*. Such a network is realisable for any* $N\geq 1$*.*

**Proof.** By induction. Assume that the split motif is realisable for a given $N=M>0$. If so, a valid domain level implementation exists for $N=M$. Now consider the species $B^{\mathrm{off}}_{M+1}, B^{\mathrm{on}}_{M+1}, AB_{M+1}W_{AB_{M+1}\rightarrow B^{\mathrm{on}}_{M+1}}, F_{AB_{M+1}\rightarrow B^{\mathrm{on}}_{M+1}}$ *related* to a putative additional node $\mathrm{B}_{M+1}$. Let these species be identical to those of $\mathrm{B}_1$, except with the domains that function as the downstream end $(SO5, SH5, IO3, IH3)$ in $B^{\mathrm{off}}_{M+1}, B^{\mathrm{on}}_{M+1}$ changed to have no complementarity with any domains in the existing valid implementation for $N=M$. This assignment is possible by Assumption 5. Since the upstream domains $SO3, SH3, IO5, IH5$ of $B^{\mathrm{on}}_{M+1}, B^{\mathrm{off}}_{M+1}$ are identical to those of $B^{\mathrm{on}}_1, B^{\mathrm{off}}_1$, Definition 26 implies $\mathrm{A}\rightarrow\mathrm{B}$ as required by condition 1 of Definition 6. Moreover, since the species related to node $\mathrm{B}_{M+1}$ are identical to those of the existing node $\mathrm{B}_1$, except for the downstream domains with no complementarity to the rest of the network, no new violations of conditions 2-4 of Definition 6 can occur due to interactions between the species related to $\mathrm{B}_{M+1}$ and $A^{\mathrm{on}}$ or those related to $\mathrm{B}_i$ for $1<i\leq M$. By considering the species defined in Figure 4 for $\mathrm{B}_1$, and replacing domains $h,i,j,k$ with $h_{M+1},i_{M+1},j_{M+1},k_{M+1}$ to define $\mathrm{B}_{M+1}$, it is straightforward to establish that no violations of conditions 2-4 of Definition 6 occur between the species related to $\mathrm{B}_{M+1}$ and $\mathrm{B}_1$. Therefore if a split motif of size $N=M$ is realisable, a split motif of size $N=M+1$ is realisable. Given the valid implementation for $N=1$ in Figure 4, split motifs of arbitrary $N>0$ are realisable. ◀

▶ **Theorem 9** (Integrate motifs are realisable). *Consider the $N$ reactions $\mathrm{A}_1 \to \mathrm{B}, \mathrm{A}_2 \to \mathrm{B}, \ldots \mathrm{A}_N \to \mathrm{B}$, in which all $\mathrm{A}_i$ are distinct from* B. *This network is realisable for any $N \geq 1$.*

**Proof.** The proof is identical to that of Theorem 8 with the direction of catalysis interchanged. ◀

▶ **Lemma 10** (The ancillary species of a catalyst's upstream reactions and substrate's downstream reactions cause leak reactions). *Consider a reaction* $\mathrm{B} \to \mathrm{C}$, *and further assume that* $\mathrm{A} \to \mathrm{B}$ *and* $\mathrm{C} \to \mathrm{D}$ *for a species* A *and a species* D. *Then $AB$ and $CD$, and $F_{AB \to B^{\mathrm{on}}}$ and $F_{CD \to D^{\mathrm{on}}}/W_{CD \to D^{\mathrm{on}}}$ possess two available toehold pairs that could form a contiguous complementary duplex. No other violations of realisability occur.*

**Proof.** Consider the following major species:

$$
\begin{aligned}
A^{\mathrm{on}} &:= [a \;\; b \;\; c \;\; d \;\; e\;] \\
&\quad\;\; [g^* \; b^* \; c^* \; d^* \; f^*] \\
B^{\mathrm{off}} &:= [h^* \; i^* \; c^* \; b^* \; a^*] \\
&\quad\;\; [j \;\; i \;\; c \;\; b \;\; g\;] \\
C^{\mathrm{off}} &:= [l \;\; m \;\; c \;\; i \;\; k\;] \\
&\quad\;\; [n^* \; m^* \; c^* \; i^* \; j^*] \\
D^{\mathrm{off}} &:= [p^* \; q^* \; c^* \; m^* \; o^*] \\
&\quad\;\; [r \;\; q \;\; c \;\; m \;\; n\;]
\end{aligned}
\qquad\qquad
\begin{aligned}
B^{\mathrm{on}} &:= [k^* \; i^* \; c^* \; b^* \; a^*] \\
&\quad\;\; [j \;\; i \;\; c \;\; b \;\; g\;] \\
C^{\mathrm{on}} &:= [o \;\; m \;\; c \;\; i \;\; k\;] \\
&\quad\;\; [n^* \; m^* \; c^* \; i^* \; j^*] \\
D^{\mathrm{on}} &:= [s^* \; q^* \; c^* \; m^* \; o^*] \\
&\quad\;\; [r \;\; q \;\; c \;\; m \;\; n\;]
\end{aligned}
$$

where the top (bottom) strand of each species is the state (identity) strand in 5'-3' (3'-5') direction. These species and the accordingly generated ancillary species implement the cascade $\mathrm{A} \to \mathrm{B} \to \mathrm{C} \to \mathrm{D}$. Conditions 1-3 of Definition 6 are satisfied.

To establish whether condition 4 of Definition 6 is necessarily violated, consider the unbound domains on the ancillary species in the system $\mathrm{A} \to \mathrm{B} \to \mathrm{C} \to \mathrm{D}$:

- $I5(A^{\mathrm{off}})$, $I3(B^{\mathrm{off}})$ in $AB$
- $S3(A^{\mathrm{on}})$, $S5(B^{\mathrm{on}})$ in $F_{AB \to B^{\mathrm{on}}}$
- $S3(A^{\mathrm{on}})$, $S5(B^{\mathrm{off}})$ in $W_{AB \to B^{\mathrm{on}}}$
- $I5(B^{\mathrm{off}})$, $I3(C^{\mathrm{off}})$ in $BC$
- $S3(B^{\mathrm{on}})$, $S5(C^{\mathrm{on}})$ in $F_{BC \to C^{\mathrm{on}}}$
- $S3(B^{\mathrm{on}})$, $S5(C^{\mathrm{off}})$ in $W_{BC \to C^{\mathrm{on}}}$
- $I5(C^{\mathrm{off}})$, $I3(D^{\mathrm{off}})$ in $CD$
- $S3(C^{\mathrm{on}})$, $S5(D^{\mathrm{on}})$ in $F_{CD \to D^{\mathrm{on}}}$
- $S3(C^{\mathrm{on}})$, $S5(D^{\mathrm{off}})$ in $W_{CD \to D^{\mathrm{on}}}$.

Definition 26 requires that $I3(B^{\mathrm{off}}) \diamond I5(C^{\mathrm{off}})$, $S5(B^{\mathrm{on}}) \diamond S3(C^{\mathrm{on}})$. These constraints are manifested in the example above as $[j, i]$ being present in the identity strand of $B^{\mathrm{off}}/B^{\mathrm{on}}$ and $[i^*, j^*]$ in the identity strand of $C^{\mathrm{off}}/C^{\mathrm{on}}$, and $[i, k]$ being present in the state strand of $C^{\mathrm{off}}/C^{\mathrm{on}}$ and $[k^*, i^*]$ in the state strand of $B^{\mathrm{on}}$. Consequently $AB$ and $BC$ can bind by the two contiguous toehold domains $I3(B^{\mathrm{off}})$, $I5(C^{\mathrm{off}})$, and $F_{AB \to B^{\mathrm{on}}}$ can bind with $F_{CD \to D^{\mathrm{on}}}$ and $W_{CD \to D^{\mathrm{on}}}$ by the two contiguous toehold domains in $S5(B^{\mathrm{on}})$, $S3(C^{\mathrm{on}})$. No other violations of condition 4 occur in the proposed implementation. ◀

▶ **Theorem 12** (Long cascades are non-realisable due to a particular type of leak reaction only). *Consider the set of reactions $\mathrm{A}_1 \to \mathrm{A}_2$, $\mathrm{A}_2 \to \mathrm{A}_3 \ldots \mathrm{A}_{N-1} \to \mathrm{A}_N$ for $N > 3$, in which all $\mathrm{A}_i$ are distinct. This network would be realisable if reactions between ancillary species $A_i A_{i+1}$ and $A_{i+2} A_{i+3}$, and $F_{A_i A_{i+1} \to A^{\mathrm{on}}_{i+1}}$ and $F_{A_{i+2} A_{i+3} \to A^{\mathrm{on}}_{i+3}}/W_{A_{i+2} A_{i+3} \to A^{\mathrm{on}}_{i+3}}$, were absent.*

**Proof.** By induction. Assume that an implementation of a cascade of length $N > 3$ exists in which: (a) for any toehold domain $x$ present in the downstream [upstream] end of $A_M^{\text{off}}$ or $A_M^{\text{on}}$, $S5(A_M^{\text{off}}), S5(A_M^{\text{on}}), I3(A_M^{\text{off}}) = I3(A_M^{\text{on}})$ $\left[S3(A_M^{\text{off}}) = S3(A_M^{\text{on}}), I5(A_M^{\text{off}}) = I5(A_M^{\text{on}})\right]$, the presence of $x$ and $x^*$ in major species is restricted to the downstream [upstream] end of $A_M^{\text{off}}$ and $A_M^{\text{on}}$ and the upstream [downstream] end of $A_{M+1}^{\text{off}}$ and $A_{M+1}^{\text{on}}$ [$A_{M-1}^{\text{off}}$ and $A_{M-1}^{\text{on}}$], $S3(A_{M+1}^{\text{off}}) = S3(A_{M+1}^{\text{on}}), I5(A_{M+1}^{\text{off}}) = I5(A_{M+1}^{\text{on}})$ $\left[S5(A_{M-1}^{\text{off}}), S5(A_{M-1}^{\text{on}}), I3(A_{M-1}^{\text{off}}) = I3(A_{M-1}^{\text{on}})\right]$; and (b) the only violations of realisability are those stated in this theorem. Lemma 10 gives an implementation for $N = 4$ satisfying these conditions.

Let us consider adding a new layer $\mathrm{A}_{N+1}$ to the cascade. The toeholds $S3(A_{N+1}^{\text{off}}) = S3(A_{N+1}^{\text{on}}), I5(A_{N+1}^{\text{off}}) = I5(A_{N+1}^{\text{on}})$ are complements of $S5(A_N^{\text{on}}), I3(A_N^{\text{on}})$, respectively, and the toeholds $S5(A_{N+1}^{\text{off}}), S5(A_{N+1}^{\text{on}}), I3(A_{N+1}^{\text{off}}) = I3(A_{N+1}^{\text{on}})$ can be orthogonal to all other toeholds by Assumption 5. This choice preserves assumption (a) above for the $N+1$-layer cascade. Definition 26 indicates that when the implied ancillary species are included, $\mathrm{A}_N \to \mathrm{A}_{N+1}$ as required by condition 1 of Definition 6. Moreover, the only toeholds in the new species are either non-complementary to the rest of the network, or taken from the upstream and downstream ends of $A_N^{\text{off}}$ and $A_N^{\text{on}}$. By (a), these toeholds are only present in major species of nodes $\mathrm{A}_{N-1}$, $\mathrm{A}_N$ and $\mathrm{A}_{N+1}$ and the ancillary species associated with them. To identify violations of conditions 2-4 of Definition 6, it is therefore sufficient to consider the isolated 4-level cascade $\mathrm{A}_{N-2} \to \mathrm{A}_{N-1} \to \mathrm{A}_N \to \mathrm{A}_{N+1}$ only. This analysis proceeds exactly as in Lemma 10; the proposed $N+1$-layer cascade therefore preserves assumption (b) as well as (a). Given that a domain-level implementation satisfying assumption (a) and (b) is given in Lemma 10 for $N = 4$, we therefore conclude that an implementation satisfying (a) and (b) can be constructed for arbitrary $N > 3$. Consequently there are no restrictions on realisability of cascades for $N > 3$ other than those stated in the theorem. ◀

▶ **Theorem 14** (Self interactions and bidirectional edges are not realisable). *Consider a system of reactions* $\mathrm{A}_1 \to \mathrm{A}_2 \to \mathrm{A}_3 \ldots \mathrm{A}_{N-1} \to \mathrm{A}_1$. *This network is not realisable if* $N \leq 2$.

**Proof.** The result for $N = 1$ is a direct consequence of Theorem 13. For $N = 2$, consider the set of reactions: $\mathrm{A} \to \mathrm{B}$, $\mathrm{B} \to \mathrm{A}$. By Definition 26, $\mathrm{A} \to \mathrm{B}$ implies $I3(A^{\text{off}}) \diamond I5(B^{\text{off}})$ and $IL(A^{\text{off}}) \diamond IL(B^{\text{off}})$. In addition, $\mathrm{B} \to \mathrm{A}$ implies $I5(A^{\text{off}}) \diamond I3(B^{\text{off}})$. The identity strands of A and B are then fully complementary, violating condition 3 of Definition 6. ◀

▶ **Theorem 16** (Long feedback loops with an even number of units are non-realisable due to a particular type of leak reaction only). *Consider the feedback loop* $\mathrm{A}_1 \to \mathrm{A}_2, \mathrm{A}_2 \to \mathrm{A}_3, \ldots \mathrm{A}_{N-1} \to \mathrm{A}_N, \mathrm{A}_N \to \mathrm{A}_1$. *For $N$ even, $N \geq 4$, this network would be realisable if reactions between ancillary species $A_iA_{i+1}$ and $A_{i+2}A_{i+3}$, and $F_{A_iA_{i+1} \to A_{i+1}^{\text{on}}}$ and $F_{A_{i+2}A_{i+3} \to A_{i+3}^{\text{on}}}/W_{A_{i+2}A_{i+3} \to A_{i+3}^{\text{on}}}$, were absent. Here, the index $j$ in $\mathrm{A}_j$ should be interpreted modularly: $\mathrm{A}_j = \mathrm{A}_{j-N}$ for $j > N$.*

**Proof.** For $N$ even, $N \geq 4$, a loop obeying condition 1 of Definition 6 can be constructed from the cascades identified in the proof of Theorem 12 by setting the otherwise unconstrained toeholds $S5(A_N^{\text{on}}), I3(A_N^{\text{on}})$ to $S5(A_N^{\text{on}}) \diamond S3(A_1^{\text{off}}) = S3(A_1^{\text{on}}), I3(A_N^{\text{on}}) \diamond I5(A_1^{\text{off}}) = I5(A_1^{\text{off}})$. To identify the violations of realisability that arise from conditions 2-4 of Definition 6, let us first consider a cascade without the $\mathrm{A}_N \to \mathrm{A}_1$ reaction. The only violations of realisability are those identified in Theorem 12: between $A_iA_{i+1}$ and $A_{i+2}A_{i+3}$, and $F_{A_iA_{i+1} \to A_{i+1}^{\text{on}}}$ and $F_{A_{i+2}A_{i+3} \to A_{i+3}^{\text{on}}}/W_{A_{i+2}A_{i+3} \to A_{i+3}^{\text{on}}}$, without interpreting the index modularly. Now we consider the additional effect of requiring $\mathrm{A}_N \to \mathrm{A}_1$. The only domains that must be changed are $S5(A_N^{\text{on}})$ and $I3(A_N^{\text{on}})$. These domains and their complements are only present in the species of $\mathrm{A}_{N-1} \to \mathrm{A}_N$, $\mathrm{A}_N \to \mathrm{A}_1$, $\mathrm{A}_1 \to \mathrm{A}_2$, and so it is sufficient to consider only this cascade to identify additional violations of realisability. By Lemma 10, the resultant violations of realisability are exactly those stated in the theorem. ◀

▶ **Theorem 22** (Mismatches successfully destabilize unintended complexes)**.** *The scheme proposed in Definition 20 satisfies the following:*

1. *All motifs that are realisable in the mismatch-free ACDC design remain realisable in the mismatch-based scheme.*
2. *Cascades of arbitrary length $N$ with at most the first and last reactions deactivating are realisable;*
3. *Feedback loops with $N$ even and $N \geq 6$ in which all reactions are activating are realisable;*
4. *Feedforward loops with $N \geq 1$, $M \geq 1$, $N - M$ even, in which at most the first and last reactions are deactivating in each branch, are realisable.*

**Proof.** Consider the first claim. For any network in which it is possible to select domains that satisfy Definition 25 and Definition 26, it is trivial to convert those domains to satisfy 25 and 27 by introducing the mismatches in major species, and adjusting ancillary species compensate. By Assumption 19, these changes do not introduce new violations of realisability.

Now consider the second claim. By the first claim and the construction in Theorem 12, it is sufficient to consider whether the sequestration reactions identified in Lemma 10 for an $N = 4$ cascade occur in the mismatch-based scheme of Definition 20. First, consider the unbound domains in the ancillary species in the system $A \rightarrow / \dashv B \rightarrow C \rightarrow / \dashv D$, with mismatches placed as per Definition 20:

- $I5(A^{\text{off}})$, $I3(B^{\text{off}})$ in $AB$
- $S3(A^{\text{on}})$, $S5(B^{\text{on}})$ in $F_{AB\rightarrow B^{\text{on}}}/W_{AB\rightarrow B^{\text{off}}}$
- $S3(A^{\text{on}})$, $S5(B^{\text{off}})$ in $W_{AB\rightarrow B^{\text{on}}}/F_{AB\rightarrow B^{\text{off}}}$
- $I5(B^{\text{off}})$, $I3(C^{\text{off}})$ in $BC$
- $S3(B^{\text{on}})$, $S5(C^{\text{on}})$ in $F_{BC\rightarrow C^{\text{on}}}$
- $S3(B^{\text{on}})$, $S5(C^{\text{off}})$ in $W_{BC\rightarrow C^{\text{on}}}$
- $I5(C^{\text{off}})$, $I3(D^{\text{off}})$ in $CD$
- $S3(C^{\text{on}})$, $S5(D^{\text{on}})$ in $F_{CD\rightarrow D^{\text{on}}}/W_{CD\rightarrow D^{\text{off}}}$
- $S3(C^{\text{on}})$, $S5(D^{\text{off}})$ in $W_{CD\rightarrow D^{\text{on}}}/F_{CD\rightarrow D^{\text{off}}}$.

By Definition 27, the reaction B $\rightarrow$ C implies $I3(B^{\text{off}})\Box I5(C^{\text{off}})$, $S5(B^{\text{on}})\Box S3(C^{\text{on}})$. Moreover, $\neg S5(B^{\text{off}}) \diamond S3(C^{\text{on}})$. By Assumption 19, none of the violations of realisability that would otherwise occur due to binding of $AB$ and $CD$; $F_{AB\rightarrow B^{\text{on}}}$ $W_{AB\rightarrow B^{\text{off}}}$ and $F_{CD\rightarrow D^{\text{on}}}$ $W_{CD\rightarrow D^{\text{off}}}$; and $F_{AB\rightarrow B^{\text{on}}}$ $W_{AB\rightarrow B^{\text{off}}}$ and $W_{CD\rightarrow D^{\text{on}}}$ $F_{CD\rightarrow D^{\text{off}}}$ characterised by Lemma 10, occur. Note that if B $\dashv$ C in the above network, Definition 27 implies $S5(B^{\text{on}}) \diamond S3(C^{\text{on}})$, meaning that sequestration reactions still occur between ancillary species. Cascades with deactivation reactions as intermediate steps are therefore not realisable.

Now consider the third claim. By the construction in Theorem 16 and the first claim of this Theorem, it is sufficient to consider only the sequestration reactions listed in Theorem 16. Further, since the only difference between a feedback loop with exclusively activating interactions and an activating cascade with $N$ species is that $\text{A}_N \rightarrow \text{A}_1$, by the second claim of this Theorem we need only consider changes in realisability due to the introduction of $\text{A}_N \rightarrow \text{A}_1$ to a cascade. For $N \geq 6$, imposing $I3(A_N^{\text{off}})\Box I5(A_1^{\text{off}})$, $S5(A_N^{\text{on}})\Box S(3)A_1^{\text{on}}$, as required by $\text{A}_N \rightarrow \text{A}_1$, does not create new realisability violations for a cascade of length $N$ with exclusively activating reactions. The ancillary species of the reactions $\text{A}_{N-2} \rightarrow \text{A}_{N-1}, \text{A}_{N-1} \rightarrow \text{A}_N, \text{A}_N \rightarrow \text{A}_1, \text{A}_1 \rightarrow \text{A}_2, \text{A}_2 \rightarrow \text{A}_3$ can only form complexes held together by two contiguous toehold domains with a central mismatch, and thus do not violate realisability by Assumption 19.

The above argument does not apply to FBLs of length $N = 4$, which remain non-realisable. In that case, adding the reaction $A_N \rightarrow A_1$ allows complexes of ancillary species bound by two separate sets of contiguous toehold domains, each with a central mismatch, either side of a 4-way junction. The short periodicity of an $N = 4$ loop means that the unwanted interaction identified in Lemma 10 happens twice for each pair of ancillary species. We do not assume in Assumption 19 that such a structure will dissociate. We also note that feedback loops with any deactivating reactions remain non-realisable, since each reaction $A_i \rightarrow A_{i+1}$ is effectively an intermediate reaction between $A_{i-1} \rightarrow A_i$ and $A_{i+1} \rightarrow A_{i+2}$.

Finally we turn to the fourth claim. By the first claim of this Theorem, and Theorem 17, it is sufficient to consider only the potential unwanted sequestration reactions between ancillary species identified in Theorem 17 for each feed-forward branch. The proof is then identical to that of the second claim of this Theorem. ◀

# Design Automation of Polyomino Set That Self-Assembles into a Desired Shape

**Yuta Matsumura**
Department of Robotics, Graduate School of Engineering, Tohoku University, Sendai, Japan
matsumura@molbot.mech.tohoku.ac.jp

**Ibuki Kawamata**
Department of Robotics, Graduate School of Engineering, Tohoku University, Sendai, Japan
Natural Science Division, Faculty of Core Research, Ochanomizu University, Tokyo, Japan
kawamata@molbot.mech.tohoku.ac.jp

**Satoshi Murata**
Department of Robotics, Graduate School of Engineering, Tohoku University, Sendai, Japan
murata@molbot.mech.tohoku.ac.jp

## Abstract

The problem of finding the smallest DNA tile set that self-assembles into a desired pattern or shape is a research focus that has been investigated by many researchers. In this paper, we take a polyomino, which is a non-square element composed of several connected square units, as an element of assembly and consider the design problem of the minimal set of polyominoes that self-assembles into a desired shape. We developed a self-assembly simulator of polyominoes based on the agent-based Monte Carlo method, in which the potential energy among the polyominoes is evaluated and the simulation state is updated toward the direction to decrease the total potential. Aggregated polyominoes are represented as an agent, which can move, merge, and split during the simulation. In order to search the minimal set of polyominoes, two-step evaluation strategy is adopted, because of enormous search space including many parameters such as the shape, the size, and the glue types attached to the polyominoes. The feasibility of the proposed method is shown through three examples with different size and complexity.

**2012 ACM Subject Classification** Applied computing → Systems biology; Applied computing → Chemistry; Hardware → Biology-related information processing

**Keywords and phrases** DNA polyomino, DNA nanostructure, DNA tile, Agent based simulation, Self-assembly, Combinatorial optimization, Simulated annealing

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.8

## 1 Introduction

As a method of creating artificial nanostructures, programed self-assembly of molecules is attracting attentions [3, 5, 7]. Since DNA has an excellent property of double helix formation between complementary base sequences, it is thought to be the most promising molecule for this purpose. One of the methods to make DNA nanostructures is called DNA tile [11, 12, 2]. In this method a unit called DNA tile composed of a few short DNA strands assemble into a large two-dimensional nanostructure. The DNA tile is a rectangle molecule having sticky ends (i.e. bonding edges with sequence specificity) on its sides. By arranging the sticky ends, it is possible to program the connectivity between the tiles. We can design a tile set to assemble periodic or aperiodic patterns, while the production cost depends on the complexity of the tile set (e.g. the number of sticky end types and the number of tile types), also the more complicated the tile set, the lower the quality and yield of the obtained assembly. From this point of view, the problem of finding the smallest tile set that forms the desired pattern (Pattern self-Assembly Tile-set Synthesis, PATS) has been studied [4, 1].


In this paper, we deal with self-assembly problem of a non-square element composed of several connected square units called a polyomino. We propose an algorithm to search for the minimum set of polyominoes required to assemble a desired outer shape. By using a polyomino as an element of assembly, it becomes possible to utilize the shape complementarity of the polyomino, in addition to the complementarity of sticky ends on the polyomino. This enables us to make relatively complex shapes also given as connected polyominoes. Since DNA origami technique enables us to make various three-dimensional shapes, it is expected that such non-square-shaped element made by DNA origami will allow us to construct a large structure with desired shape.

In the following sections, we consider the problem of finding the smallest polyomino set to fill a given shape. In Section 2, we introduce an assembly model that simulates the stochastic process of polyomino assembling. Section 3 describes a searching method for the simplest polyomino set that forms the target shape. In Section 4, we show the results of automatic design for target shapes with different size and complexity to verify the validity of the proposed method. Section 5 gives discussions.

# 2 Self-assembly model of polyominoes

## 2.1 Outline

This section explains the mathematical model of a polyomino and then introduces a stochastic simulation technique to predict the behavior of polyominoes. Unlike the abstracted kinetic tile assembly model [9, 10, 6], we employ an agent-based technique for the simulation.

Our model is illustrated in Fig. 1. The following summarises the outline.

- *Polyomino* is represented as a set of connected *square units*.
- An integer number named *glue type* is assigned to each side of the square unit.
- *Agent* is defined as a naive set of polyominoes.
- At the beginning of simulation, agents are randomly distributed over discretized space.
- In each step of simulation, agents can translate or rotate in the space.
- Potential energy computed from the interactions among polyominoes is minimized through the agent-based Monte Carlo simulation.

## 2.2 Square unit

A polyomino consists of several connected square units. To define the square unit, we need some prerequisite notations. $\mathrm{D} = \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}\}$ is a set of four cardinal directions (north, east, south, west) such that $\hat{\mathrm{N}} = \mathrm{S}, \hat{\mathrm{S}} = \mathrm{N}, \hat{\mathrm{E}} = \mathrm{W}, \hat{\mathrm{W}} = \mathrm{E}$. The neighboring cell of $\boldsymbol{x} = (x, y) \in \mathbb{N}^2$ in the direction $d \in \mathrm{D}$ is given by $\mathrm{coord}(\boldsymbol{x}, d)$ assuming a periodic boundary condition of the square lattice space.

$$\mathrm{coord}(\boldsymbol{x}, d) = \begin{cases} (x, y-1 \bmod m_{\mathrm{cell}}) & (d = \mathrm{N}) \\ (x+1 \bmod n_{\mathrm{cell}}, y) & (d = \mathrm{E}) \\ (x, y+1 \bmod m_{\mathrm{cell}}) & (d = \mathrm{S}) \\ (x-1 \bmod n_{\mathrm{cell}}, y) & (d = \mathrm{S}) \end{cases},$$

where $m_{\mathrm{cell}}, n_{\mathrm{cell}} \in \mathbb{N}$ are the total number of rows and columns in the lattice, respectively. Hereafter, $m_{\mathrm{cell}}$ and $n_{\mathrm{cell}}$ are both set to 32.

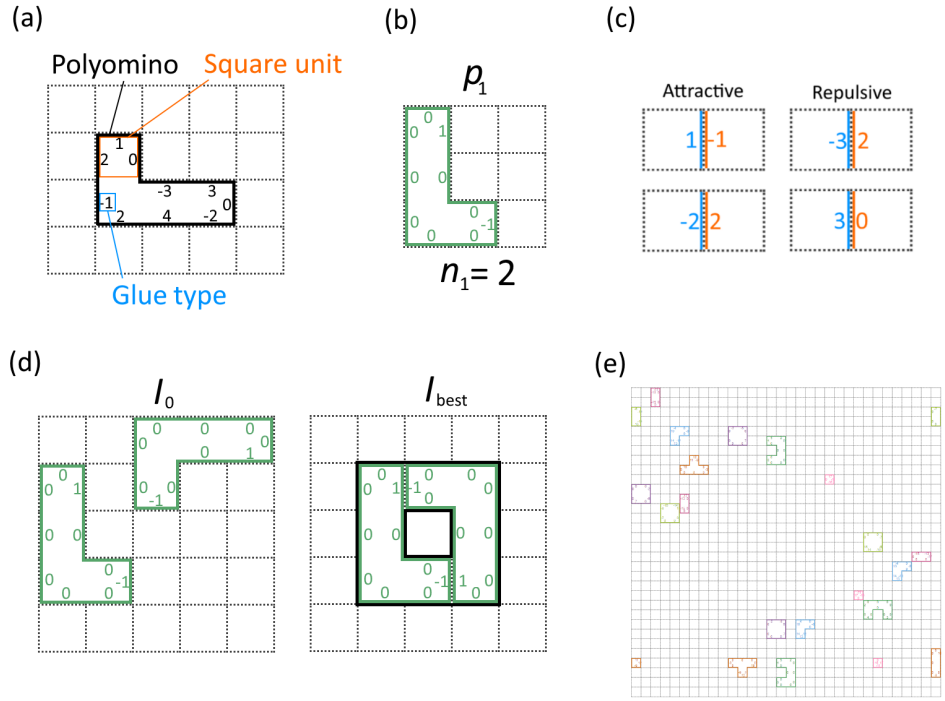



**Figure 1** (a) Model of polyomino. (b) polyomino sets. (c) Interaction between square unit. (d) Initial simulation state $I_0$ and most stable simulation state $I_{\text{best}}$. (e) Snapshot of the simulation.

Square unit $u$ is defined as a tuple of a position $x, y \in \mathbb{N}$ and a map $\mathrm{g} \in \mathbb{Z}^{\mathrm{D}}$ that gives the glue type of the cardinal direction D (i.e. $u = (x, y, \mathrm{g})$). The pair $(x, y)$, and coordinates $x$ and $y$ of a square unit $u = (x, y, \mathrm{g})$ can be obtained by $\mathrm{pos}(u) = (x, y)$, $\mathrm{pos_x}(u) = x$, $\mathrm{pos_y}(u) = y$, respectively. Similarly, the glue type of a square unit $u = (x, y, \mathrm{g})$ in the direction $d \in \mathrm{D}$ can be obtained by $\mathrm{gl}(u, d) = \mathrm{g}(d)$. Non-zero glue types $g_1$ and $g_2$ are *complementary* when $g_1 + g_2 = 0$ stands.

## 2.3 Polyomino

A polyomino $p$ is defined as a nonempty set of connected square units (i.e. $p = \{u_1, u_2, \ldots\}$ such that $\forall u_i, u_j \in p, \exists u'_1 = u_i, u'_2, \ldots, u'_k = u_j \in p, \forall l \in \{z \in \mathbb{N} | 1 \leq z \wedge z \leq k\}, \exists d \in \mathrm{D}, \mathrm{pos}(u'_{l+1}) = \mathrm{coord}(\mathrm{pos}(u'_l), d)$). To avoid an overlap, we assume that a square unit $u_1 \in p_1$ never belongs to other polyomino $p_2$ (i.e. $\forall u_1 \in p_1, u_2 \in p_2, u_1 = u_2 \rightarrow p_1 = p_2$). The center of mass of a polyomino $p$ is defined as $\mathrm{c_M}(p) = (\mathrm{c_x}(p), \mathrm{c_x}(p))$, where $\mathrm{c_x}(p) = \mathrm{round}(\sum_{u \in p} \mathrm{pos_x}(u)/|p|)$ and $\mathrm{c_y}(p) = \mathrm{round}(\sum_{u \in p} \mathrm{pos_y}(u)/|p|)$. The nearest integer is obtained by $\mathrm{round}(x) \in \mathbb{Z}$ from a real number $x \in \mathbb{R}$.

## 2.4 Movement of polyomino

A polyomino is capable of performing a movement $m \in \mathrm{M_{poly}}$, which is a map from polyominoes to polyominoes. Here, we define 7 possible movements : translation to the north, east, south or west, or rotation to the left, back or right. Here "back rotation" means rotation of 180 degrees. The set of these movements is defined as $\mathrm{M_{poly}} = \{\mathrm{north}, \mathrm{east}, \mathrm{south}, \mathrm{west}, \mathrm{right}, \mathrm{back}, \mathrm{left}\}$. Formal description of the movement is given in Appendix A.1.

Polyominoes $p_1$ and $p_2$ are *isomorphic* ($p_1 \equiv p_2$) when there are finite movements that can move $p_1$ to $p_2$, which is defined as $p_1 \equiv p_2 \leftrightarrow \exists n \in \mathbb{N}, \exists m_1, m_2, \ldots m_n \in \mathrm{M_{poly}}, m_1 \circ m_2 \circ \ldots \circ m_n(p_1) = p_2$. When $p_1$ and $p_2$ are not isomorphic, they are called *non-isomorphic*.

## 2.5 Polyomino species

The concept of polyomino set was ambiguously used so far to illustrate the goal of our research. Here, we introduce the formal definition of *polyomino species*, which is more accurate to describe the polyomino set. A polyomino species is a multiset of quotient set of polyominoes by the isomorphic relationship $\equiv$, which is not a naive set of polyomino. Namely, polyomino species $P$ can be expressed as a set of tuples of representative polyomino $p_i$ and its occurrence count $n_i$ (i.e. $P = \{(p_1, n_1), (p_2, n_2), \ldots\}$). The number of representative polyominoes is denoted as $|P|$, and the set of glue types in $P$ is defined as $\mathrm{Gl}(P) = \{\mathrm{gl}(u, d) \in \mathrm{D}, u \in p, (p, n) \in P\}$. We say polyomino set to simply explain the target problem, although it formally means polyomino species throughout this paper.

## 2.6 Agent

In the proposed simulation model, we introduce a concept of agent which represents a naive set of polyominoes [8]. Instead of applying the movement to each polyomino, we move the agent in order to improve energy convergence.

Agent $a = \{p_1, p_2, \ldots\}$ is a non-empty set of polyominoes, connected by the complementary glue types. (i.e. $a = \{p_1, p_2, \ldots\}$ such that $\forall p_i, p_j \in a, \exists p'_1 = p_i, p'_2, \ldots, p'_k = p_j \in a, \exists u \in p'_{l+1}, \exists u' \in p_l, \forall l \in \{z \in \mathbb{N} | 1 \leq z \wedge z \leq k\}, \exists d \in \mathrm{D}, \mathrm{pos}(u) = \mathrm{coord}(\mathrm{pos}(u'), d) \wedge \mathrm{gl}(u, \hat{d}) + \mathrm{gl}(u', d) = 0 \wedge \mathrm{gl}(u', d) \neq 0)$. We define a set of square units in an agent $a$ as $\mathrm{U}(a)$ and the number of square units in the agent $a$ as $|\mathrm{U}(a)|$. Similar to the polyomino, there are also 7 movements $\mathrm{M_{agent}}$ for the agent (see Appendix A.1).

At the beginning of the simulation, each polyomino is assumed to belong to a different agent, and is able to move independently. Through the simulation process, agents can *merge* or *split*, resulting in a unified movement of several polyominoes. Details of the process is described in the following.

## 2.7 Simulation state

*Simulation state* $I = \{a_1, a_2, \ldots\}$ is defined as a set of agent at specific time step. We define a naive set of polyominoes in the simulation state $I$ as $\mathrm{P}(I) = \{p | p \in a, a \in I\}$, and a set of square units as $\mathrm{U}(I) = \{u | u \in p, p \in \mathrm{P}(I)\}$.

The initial simulation state $I_0$ is defined for a given polyomino species $P = \{(p_1, n_1), (p_2, n_2), \ldots\}$. There are $n_i$ copies of polyomino $p_i$ without any overlap at the beginning. Namely, $\forall u_1, u_2 \in \mathrm{U}(I_0), \mathrm{pos}(u_1) = \mathrm{pos}(u_2) \rightarrow u_1 = u_2$.

## 2.8 Cluster

A *cluster* $c$ is a naive set of polyominoes in a simulation state $I$, such that there are no polyomino $p \in \mathrm{P}(I) \backslash c$ neighboring to $c$. This is formalized as $\forall p_1 \in c, \forall p_2 \in \mathrm{P}(I) \backslash c, \forall u_1 \in p_1, \forall u_2 \in p_2, \forall d \in \mathrm{D}, \mathrm{coord}(\mathrm{pos}(u_1), d) \neq \mathrm{pos}(u_2)$. Note that a cluster does not necessarily have to contain polyominoes with matching glues. Unlike agents that can translate and rotate, the cluster only refers to an static assembly of polyominoes. They are used to evaluate the state of the simulation. When a simulation state $I$ is given, the set of all clusters are defined as $\mathrm{cl}(I)$.

The same movements $\mathrm{M_{agent}}$ of agent can be applied to cluster (see Appendix A).

## 2.9 Potential energy

During the simulation, the total *potential energy* is evaluated as a sum of local energy gained from interactions among the square units. When two units are not neighboring, there is no local energy between them. If they are located in the neighboring cells, there is an attractive force between them when the facing glue types are complementary, otherwise, there is a repulsive force. Given two square units $u_1$ and $u_2$, the local energy between them $\mathrm{e}_{\mathrm{unit}}(u_1, u_2)$ is defined as

$$\mathrm{e}_{\mathrm{unit}}(u_1,u_2) = \begin{cases} 0 & (\forall d \in \mathrm{D}, \mathrm{coord}(\mathrm{pos}(u_1),d) \neq \mathrm{pos}(u_2)) \\ e_{\mathrm{att}} & (\exists d \in \mathrm{D}, \mathrm{coord}(\mathrm{pos}(u_1),d) = \mathrm{pos}(u_2) \wedge \mathrm{gl}(u_1,d) + \mathrm{gl}(u_2,\hat{d}) = 0) \wedge \mathrm{gl}(u_1,d) \neq 0) \\ e_{\mathrm{rep}} & (\text{otherwise}) \end{cases},$$

where $e_{\mathrm{att}}$ and $e_{\mathrm{rep}}$ are local energy caused by the attractive and the repulsive forces, respectively. Hereafter, we use $e_{\mathrm{att}} = -11$ and $e_{\mathrm{rep}} = 2$, referring to a reported agent-based simulation method [8].

The potential energy $\mathrm{e}_{\mathrm{poly}}(p_1, p_2)$ between polyominoes $p_1, p_2$ is a sum of all energy of the square units in them. Namely,

$$\mathrm{e}_{\mathrm{poly}}(p_1,p_2) = \begin{cases} \sum_{u_1 \in p_1, u_2 \in p_2} \mathrm{e}_{\mathrm{unit}}(u_1,u_2) & (p_1 \neq p_2) \\ 0 & (\text{otherwise}) \end{cases}.$$

Similarly, the potential energy $\mathrm{e}_{\mathrm{agent}}(a_1, a_2)$ between agents $a_1, a_2$ can be defined as

$$\mathrm{e}_{\mathrm{agent}}(a_1,a_2) = \begin{cases} \sum_{p_1 \in a_1, p_2 \in a_2} \mathrm{e}_{\mathrm{poly}}(p_1,p_2) & (a_1 \neq a_2) \\ 0 & (\text{otherwise}) \end{cases}.$$

For convenience, we also define the potential $\mathrm{e}_{\mathrm{in}}(a)$ of a given agent $a$ as

$$\mathrm{e}_{\mathrm{in}}(a) = \sum_{p_1,p_2 \in a} \mathrm{e}_{\mathrm{poly}}(p_1,p_2)/2.$$

The total potential energy $\mathrm{e}_{\mathrm{state}}(I)$ of a given simulation state $I$ is defined as

$$\mathrm{e}_{\mathrm{state}}(I) = \sum_{a_1,a_2 \in I} \mathrm{e}_{\mathrm{agent}}(a_1,a_2)/2.$$

## 2.10 Time development of the simulation state

By using the agent-based simulation, we are able to minimize the total energy of a simulation state. From a state $I_i$ of $i$-th step of the simulation, the next state $I_{i+1}$ can be obtained by the algorithm shown in Fig. 2. First, an agent $a_{\mathrm{sel}}$ is randomly selected from the state $I_i$, and one of the three actions (i.e. split, move or merge) takes place to update the state. If none of the actions are admissible, $I_i$ becomes the next state.

- Split of agent
  Namely, the agent $a_n$ with an energy $\mathrm{e}_{\mathrm{in}}(a_n)$ is split into two, if it is composed of $n$ ($n \geq 2$) polyominoes that satisfy

  $$\exists i \in \mathbb{N},\ 1 < i \leq n, \mathrm{e}_{\mathrm{in}}(a_n)/n > \mathrm{e}_{\mathrm{min}}(i,n)/i,$$

  where $\mathrm{e}_{\mathrm{min}}(i, n)$ is the smallest energy of the agent with $i$ square units among all the simulation states before the current $n$-th step. Namely,

  $$\mathrm{e}_{\mathrm{min}}(i,n) = \min(\bigcup_{j \leq n} \{\mathrm{e}_{\mathrm{in}}(a) | a \in I_j \wedge |\mathrm{U}(a)| = i\}).$$

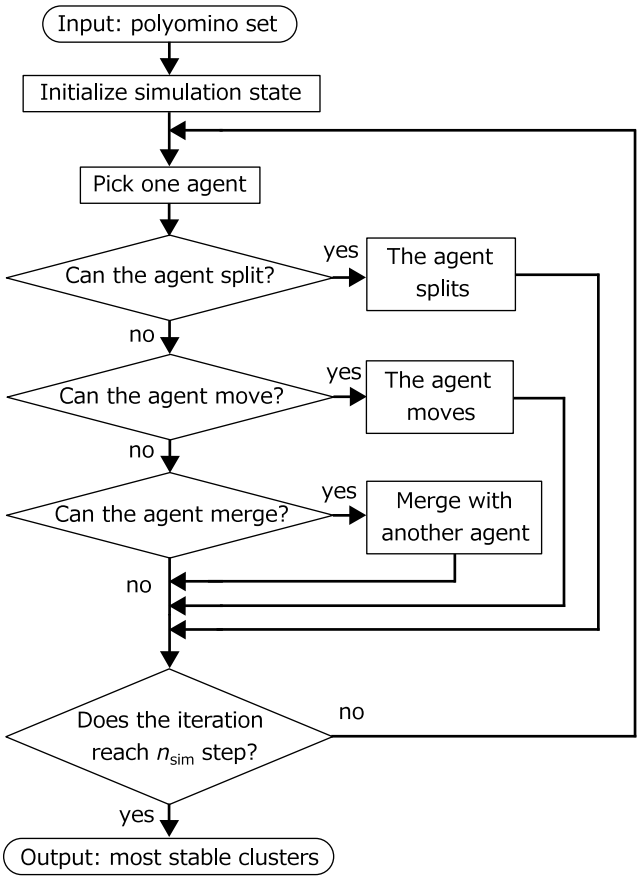



**Figure 2** Flowchart of simulation.

An agent can be split in several ways. One polyomino $p_1$ is removed from the agent and becomes a new agent when $p_1$ has the worst (biggest) contribution to the potential. The polyomino $p_1$ satisfies $\forall p_2 \in a_{\mathrm{sel}}, \mathrm{e}_{\mathrm{agent}}(a_{\mathrm{sel}}, \{p_1\}) \leq \mathrm{e}_{\mathrm{agent}}(a_{\mathrm{sel}}, \{p_2\})$. When splitting takes place, the next simulation state $I_{i+1}$ becomes $I_i \backslash \{a_{\mathrm{sel}}\} \cup \{a_{\mathrm{sel}} \backslash \{p_1\}, \{p_1\}\}$.

- Move of agent

When the agent cannot split, one or several polyominoes try to move together as a unified agent. When the agent $a_{\mathrm{sel}}$ takes a move $m \in \mathrm{M}_{\mathrm{agent}}$, a simulation state transits to $I_{i+1}^m = I_i \backslash \{a_{\mathrm{sel}}\} \cup \{m(a_{\mathrm{sel}})\}$. As there are 7 movements, there are 7 possible simulation states $I_{i+1}^{\mathrm{north}}$, $I_{i+1}^{\mathrm{east}}$, $I_{i+1}^{\mathrm{south}}$, $I_{i+1}^{\mathrm{west}}$, $I_{i+1}^{\mathrm{right}}$, $I_{i+1}^{\mathrm{back}}$, $I_{i+1}^{\mathrm{left}}$. One of them is stochastically selected as the next simulation state $I_{i+1}$ with the probability $\mathrm{P}(I_i, I_{i+1}^m)$ given as

$$\mathrm{P}(I_i, I_{i+1}^m) = \begin{cases} \frac{\min(1, \exp((\mathrm{e}_{\mathrm{state}}(I_i) - \mathrm{e}_{\mathrm{state}}(I_{i+1}^m))/k_{\mathrm{B}}\tau_{\mathrm{sim}}))}{|\mathrm{M}_{\mathrm{agent}}|} & (\text{condition A}) \\ 0 & (\text{otherwise}) \end{cases},$$

where $\tau_{\mathrm{sim}}$ is a temperature parameter introduced to overcome the energetic barrier (i.e. local minima), and $k_B$ is the Boltzmann constant. Hereafter, we use $\tau_{\mathrm{sim}} k_B = 5$, which is an empirically good value for the energy convergence. "Condition A" means that there is no overlap of agents as a result of the movement and the agent $a_{\mathrm{sel}}$ is not rotational symmetry, which can be formalized as

$$(\forall p_1, p_2 \in I_i^m, p_1 \neq p_2, \forall u_1 \in p_1, u_2 \in p_2, \mathrm{pos}(u_1) = \mathrm{pos}(u_2) \rightarrow u_1 = u_2) \wedge (\mathrm{m}(a_{\mathrm{sel}}) \neq \mathrm{m}(a_{\mathrm{sel}})).$$

- Merge of agents
  If all the possible movements increase the potential energy (i.e. $\forall m \in \mathrm{M_{agent}}, \mathrm{P}(I_i, I_{i+1}^m) < 1/|\mathrm{M_{agent}}|$) and also none of the movements are chosen by the calculated possibilities, the agent then try to merge with a neighboring agent. This condition implies that there is an attractive interaction between $a_{\mathrm{sel}}$ and the neighboring agent.
  The agent $a_{\mathrm{sel}}$ merges with another agent $a_1$ that can make the assembly most stable in respect to the potential energy. The agent $a_1$ satisfies $\forall a_2 \in I_i, \mathrm{e_{agent}}(a_{\mathrm{sel}}, a_1) \leq \mathrm{e_{agent}}(a_{\mathrm{sel}}, a_2)$. When the agent $a_{\mathrm{sel}}$ merges with the agent $a_1$, the simulation state becomes

$$I_{i+1} = I_i \backslash \{a_{\mathrm{sel}}, a_1\} \cup \{a_{\mathrm{sel}} \cup a_1\}.$$

The most stable simulation state is predicted by iterating the above state transition for $n_{\mathrm{sim}}$ times. When a polyomino species $P$ is given, the resulting assembly is defined as a set of clusters $\mathrm{A}(P)$ such that

$$\mathrm{A}(P) = \mathrm{cl}(I_{\mathrm{best}}) \qquad (\exists I_{\mathrm{best}} \in X, \forall I \in X, \mathrm{e_{state}}(I_{\mathrm{best}}) \leq \mathrm{e_{state}}(I)),$$

where $X$ is the set of simulation state through the simulation ($X = \{I_0, I_1, \ldots, I_{n_{\mathrm{sim}}}\}$).

# 3 Design automation

## 3.1 Criteria

By using the simulation model in Section 2, we solve a *shape self-assembly polyomino set* (SAP) problem, which is formalized as follows.

- The target assembly is given as a *shape* defined as a finite set of positions $s = \{(x_1, y_1), (x_2, y_2), \ldots\}$ with the size $m_{\mathrm{shape}} = \max(x_1, x_2, \ldots) - \min(x_1, x_2, \ldots)$, and $n_{\mathrm{shape}} = \max(y_1, y_2, \ldots) - \min(y_1, y_2, \ldots)$.
- If a polyomino species $P$ can construct a shape $s$ through self-assembly, then $P$ is said to be an polyomino species of $s$.
- The size of polyominoes in the polyomino species $P$ is less than or equal to $m_{\mathrm{poly}} \times n_{\mathrm{poly}}$, and must be smaller than that of the target shape $s$.
- There are no limitations on the number of representative polyominoes $|P|$ and glue types $|\mathrm{Gl(P)}|$.
- An optimum polyomino species for a shape of finite size is the polyomino species of minimum cardinality (i.e., with the smallest number of representative polyominoes).
- The SAP (shape self-assembly polyomino species) problem is defined as a problem to find the optimum polyomino species for a given finite-size shape.

## 3.2 Outline

To tackle the SAP problem, we employ a simulated-annealing algorithm which is one of the meta-heuristics approaches. The flowchart of the algorithm is given in Fig. 3. An initial polyomino species is randomly generated from a given shape $s$ and evaluated by the simulation. In our optimization strategy, a polyomino species is rated better when the predicted assembly is closer to the target shape, and also the number of representative polyominoes and the number of glue types are smaller. A polyomino species is gradually improved by repeating evolutionary process.

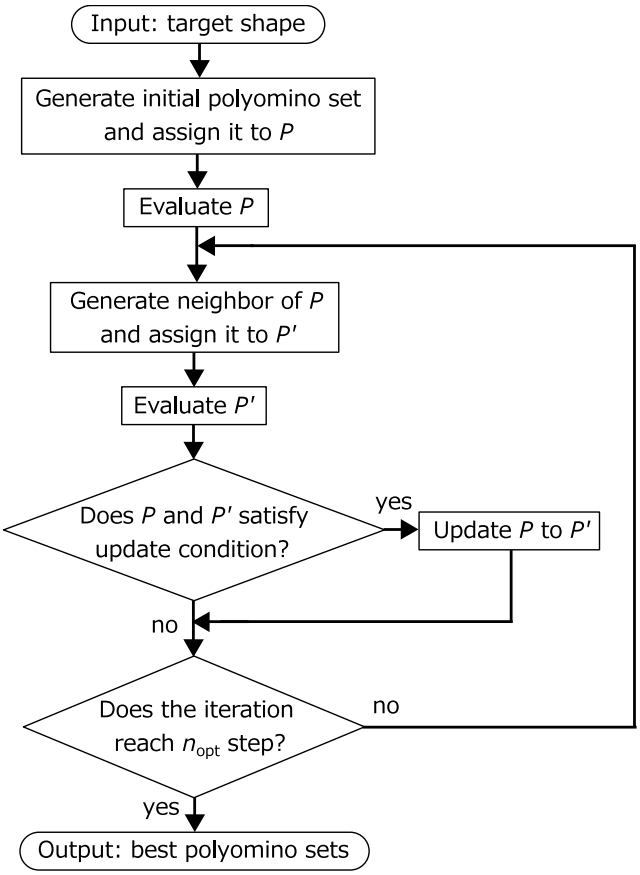



**Figure 3** Flowchart of automatic design.

## 3.3 Evaluation of polyomino species

To evaluate a polyomino species $P$, we introduce an inaccurate but light-cost function $\mathrm{loss}_{\mathrm{light}}$ and an accurate but heavy-cost function $\mathrm{loss}_{\mathrm{heavy}}$. The function is named "loss" because the smaller the value, the better the polyomino species. In order to minimize the time of computation, the light-cost function is first used for rough evaluation, then heavy-cost function is further used when it meets a certain criteria.

The light cost function is defined as

$$\mathrm{loss}_{\mathrm{light}}(P) = |P|^2 + \frac{1}{2}|\mathrm{Gl}(P)|.$$

When $\mathrm{loss}_{\mathrm{light}}(P) < \alpha_{\mathrm{th}}$ holds, the heavy-cost function is applied, where $\alpha_{\mathrm{th}} \in \mathbb{R}$ is a threshold parameter updated when $\mathrm{loss}_{\mathrm{heavy}}(P, s)$ is computed. By introducing $\alpha_{\mathrm{th}}$, the algorithm can efficiently search for polyomino species with a smaller loss value than current best value. The initial value of $\alpha_{\mathrm{th}}$ is $|s|^2 + 2|s|$, which is the maximum value of $\mathrm{loss}_{\mathrm{light}}(P)$ for given target shape $s$. The algorithm to update $\alpha_{\mathrm{th}}$ is

$$\alpha_{\mathrm{th}} := \begin{cases} \alpha_{\mathrm{th}} & (\mathrm{loss}_{\mathrm{heavy}}(P, s) - \mathrm{loss}_{\mathrm{light}}(P) > 0) \\ \min(\alpha_{\mathrm{th}}, \mathrm{loss}_{\mathrm{light}}(P)) & (\text{otherwise}) \end{cases}.$$

The condition indicates that $\alpha_{\mathrm{th}}$ is updated when all the clusters in $\mathrm{A}(P)$ have exactly the same shape as the target $s$.

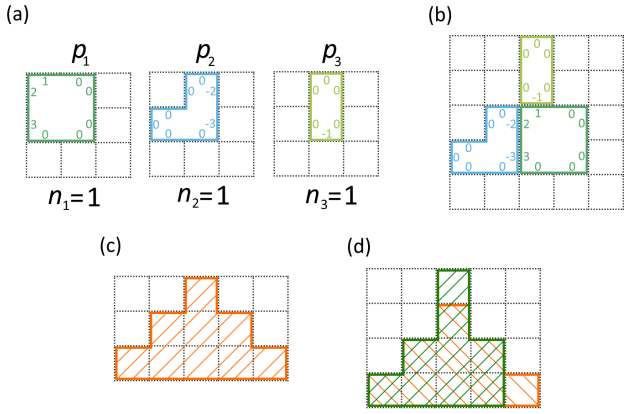



**Figure 4** Example of loss value calculation. (a) Polyomino species $P$. (b) Cluster $\mathrm{A}(P)$ which $P$ self-assembles into. (c) Target shape $s$. (d) A state that gives maximum overlap between the cluster and the shape.

The heavy-cost function is computationally heavy because it is necessary to estimate the formed clusters $\mathrm{A}(P)$ by the simulation. In order to define the heavy-cost function, we need to introduce a function to evaluate similarity between shapes.

The shape of a given cluster $c$ is represented as a set of $x, y$ coordinates in the cluster, $\mathrm{shape}(c) = \{\mathrm{pos}(u) \,|\, u \in p,\, p \in c\}$. The similarity $\mathrm{V_{ss}}$ between a cluster $c$ and a shape $s$ is defined as the number of square units that does not belong to the overlap, which is

$$\mathrm{V_{ss}}(c,\, s) = \sum_{\boldsymbol{x} \in \mathrm{shape}(c)} \mathrm{incl}(\boldsymbol{x},\, s) + \sum_{\boldsymbol{y} \in s} \mathrm{incl}(\boldsymbol{y},\, \mathrm{shape}(c)),$$

where

$$\mathrm{incl}(p,\, s) = \begin{cases} 0 & (p \in s) \\ -1 & (\text{otherwise}) \end{cases}.$$

The maximum volume of the similarity $\mathrm{V_{ss}^{max}}(c, s)$ is then defined by moving cluster $c$ to have the maximum overlap, which means

$$\mathrm{V_{ss}^{max}}(c,\, s) = \max(\bigcup_{n \in \mathbb{N}} \{\mathrm{V_{ss}}(c', s) | \forall m_1, m_2, \ldots m_n \in \mathrm{M_{agent}}, c' = m_1 \circ m_2 \circ \ldots \circ m_n(c)\}).$$

Using the above definitions, the heavy-cost function is defined as

$$\mathrm{loss_{heavy}}(P, s) = |P|^2 + \frac{1}{2}|\mathrm{Gl}(P)| + (\sum_{c \in \mathrm{A}(P)} \frac{\mathrm{V_{ss}^{max}}(c, s)}{|\mathrm{A}(P)|})^2.$$

As the result, a polyomino species $P$ is evaluated as

$$\mathrm{loss}(P,\, s) = \begin{cases} \mathrm{loss_{heavy}}(P, s) & (\mathrm{loss_{light}}(P) < \alpha_{\mathrm{th}}) \\ |s|^2 + 3|s| + n_{\mathrm{cell}} \times m_{\mathrm{cell}} & (\text{otherwise}) \end{cases}.$$

## 3.4 Search of polyomino species with low loss value

Fig. 5 illustrates an example of initial and neighbor polyomino species generation. The initial polyomino species is generated by randomly decomposing the target shape into smaller polyominoes. This process is realized by repeating following two actions after generating a naive set of polyomino $S = \{p\}$ that has only one element $p$ with the shape $s$.

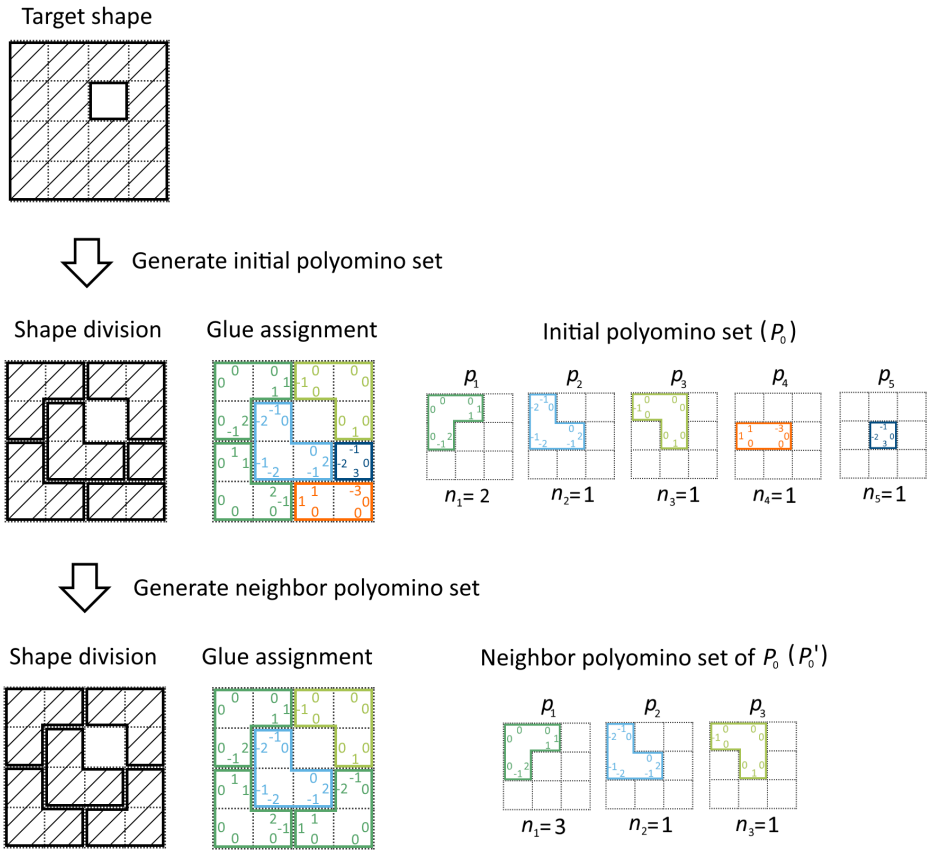



**Figure 5** Generation of initial polyomino species and neighbor polyomino species.

- Action1
  A square unit $u \in p$ is randomly selected. If randomly selected polyomino $p \in S$ satisfies "condition B", the algorithm removes the square unit $u$ from the polyomino $p$ and generate a new polyomino $\{u\}$. Here, condition B for a given polyomino $p$ means that there are no other polyominoes with the same shape, or the size of $p$ is smaller than or equal to $m_{\mathrm{fix}} \times n_{\mathrm{fix}}$. The $S$ is updated to $S \backslash \{p\} \cup \{p \backslash \{u\}, \{u\}\}$. The probability to perform this action is $r_{\mathrm{gen}}$.
- Action2
  A pair of neighboring polyominoes $p$ and $p'$ are randomly selected from $S$. If the polyominoes $p$ and $p'$ satisfy condition B, the algorithm removes a randomly selected square unit $u \in p$ from $p$ that is adjacent to $p'$, and add it to $p'$. The $S$ is updated to $S \backslash \{p, p'\} \cup \{p \backslash \{u\}, p' \cup \{u\}\}$. The probability to perform this action is $1 - r_{\mathrm{gen}}$.

If there are no polyominoes which meet condition B, one of these two actions takes place ignoring condition B. These two actions are repeated more than $n_{\mathrm{new}}$ steps, so that the sizes of all the polyominoes become smaller or equal to $m_{\mathrm{poly}} \times n_{\mathrm{poly}}$. Hereafter, we use $m_{\mathrm{fix}} = 2$ and $n_{\mathrm{fix}} = 1$ and $r_{\mathrm{gen}} = 0.05$, $n_{\mathrm{new}} = 100$.

Next, the algorithm assigns the glue types of polyominoes. Each glue type of polyominoes is set to all different value such that the square units have complementary glue types in contacting face with another polyomino. Glue types which are not contacting with any other polyomino are fixed to 0. Polyominoes with the equivalent shape are converted to equivalent polyominoes by assigning glue types properly (see Appendix A.2). From the set of polyominoes, corresponding polyomino species can be trivially constructed.

To make a neighbor polyomino species, one of the two decomposing actions is applied as a mutation and then new glue types are assigned.

When the current polyomino species is $P$ and its new neighbor is $P'$, the possibility to accept $P'$ is

$$\mathrm{P}(P,\, P') = \begin{cases} 1 & (\mathrm{loss}(P,\, s) - \mathrm{loss}(P',\, s) \geq 0) \\ \exp((\mathrm{loss}(P,\, s) - \mathrm{loss}(P',\, s)) \,/\, \tau_{\mathrm{sa}}) & (\mathrm{otherwise}) \end{cases},$$

where $\tau_{\mathrm{sa}}$ is a constant temperature parameter. Hereafter, we use $\tau_{\mathrm{sa}} = 10$, which empirically accepts 20% of transitions that increase the loss values. The total iteration $n_{\mathrm{opt}}$ is set to 100.

## 4 Result

To demonstrate the validity of proposed algorithm, we tested 3 target shapes with different complexities (small, medium, and large), where $m_{\mathrm{poly}}$ and $n_{\mathrm{poly}}$ are both set to 3. For each case, we run 100 searches for statistical analysis. The small target is given in $4 \times 4$ lattice (Fig. 6(a)). For this target, reasonably good polyomino species were always obtained such as the example in Fig. 6(b,c). The loss function development of 10 representative searches are shown in Fig. 6(d). The average loss value over 100 searches was 12.5 with a standard deviation of $\pm 0.37$, which is smaller than $67.3 \pm 14.6$ of 100 random searches that find the best candidate from randomly generated 100 polyomino species.

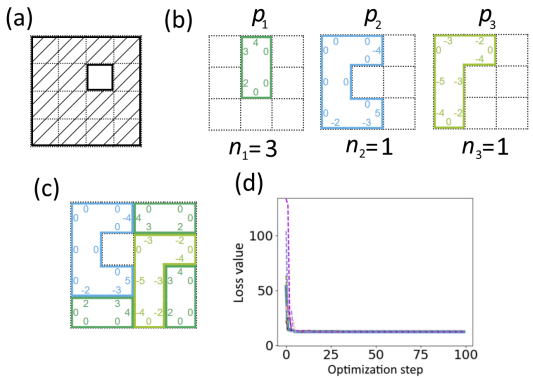



**Figure 6** (a) Target shape. (b) An example of polyomino species $P$. (c) Cluster which $P$ self-assembles into. (d) Development of loss function. The illustrated solution is shown in bold.

The medium target is given in a $5 \times 5$ lattice (Fig. 7(a)). A polyomino species that self-assembles into the target shape was also found as expected (Fig. 7(b,c)). The average loss value was $15.4 \pm 2.7$, which is significantly smaller than $134.2 \pm 29.0$ of random search. Ten representative results are shown in Fig. 7(d).

The large target is given in a $6 \times 6$ lattice (Fig. 8(a)). Some of the searches succeeded in finding a polyomino species that self-assembles into the target shape as in the example of Fig. 8(b,c). The polyominoes in the set, however, were all different and none of them were recycled in different places. The average loss value was $144.7 \pm 87.7$, which is smaller than $201.0 \pm 121.6$ of random search. Ten representative results are shown in Fig. 8(d).

The performance of the proposed algorithm is summarized in Fig. 9. In the small and medium cases, the loss values of proposed algorithm got significantly smaller than those of random search. In the large case, however, the difference between the proposed algorithm and random search was not as significant. This may due to insufficient iteration of the

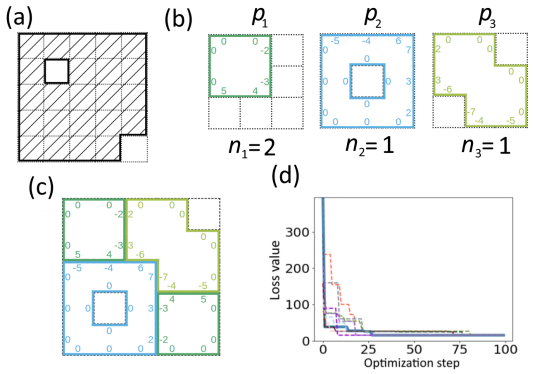



**Figure 7** (a) Target shape. (b) An example of polyomino species $P$. (c) Cluster assembled by $P$. (d) Development of loss function. The illustrated solution is shown in bold.

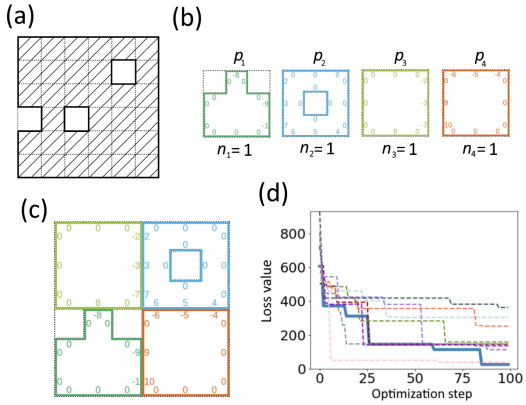



**Figure 8** (a) Target shape. (b) An example of polyomino species $P$. (c) Cluster assembled by $P$. (d) Development of loss function. The illustrated solution is shown in bold.

search. We further quantified the convergence speed of the search using a logarithmic fit. The development of loss values in respect to the logarithmic optimization step with estimated slopes are shown in inset of Fig. 9(b). The number of iteration that is necessary to optimize the polyomino species using our strategy may grow exponentially to the size of the target.

## 5 Discussion

In this paper, we consider the problem of finding minimum set of polyominoes that assemble into a desired shape. A simulator developed on the agent-based Monte Carlo method evaluates the potential energy among the polyominoes and updates the simulation state to decrease the total potential. Since the geometrical interactions between polyominoes have to be taken into account, the developed simulator become complicated compared with the simulators for homogeneous units such as kTAM, where a set of polyominoes is represented as an agent, which can move, merge, and split during the simulation. With this framework, a self-assembly processes of polyominoes can be efficiently simulated.

In the proposed algorithm, meta-heuristic method called simulated annealing was adopted. Because of the enormous search space for the design problem, a two-step evaluation strategy was adopted to prune unpromising solution spaces. Automatic design for three example targets with different size and complexity was tested to show the feasibility of the proposed method.

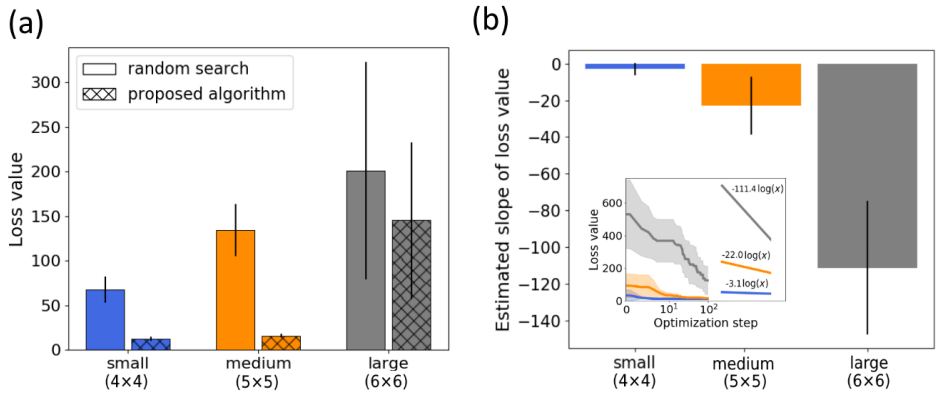



**Figure 9** (a) The average of loss values at the last iteration of the searches of small ($4 \times 4$), medium ($5 \times 5$), and large ($6 \times 6$) cases. The results of random search and proposed algorithms are compared. (b) Convergence speed of the proposed algorithm using a logarithmic fit. The inset shows the log-scale mean development of loss values, where standard deviation is illustrated as transparent area. The bars summarize the estimated slopes in the log-scale graph. The algorithms are run 100 times, and error bar indicates the standard deviation.

In order to solve a larger problem, we need to improve the efficiency of the algorithm, especially to reduce the computational cost of Monte Carlo simulation. For this purpose, it is necessary to redesign the potential energy between polyominoes to avoid kinetic traps. Introducing a new criterion to terminate the simulation at an appropriate step will be also effective. Larger-scale problems can be solved by introducing parallel computing hardware such as GPU along with the above improvements of the algorithms. From a computer science point of view, whether or not the automatic design problem of the polyomino set is NP is an interesting issue. Also, extending the problem to three-dimensional polycube is remained for a future work.

## References


1 Mika Göös and Pekka Orponen. Synthesizing minimal tile sets for patterned dna self-assembly. In *International Workshop on DNA-Based Computers*, pages 71–82. Springer, 2010. doi:10.1007/978-3-642-18305-8_7.

2 Yu He, Yi Chen, Haipeng Liu, Alexander E Ribbe, and Chengde Mao. Self-assembly of hexagonal dna two-dimensional (2d) arrays. *Journal of the American Chemical Society*, 127(35):12202–12203, 2005. doi:10.1021/ja0541938.

3 Chenxiang Lin, Yan Liu, Sherri Rinker, and Hao Yan. Dna tile based self-assembly: building complex nanoarchitectures. *ChemPhysChem*, 7(8):1641–1647, 2006. doi:10.1002/cphc.200600260.

4 Xiaojun Ma and Fabrizio Lombardi. Synthesis of tile sets for dna self-assembly. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(5):963–967, 2008. doi:10.1109/tcad.2008.917973.

5 Sung Ha Park, Robert Barish, Hanying Li, John H Reif, Gleb Finkelstein, Hao Yan, and Thomas H LaBean. Three-helix bundle dna tiles self-assemble into 2d lattice or 1d templates for silver nanowires. *Nano letters*, 5(4):693–696, 2005. doi:10.1021/nl050108i.

6 Matthew J Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014. URL: https://link.springer.com/content/pdf/10.1007/s11047-013-9379-4.pdf.

7 Paul WK Rothemund. Folding dna to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006. doi:10.1038/nature04586.

8 Alessandro Troisi, Vance Wong, and Mark A Ratner. An agent-based approach for modeling molecular self-organization. *Proceedings of the National Academy of Sciences*, 102(2):255–260, 2005. doi:10.1073/pnas.0408308102.

9 Erik Winfree. Simulations of computing by self-assembly. In *Fourth International Meeting on DNA-Based Computing*. California Institute of Technology, 1998. doi:10.7907/Z9TB14X7.

10 Erik Winfree and Renat Bekbolatov. Proofreading tile sets: Error correction for algorithmic self-assembly. In *International Workshop on DNA-Based Computers*, pages 126–144. Springer, 2003. doi:10.1007/978-3-540-24628-2_13.

11 Erik Winfree, Furong Liu, Lisa A Wenzler, and Nadrian C Seeman. Design and self-assembly of two-dimensional dna crystals. *Nature*, 394(6693):539–544, 1998. doi:10.1038/28998.

12 Hao Yan, Sung Ha Park, Gleb Finkelstein, John H Reif, and Thomas H LaBean. Dna-templated self-assembly of protein arrays and highly conductive nanowires. *science*, 301(5641):1882–1884, 2003. doi:10.1126/science.1089389.


# A Appendix

## A.1 Movements

Each movement of the polyomino $p$ is defined as

$$
\begin{aligned}
\mathrm{north}(p) &= \bigcup_{u\in p}\{(\mathrm{coord}(\mathrm{pos}(u),\mathrm{N}),\mathrm{gl}(u))\},\\
\mathrm{east}(p) &= \bigcup_{u\in p}\{(\mathrm{coord}(\mathrm{pos}(u),\mathrm{E}),\mathrm{gl}(u))\},\\
\mathrm{south}(p) &= \bigcup_{u\in p}\{(\mathrm{coord}(\mathrm{pos}(u),\mathrm{S}),\mathrm{gl}(u))\},\\
\mathrm{west}(p) &= \bigcup_{u\in p}\{(\mathrm{coord}(\mathrm{pos}(u),\mathrm{W}),\mathrm{gl}(u))\},\\
\mathrm{right}(p) &= \bigcup_{u\in p}\{(-(\mathrm{pos_y}(u)-\mathrm{c_y}(p))+\mathrm{c_x}(p),\mathrm{pos_x}(u)-\mathrm{c_x}(p)+\mathrm{c_y}(p),\mathrm{gl}(u)|_{\mathrm{r}})\},\\
\mathrm{back}(p) &= \bigcup_{u\in p}\{(-(\mathrm{pos_x}(u)-\mathrm{c_x}(p))+\mathrm{c_x}(p),(\mathrm{pos_y}(u)-\mathrm{c_y}(p))+\mathrm{c_y}(p),\mathrm{gl}(u)|_{\mathrm{b}})\},\\
\mathrm{left}(p) &= \bigcup_{u\in p}\{(\mathrm{pos_y}(u)-\mathrm{c_y}(p)+\mathrm{c_x}(p),-(\mathrm{pos_x}(u)-\mathrm{c_x}(p))+\mathrm{c_y}(p),\mathrm{gl}(u)|_{\mathrm{l}})\},
\end{aligned}
$$

where $\mathrm{gl}(u)$ is the glue types g of $u=(x,y,\mathrm{g})$ and $\mathrm{g}|_{\mathrm{r}}$, $\mathrm{g}|_{\mathrm{b}}$, $\mathrm{g}|_{\mathrm{l}}$ are the glue types which can be obtained by rotating g. Namely,

$$
\mathrm{g}|_{\mathrm{r}}(d)=\begin{cases}\mathrm{g(W)} & (d=\mathrm{N})\\ \mathrm{g(N)} & (d=\mathrm{E})\\ \mathrm{g(E)} & (d=\mathrm{S})\\ \mathrm{g(S)} & (d=\mathrm{W})\end{cases},
$$

$$
\mathrm{g}|_{\mathrm{b}}(d)=\begin{cases}\mathrm{g(S)} & (d=\mathrm{N})\\ \mathrm{g(W)} & (d=\mathrm{E})\\ \mathrm{g(N)} & (d=\mathrm{S})\\ \mathrm{g(E)} & (d=\mathrm{W})\end{cases},
$$

$$\mathrm{g}|_{\mathrm{l}}(d) = \begin{cases} \mathrm{g(E)} & (d = \mathrm{N}) \\ \mathrm{g(S)} & (d = \mathrm{E}) \\ \mathrm{g(W)} & (d = \mathrm{S}) \\ \mathrm{g(N)} & (d = \mathrm{W}) \end{cases}.$$

Similarly, each movement of the agent $a$ is defined as

$$\begin{aligned}
\mathrm{north}(a) &= \bigcup_{p \in a} \{\mathrm{north}(p)\}, \\
\mathrm{east}(a) &= \bigcup_{p \in a} \{\mathrm{east}(p)\}, \\
\mathrm{south}(a) &= \bigcup_{p \in a} \{\mathrm{south}(p)\}, \\
\mathrm{west}(a) &= \bigcup_{p \in a} \{\mathrm{west}(p)\}, \\
\mathrm{right}(a) &= \bigcup_{p \in a} \{\bigcup_{u \in p} \{(-(\mathrm{pos_y}(u) - \mathrm{c_y}(p)) + \mathrm{c_x}(a), \mathrm{pos_x}(u) - \mathrm{c_x}(a) + \mathrm{c_y}(a), \mathrm{gl}(u)|_{\mathrm{r}})\}\}, \\
\mathrm{back}(a) &= \bigcup_{p \in a} \{\bigcup_{u \in p} \{(-(\mathrm{pos_x}(u) - \mathrm{c_x}(a)) + \mathrm{c_x}(a), -(\mathrm{pos_y}(u) - \mathrm{c_y}(a)) + \mathrm{c_y}(a), \mathrm{gl}(u)|_{\mathrm{b}})\}\}, \\
\mathrm{left}(a) &= \bigcup_{p \in a} \{\bigcup_{u \in p} \{(\mathrm{pos_y}(u) - \mathrm{c_y}(a) + \mathrm{c_x}(a), -(\mathrm{pos_x}(u) - \mathrm{c_x}(a)) + \mathrm{c_y}(a), \mathrm{gl}(u)|_{\mathrm{l}})\}\},
\end{aligned}$$

where $\mathrm{c_x}(a)$ and $\mathrm{c_y}(a)$ are the center of mass of an agent $a$, which is defined as $\mathrm{c_x}(a) = \mathrm{round}(\sum_{u \in p, p \in a} \mathrm{pos_x}(u)/|a|)$ and $\mathrm{c_y}(a) = \mathrm{round}(\sum_{u \in p, p \in a} \mathrm{pos_y}(u)/|a|)$.

## A.2 Glue type assignment

Given a naive set of polyominoes $S$, the assignment of glue types satisfies the conditions;

$$\begin{aligned}
&\forall p_1, p_2 \in S, \forall u_1 \in p_1, \forall u_2 \in p_2, \forall d \in \mathrm{D}, && \mathrm{gl}(u_1, d) = 0 \rightarrow \\
&&& \mathrm{coord}(\mathrm{pos}(u_1), d) \neq \mathrm{pos}(u_2), \text{ and} \\
&\forall p_1, p_2 \in S, \forall u_1 \in p_1, \forall u_2 \in p_2, \forall d \in \mathrm{D}, && \mathrm{coord}(\mathrm{pos}(u), d) = \mathrm{pos}(u_2) \rightarrow \\
&&& \mathrm{gl}(u_1, d)) + \mathrm{gl}(u_2, \hat{d}) = 0 \wedge \mathrm{gl}(u_1, d)) \neq 0.
\end{aligned}$$

The first condition means that the glue type is 0 when there are no neighboring square units. The second condition guarantees that the the neighboring units have complementary glue types. Finally, the number of representative polyominoes are decreased as much as possible by assigning glue types through an ad-hoc trial and error. The assignment of glue types is applied to construct the initial and neighbor polyomino species.

# scadnano: A Browser-Based, Scriptable Tool for Designing DNA Nanostructures

**David Doty**[1]
University of California, Davis, CA, USA
https://web.cs.ucdavis.edu/~doty/
doty@ucdavis.edu

**Benjamin L Lee**
University of California, Davis, CA, USA
bnllee@ucdavis.edu

**Tristan Stérin**
Maynooth University, Ireland
https://dna.hamilton.ie/tsterin/index.html
Tristan.Sterin@mu.ie

## Abstract

We introduce *scadnano* (short for "scriptable cadnano"), a computational tool for designing synthetic DNA structures. Its design is based heavily on cadnano [24], the most widely-used software for designing DNA origami [33], with three main differences:

1. scadnano runs entirely in the browser, with *no software installation* required.
2. scadnano designs, while they can be edited manually, can also be created and edited by a *well-documented Python scripting library*, to help automate tedious tasks.
3. The scadnano file format is *easily human-readable*. This goal is closely aligned with the scripting library, intended to be helpful when debugging scripts or interfacing with other software. The format is also somewhat more *expressive* than that of cadnano, able to describe a broader range of DNA structures than just DNA origami.

**2012 ACM Subject Classification** Applied computing → Computer-aided design

**Keywords and phrases** computer-aided design, structural DNA nanotechnology, DNA origami

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.9

**Supplementary Material** *stable/dev apps:* https://scadnano.org, https://scadnano.org/dev
*repositories:* https://github.com/UC-Davis-molecular-computing/scadnano
https://github.com/UC-Davis-molecular-computing/scadnano-python-package
*Python library API:* https://scadnano-python-package.readthedocs.io
*tutorials:* https://github.com/UC-Davis-molecular-computing/scadnano-python-package/blob/master/tutorial/tutorial.md, https://github.com/UC-Davis-molecular-computing/scadnano/blob/master/tutorial/tutorial.md

**Funding** *David Doty*: Supported by NSF grants 1619343, 1900931, and CAREER grant 1844976.
*Benjamin L Lee*: Supported by REU supplement through NSF CAREER grant 1844976.
*Tristan Stérin*: Supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 772766, Active-DNA project), and Science Foundation Ireland (SFI) under Grant number 18/ERCS/5746.

**Acknowledgements** We thank Matthew Patitz for beta-testing and feedback, and Pierre-Étienne Meunier, author of codenano, for valuable discussions regarding the data model/file format. We are grateful to anonymous reviewers whose detailed feedback has increased the presentation quality.

[1] Corresponding author

# 1 Introduction

## 1.1 DNA origami and cadnano

Since its inception almost 15 years ago, DNA origami [33] has stood as the most reliable, high-yield, and low-cost method for synthesizing uniquely addressed DNA nanostructures, on the order of 100 nm wide, with $\approx 6$ nm addressing resolution (i.e., that's how far apart individual strands are).[2] To create the original designs, Rothemund wrote custom Matlab scripts to generate and visualize the designs (with ASCII art). Soon after, the software cadnano was developed by Douglas et al. [24], as part of a project extending the original 2D DNA origami results to 3D structures [23]. cadnano has become a standard tool in structural DNA nanotechnology, used for describing most major DNA origami designs.

## 1.2 scadnano

The scadnano graphical interface is shown in Figure 1; it mimics that of cadnano.

The goal of scadnano is to aid in designing large-scale DNA nanostructures, such as DNA origami, with ability to edit structures either manually, or programmatically through a scripting library. scadnano seeks to imitate most of the features of cadnano, with three major differences that enhance the usability and interoperability of scadnano:

1. scadnano runs entirely in the browser, with *no software installation* required. It aims, above all else, to be simple and easy to use, well-suited for teaching, for example.
2. scadnano designs, while they can be edited manually, can also be created and edited by a *well-documented Python scripting library*, to help automate tedious tasks.[3]
3. The scadnano file format is *easily human-readable* and expressive, natural for describing a broader range of DNA structures than just DNA origami. This goal is closely aligned with the scripting library, useful when debugging scripts or interfacing with other software. A related project, codenano [5], uses essentially the same file format, developed simultaneously in consultation with the main author of codenano.

The major features of scadnano are described in more detail in Section 3. Designed with interoperability in mind, any cadnano design can be imported into scadnano, and scadnano designs obeying certain constraints (see Section 2.3) can be exported to cadnano.

## 1.3 Related work

cadnano is the most related prior work, and its design was the inspiration for scadnano. Section 3.1 goes into detail about features that scadnano shares in common with cadnano, and the rest of Section 3 discusses some extra features in scadnano. codenano is close in purpose to scadnano [5], being also browser-based and scriptable. Unlike scadnano, codenano includes 3D visualisation components but not graphical editing.

---

[2] The basic idea of DNA origami is to use a long *scaffold* strand (either synthesized or natural; the most common choice is the natural circular single-stranded virus known as M13mp18, 7249 bases long), and to synthesize shorter (a few dozen bases long) *staple* strands designed to bind to multiple regions of the scaffold. Upon mixing in standard DNA self-assembly buffer conditions (e.g., 10 mM Tris, 1 mM EDTA, pH 8.0, 12.5 mM $MgCl_2$), with staples "significantly" more concentrated than the scaffold (typical concentrations are 1 nM scaffold and 10 nM each staple), and annealing from 90°C to 20°C for one hour, the staples bind to the scaffold and fold it into the desired shape, while excess staples remain free in solution and are easily separated from the formed structures by standard purification techniques.

[3] cadnano v2.5 has a Python scripting library, but its documentation is incomplete [3], and cadnano v2.5 has not been updated for two years [2] at the time of this writing.

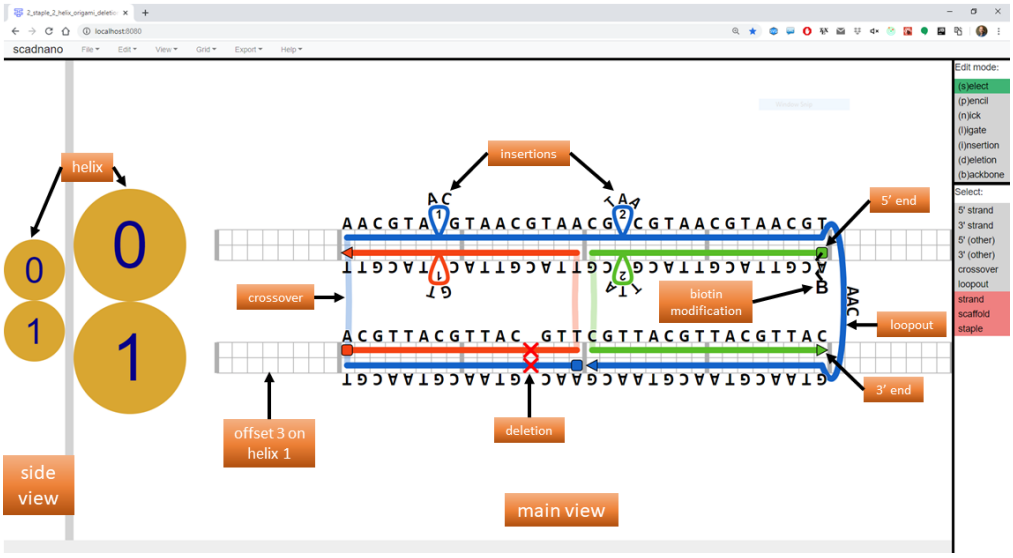



**Figure 1** screenshot of scadnano, annotated with some labels (in orange rectangles) to point out various parts of the data model.[4] The center part is the *main view*, which shows the $x$ and $y$ coordinates; most editing takes place here. On the left is the *side view*, which shows the $z$ and $y$ coordinates. $y$ increases going *down* in both views (so-called "screen coordinates), $x$ increases going *right* in the main view and going *into* the screen in the side view. $z$ increases going *right* in the side view and going *out of* the screen in the main view. The Edit modes on the right change what sorts of edits are possible, and the Select modes change what sort of objects can be selected while in the "select" edit mode.

vHelix [18] offers comprehensive 3D origami editing and visualisation features but relies on Autodesk Maya. Adenita [21] is a design and visualisation tool that allows one to work with various DNA nanostructures: standard parallel-helix DNA origami, wireframe origamis [28], and tile-based designs. Adenita is distributed within the SAMSON [17] molecular modeling platform. Specific to the domain of 2D and 3D wireframe origamis, ATHENA [28] provides both an editing interface and sequence design algorithms that generate staple sequences from a 2D sketch. Not related to graphical or script-based DNA design editing, the following software provides structural prediction tools for various features of DNA designs: CanDo [4] (finite elements-based 3D structure prediction), NUPACK and ViennaRNA [30, 43] (thermodynamic energy of DNA strands), oxDNA [38] (kinetics prediction by molecular dynamics simulation), and MrDNA [31] (3D structure and kinetics prediction).

## 1.4 Paper outline

Section 2 describes the data model used by scadnano to represent a DNA design, and its closely related storage file format, including a comparison with cadnano's file format. Section 3 describes several features of scadnano, including some that are absent from cadnano. Section 4 explains the software architecture of scadnano. Section 4 is not necessary to understand how to use scadnano, but it helps to justify why scadnano may be simpler to maintain and enhance in the future. Section 5 discusses possible future features.

This paper is not a self-contained document describing scadnano in full. See the supplementary material links for online documentation, tutorials, and the Python library API.

[4] This design is intended merely to show some scadnano features, not to show proper design respecting DNA crossover geometry; it would be strained if actually assembled.

# 2 Data model and file format

## 2.1 scadnano data model

Although scadnano and its data model are natural for describing DNA origami, it can be used to describe any DNA nanostructure composed of several DNA strands. Like cadnano, scadnano is especially well-suited to structures where all DNA helices are parallel, which includes not only origami, but also certain tile-based designs (e.g., [39, 40, 42]), or "criss-cross slat" assembly [32]. The basic concepts, explained in more detail below, are that the design is composed of several *strands*, which are bound to each other on some domains, and possibly single-stranded on others, and double-stranded portions of DNA occupy a *helix*.

### DNA Design

An example DNA design is shown in Figure 1, showing most of the features discussed here. A design (the type of object stored in a `.sc` file produced when clicking "Save" in scadnano) consists of a `grid` type (a.k.a., *lattice*, one of the following types: `square`, `honeycomb`, `hex`, or `none`, explained below), a list of `helices`, and a list of `strands`. The order of strands in the list generally doesn't matter, although it influences which are drawn on top, so a strand later in the list will have its crossovers drawn over the top of earlier strands.

### Helices

Unlike strands, the order of the helices matters; if there are $h$ helices, the helices are numbered 0 through $h - 1$. This can be overridden by specifying a field called `idx` in each helix, but the default is to number them consecutively. Each helix defines a set of integer *offsets* with a minimum and maximum; in the example above, the minimum and maximum for each helix are 0 and 48, respectively, so 48 total offsets are shown. Each offset is a position where a DNA base of a strand can go.

Helices in a grid (meaning one of square, honeycomb, or hex) have a 2D integer `grid_position` depicted in the side view (see Figure 3). Helices without a grid (meaning grid type none) have a `position`, a 3D real vector describing their `x`, `y`, `z` coordinates. Each Helix also has fields to describe angular orientation, using the "aircraft principle axes" `pitch`, `roll`, and `yaw` (default 0), although this feature is currently not well-supported (`https://github.com/UC-Davis-molecular-computing/scadnano/issues/39`). The coordinates of helices in the main view depends on `grid_position` if a grid is used, and on `position` otherwise. (Each grid position is essentially interpreted as a position with `z` = `pitch` = `roll` = `yaw` = 0.) Helices are listed from top to bottom in the order they appear in the sequence, unless the property `helices_view_order` is specified in the design to display them in a different order, though currently this can only be done in the scripting library.

Helix.`roll` describes the DNA backbone rotation about the long axis of the helix. At the offset `Helix.min_offset`, the backbone of the forward strand on that helix has angle `Helix.roll`, where we define 0 degrees to point to straight *up* in the side view. Rotation is *clockwise* as the rotation increases from 0 up to 360 degrees. This feature is not intended as a globally predictive model of stability. Rather, it helps visualize backbone angles, to place crossovers that minimize strain, by ensuring crossovers are "locally consistent", without enforcing a global notion of absolute backbone rotation on all offsets in the system.

### Strands and domains

Each strand is defined primarily by an ordered list of `domains`. Each domain is either a single-stranded *loopout* not associated to any helix, or it is a *bound domain*: a region of the strand that is contiguous on a single helix. The phrase is a bit misleading, since a bound domain is not necessarily bound to another strand, but the intention is for most of them to be bound, and for single-stranded regions usually to be represented by loopouts.

Each bound domain is specified by four mandatory properties: `helix` (indicating the index of the helix on which the domain resides), `forward` (a direction can be forward or reverse, indicated by whether this field is true or false), `start` integer offset, and a larger `end` integer offset. As with common string/list indexing in programming languages, `start` is inclusive but `end` is exclusive. So for example, a bound domain with `end`=8 is adjacent to one with `start`=8. In the main view, `forward` bound domains are depicted on the top half of the helix, and *reverse* (those with `forward`=false) are on the bottom half. If a bound domain is forward, then `start` is the offset of its 5' end, and `end`−1 is the offset of its 3' end, otherwise these roles are reversed. There is implicitly a crossover between adjacent bound domains in a strand. Loopouts are explicitly specified as a (non-bound) domain in between two bound domains. Currently, two loopouts cannot be consecutive (and this will remain a requirement), and a loopout cannot be the first or last domain of a strand (this may be relaxed in the future).

Bound domains may have optional fields, notably `deletions` (called *skips* in cadnano) and `insertions` (called *loops* in cadnano). They are a visual trick used to allow bound domains to appear to be one length in the main view of scadnano, while actually having a different length. Normally, each offset represents a single base. If instead a deletion appears at that offset, then it does not correspond to any DNA base. If an insertion appears at that offset, it has a positive integer `length`: the number of bases represented by that offset is `length`+1.

### Strand optional fields

Each strand also has a `color` and a Boolean field `is_scaffold`. DNA origami designs have at least one strand that is a scaffold (but can have more), and a non-DNA-origami design is simply one in which every strand has `is_scaffold` = false. Unlike cadnano, a scaffold strand can have either direction on any helix. When there is at least one scaffold, all non-scaffold strands are called *staples*. The general idea behind DNA origami is that all binding is between scaffolds and staples, never scaffold-scaffold or staple-staple. However, this convention is not enforced by scadnano; there are legitimate reasons for non-scaffold strands to bind to each other (e.g., DNA walkers [26] or circuits [20] on the surface of an origami).

A strand can have an optional DNA `sequence`. Of course, since the whole point of this software is to help design DNA structures, at some point a DNA sequence should be assigned to some of the strands. However, it is often best to mostly finalize the design before assigning a DNA sequence, which is why the field is optional. Many of the operations attempt to keep things consistent when modifying a design where some strands already have DNA sequences assigned, but in some cases it's not clear what to do. (e.g., what DNA sequence results when a length-5 strand with sequence AACGT is extended to be longer?)

### DNA modifications

DNA modifications describe ways that various small molecules may be attached to synthetic DNA as part of the DNA synthesis process. Common DNA modifications include biotin (useful for binding to the protein streptavidin) and fluorophores such as Cy3 (useful for light microscopy). Modifications can be attached to the 5' end, the 3' end, or to an internal base.

A few pre-defined modifications are provided as examples in the Python scripting library. However, it is straightforward to implement a custom modification. For example, useful fields of a modification are `display_text`, which is displayed in the web interface (e.g., `B` for biotin; see Figure 1), and `idt_text`, the IDT code for the modification, used for exporting DNA sequences (e.g., `"/5Biosg/ACGT"`, which attaches a 5' biotin to the sequence `ACGT`).

Because it is common to attach one type of modification to several strands in a DNA design, modifications are defined at the top level of a DNA design, where they are given a string id, referenced on each strand that contains the modification.

## 2.2 scadnano file format

The following scadnano `.sc` file encodes the design in Figure 1 in a format called JSON, a commonly-used plain text format for describing structured data [9], with support in many programming language standard libraries. The format is not exhaustively described here, but the example shows how the JSON data maps to the data model described above.

```
{
  "grid": "square",
  "helices": [
    {"max_offset": 48, "grid_position": [0, 0]},
    {"max_offset": 48, "grid_position": [0, 1]}
  ],
  "modifications_in_design": {
    "/5Biosg/": {
      "display_text": "B",
      "idt_text": "/5Biosg/",
      "location": "5'"
    }
  },
  "strands": [
    {
      "color": "#0066cc",
      "sequence": "
          AACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACG",
      "domains": [
        {"helix": 1, "forward": false, "start": 8, "end": 24, "deletions": [20]},
        {"helix":0, "forward":true, "start":8, "end":40, "insertions":[[14,1],[26,2]]},
        {"loopout": 3},
        {"helix": 1, "forward": false, "start": 24, "end": 40}
      ],
      "is_scaffold": true
    },
    {
      "color": "#f74308",
      "sequence": "ACGTTACGTTACGTTTTACGTTACGTTACGTT",
      "domains": [
        {"helix": 1, "forward": true, "start": 8, "end": 24, "deletions": [20]},
        {"helix": 0, "forward": false, "start": 8, "end": 24, "insertions": [[14, 1]]}
      ]
    },
    {
      "color": "#57bb00",
      "sequence": "ACGTTACGTTACGTTACGCGTTACGTTACGTTAC",
      "domains": [
        {"helix": 0, "forward": false, "start": 24, "end": 40, "insertions":[[26,2]]},
        {"helix": 1, "forward": true, "start": 24, "end": 40}
      ],
      "5prime_modification": "/5Biosg/"
    }
  ]
}
```

## 2.3 Comparison to cadnano file format

The file format used by cadnano v2 is a grid of dimension (number of helices)×(maximum offset) describing at each position whether a domain is present and the direction in which it is going. Additional information about *insertions* and *deletions* is given in a similar way.

An important goal of scadnano is to ensure interoperability with cadnano (see Section 3.9). Thus every cadnano design can be imported into scadnano. However, the converse is not true; scadnano's data model can describe features not present in cadnano.

1. cadnano does not have a way to encode loopouts, modifications, or gridless designs.
2. cadnano does not store DNA sequences in its file format.
3. cadnano has the constraint that helices with even index have the scaffold going forward and helices with odd index have the scaffold going backward. scadnano designs not following that convention cannot be encoded in cadnano.
4. cadnano does not explicitly encode the grid type, instead inferring it from the maximum helix offset: multiples of 21 represent the honeycomb grid, while multiples of 32 represent the square grid. To encode a scadnano design in cadnano's convention, each helix's maximum offset is modified to the lowest multiple of 21 or 32 fitting the design.

Converting a scadnano design to cadnano v2 is straightforward: lay out all domains of all strands in a (number of helices)×(modified maximum offset) grid. Maximum offsets have to be modified because of Item 4. However, converting a cadnano design to scadnano format is a bit more involved, requiring a connected components detection algorithm performed on the grid – similar to a depth-first search – in order to identify strands and their domains.

# 3 Features

## 3.1 Features shared with cadnano v2

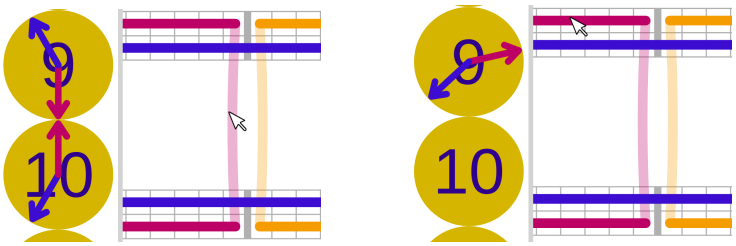
The web interface of scadnano is similar to cadnano (see Figure 1). Like cadnano, scadnano is optimal for structures consisting of parallel helices. On the left, the side view shows a cross-sectional view of the lattice where helices can be added to the design. The main view shows what the helix would look like going from left to right in the screen. Moving to the right in the main view is like moving "into the screen" in the side view.

DNA designs are drawn as they are often drawn in figures, with strands on a double-helix represented as straight lines that are connected to other helices by crossovers. Users can also add deletions and insertions (called *skips* and *loops* in cadnano) which means a strand has fewer or more bases than the interface's visually depicted length. Insertions and deletions help to use a regular spacing pattern – note the "major tick marks" every 8 bases on the helix – while allowing short regions to deviate and use more or fewer than the typical number of bases between two major tick marks. One feature scadnano adds to cadnano is the ability to customize the major tick marks, including non-regular spacing, e.g, alternating 10, 11, 10, 11 for single-stranded tiles [39, 42].

scadnano includes several "Edit modes", many similar to those of cadnano, shown in the top right corner of Figure 1. There are two main modes for editing, select mode and pencil mode, as well as several others explained in more detail in the scadnano documentation. Select mode allows users to select, resize, and delete items, just like in cadnano. (scadnano additionally allows users to copy and paste or move items; see Section 3.2). Pencil mode is used to create new objects such as helices, strands, or crossovers.

Users can assign DNA sequences to strands, and the complementary sequences for the bound strands are automatically computed. The common M13 DNA sequence is provided as a default for single-scaffold designs.

Although scadnano currently provides no 3D visualization, it does provide a primitive way to visualize the DNA backbone angles to help pick where to place crossovers; see Figure 2. This feature is slightly more flexible than the analogous feature in cadnano in that the user



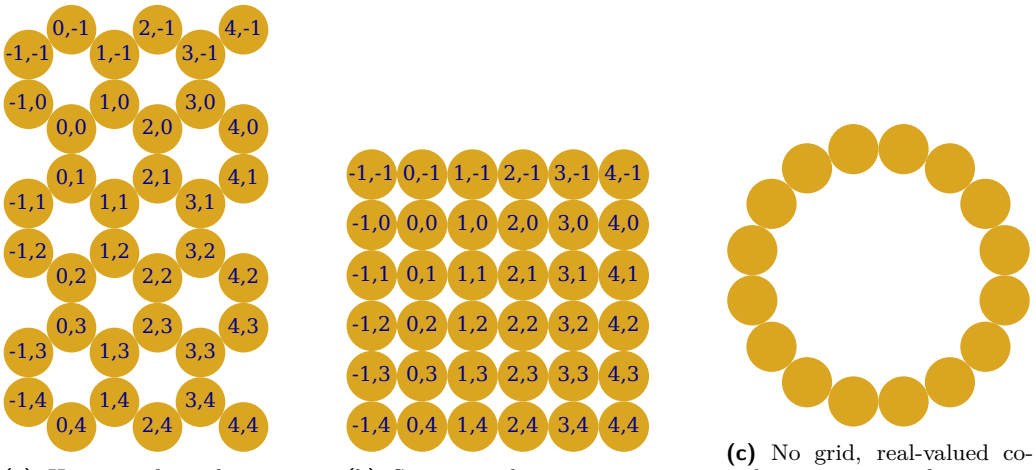

**(a)** Backbone angles at a crossover.

**(b)** Backbone angle 3 bases to the left.

**Figure 2** The side view displays the backbone angles to aid with crossover placement.

is allow to set the backbone angle at one base position to see what that implies about the backbone angle at other (typically nearby) base positions. For example, a user can "unstrain" the backbone at a crossover so that the backbone angles are perfectly aligned (see Figure 2a). The backbone angles at other positions are automatically computed (see Figure 2b).

The side and main view designs can be exported as SVG figures, and DNA sequences can be be exported into a CSV file, as well as formats recognized by the synthesis company IDT.




**(a)** Honeycomb grid, integer coordinates.

**(b)** Square grid, integer coordinates.

**(c)** No grid, real-valued coordinates in units of nanometers (coordinates not shown).

**Figure 3** scadnano grids (hex grid not shown).

Like cadnano, helices can be placed in a square or honeycomb lattice, as shown in Figure 3a and Figure 3b. scadnano provides two more grids not available on cadnano: the hex grid (allowing helices in the "holes" of the honeycomb grid) and no grid; see Section 3.8.

The remainder of Section 3 describes features not shared with cadnano v2.

## 3.2 Copy and paste

A full DNA origami design using a standard 7249-base M13mp18 scaffold uses $\approx$ 200 staples, which are tedious to create manually. In scadnano, this process is accelerated by the

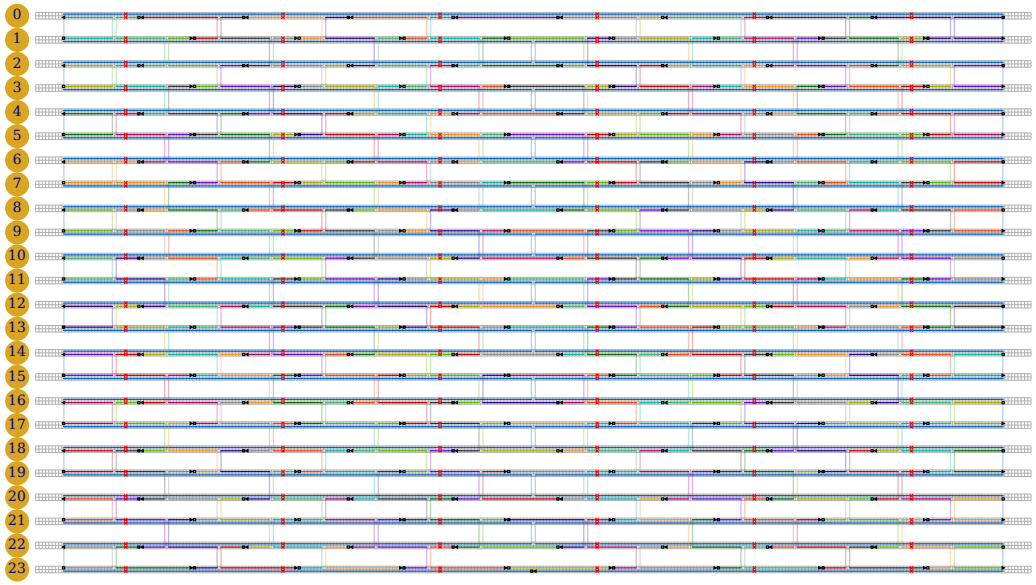



**Figure 4** A standard 24 helix DNA origami rectangle design, with "twist-correction" [41].

copy/paste feature.[5] For instance, to create a vertical "column" of 24 staples in a 24-helix rectangle (see Figure 4), one would create 2 types of staples (plus some special cases near the top/bottom), copy/paste them to make 4, copy/paste those to make 8, then copy/paste the group of 8 two more times for a total of 24 staples. Since most of the design consists of horizontally translated copies of this column, it can be created quickly by copying and pasting the column.

## 3.3 Scripting library

The scadnano Python module allows one to write scripts for creating and editing scadnano designs. (Note that cadnano v2.5, unlike v2, does have a scripting library [2], though with incomplete documentation.) The module helps automate some of the tedious tasks involved in creating DNA designs, as well as making large-scale changes to them that are easier to describe programmatically than to do by hand in scadnano.

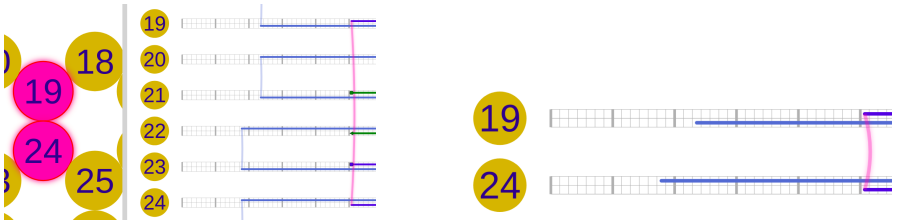
For example, the following is Python code generating the design in Figure 4, creating a `.sc` file with the design and a Microsoft Excel file with staple strand DNA sequences in a format ready to order from the DNA synthesis company IDT. It is perhaps unnecessary to read the code in detail; we provide it to demonstrate that "production-ready" designs can be created with relatively short and simple scripts. It follows the pattern described in the online tutorial (see first page).

[5] cadnano provides features to make large designs quickly, *autostaple* and *autobreak*, which are faster than copy/pasting strands, though they give less control over the outcome.

```
import scadnano as sc

def create_design():
    design = create_design_with_precursor_scaffolds()
    add_scaffold_nicks(design)
    add_scaffold_crossovers(design)
    scaffold = design.strands[0]
    scaffold.set_scaffold()
    add_precursor_staples(design)
    add_staple_nicks(design)
    add_staple_crossovers(design)
    add_twist_correcting_deletions(design)
    design.assign_m13_to_scaffold()
    return design

def create_design_with_precursor_scaffolds() -> sc.DNADesign:
    helices = [sc.Helix(max_offset=304) for _ in range(24)]
    scaffolds = [sc.Strand([sc.Domain(helix=helix, forward=helix%2 == 0, start=8, end
    =296)])
                 for helix in range(24)]
    return DNADesign(helices=helices, strands=scaffolds, grid=square)

def add_scaffold_nicks(design: sc.DNADesign):
    for helix in range(1, 24):
        design.add_nick(helix=helix, offset=152, forward=helix%2 == 0)

def add_scaffold_crossovers(design: sc.DNADesign):
    crossovers = []
    for helix in range(1, 23, 2): # scaffold interior
        crossovers.append(
            sc.Crossover(helix1=helix, helix2=helix+1, offset1=152, forward1=False))
    for helix in range(0, 23, 2): # scaffold edges
        crossovers.append(
            sc.Crossover(helix1=helix, helix2=helix+1, offset1=8, forward1=True, half=
    True))
        crossovers.append(
            sc.Crossover(helix1=helix, helix2=helix+1, offset1=295, forward1=True,half=
    True))
    design.add_crossovers(crossovers)

def add_precursor_staples(design: sc.DNADesign):
    staples = [sc.Strand([sc.Domain(helix=helix, forward=helix%2 == 1, start=8, end
    =296)])
               for helix in range(24)]
    for staple in staples:
        design.add_strand(staple)

def add_staple_nicks(design: sc.DNADesign):
    for helix in range(24):
        start_offset = 32 if helix % 2 == 0 else 48
        for offset in range(start_offset, 280, 32):
            design.add_nick(helix, offset, forward=helix%2 == 1)

def add_staple_crossovers(design: sc.DNADesign):
    for helix in range(23):
        start_offset = 24 if helix % 2 == 0 else 40
        for offset in range(start_offset, 296, 32):
            if offset != 152:  # skip crossover near seam
                design.add_full_crossover(helix1=helix, helix2=helix + 1,
                                          offset1=offset, forward1=helix % 2 == 1)

def add_twist_correcting_deletions(design: sc.DNADesign):
    for helix in range(24):
        for offset in range(27, 294, 48):
            design.add_deletion(helix, offset)

def export_idt_plate_file(design: sc.DNADesign):
    for strand in design.strands:
        if strand != design.scaffold:
            strand.set_default_idt(use_default_idt=True)
    design.write_idt_plate_excel_file(use_default_plates=True)

if __name__ == "__main__":
    design = create_design()
    export_idt_plate_file(design)
    design.write_scadnano_file()
```

## 3.4 Hiding helices to aid 3D design

The 2D main view in scadnano distorts the relative positions of the helices if they do not form a flat 2D shape as in Figure 4. For example, consider Figure 5. Helices 19 and 24, though adjacent (see side view), appear far apart in the main view. Thus crossovers between these helices, while appearing to stretch over a long distance (Figure 5a), are the same length as any other crossover (just a single phosphate group between two DNA bases).




**(a)** Without helix-hiding. **(b)** With helix-hiding.

**Figure 5** Two helices in a design, 19 and 24, are adjacent in the side view (i.e., in the actual 3D structure) but not in the main view. The selected crossover appears "long-range" in Figure 5a, but "short-range" in Figure 5b.

This can make it difficult to analyze and edit 3D designs. For example, consider the squarenut design from the original 3D origami paper [23] (see Figure 6a). This design is difficult to visualize because the 2D view is not representative of the 3D positions of the actual DNA helices, in no small part because of the "cobweb" of crossovers that results.

To aid in visualization, scadnano can display only selected helices (see Figure 6b). Helix 19 and 24 in Figure 5b can be seen in the side view are actually adjacent in 3D space. When other helices are hidden, helices 19 and 24 are displayed adjacently in the main view.

cadnano puts all helices immediately adjacent to each other in the order they are displayed in the main view. scadnano uses the distance between helices (as determined by their grid position or gridless 3D position) to determine distances. Helices are displayed in order of their index field `idx` (unless `helices_view_order` is specified to alter this order), but two helices adjacent in this order will have a vertical distance between them in the main view proportional to the distance as determined by the grid position or gridless 3D position.

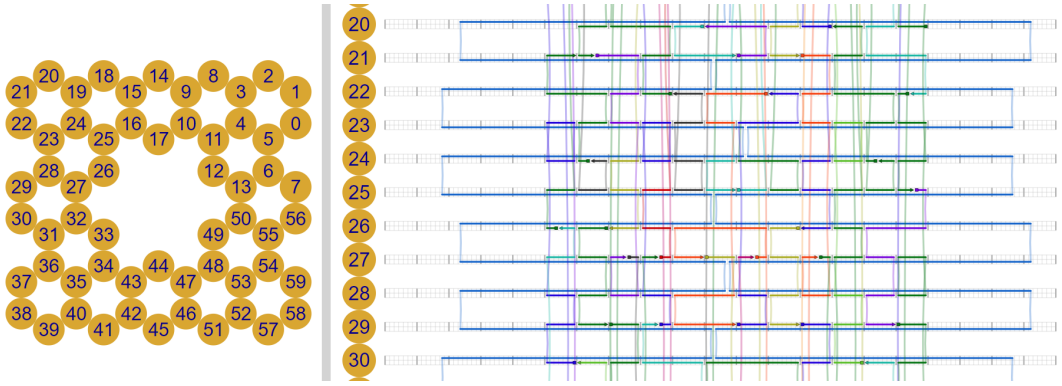
## 3.5 Single-stranded loopouts

scadnano allows a type of single-stranded domain not associated to any helix, called a *loopout*, used to describe common single-stranded features such as hairpins. In cadnano users would need to make a "fake" helix if they want to add a single-stranded DNA. For some designs, this creates awkward artifacts such as long-range crossovers to reach the fake helix.

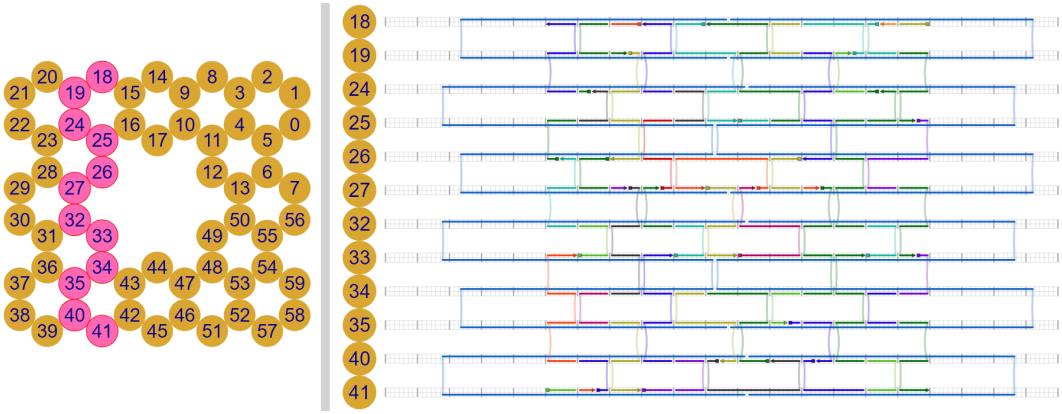
## 3.6 DNA modifications

scadnano supports for DNA modifications, such as biotin or Cy3 [8]. Figure 7a shows an example of biotin modifications to the 5' end of some staples in a 16-helix DNA origami. Users can specify a string such as `"0"` to represent the modification in the web interface.

The aspect ratio is proper for 2D origami with helices all stacked in the square lattice, helping to place modifications and visualize their relative positions to scale. Compare the scadnano display in Figure 7a to the AFM image in Figure 7b. Currently, only a few pre-loaded modifications are provided, but users can describe custom modifications.



**(a)** All helices shown, causing the dreaded *crossover cobweb*, like laser beams guarding priceless art.



**(b)** Restricted subset of helices displayed: only relevant helices and crossovers are shown.

**Figure 6** Squarenut 3D origami [23], a typical 3D origami difficult to visualize in a 2D projection.

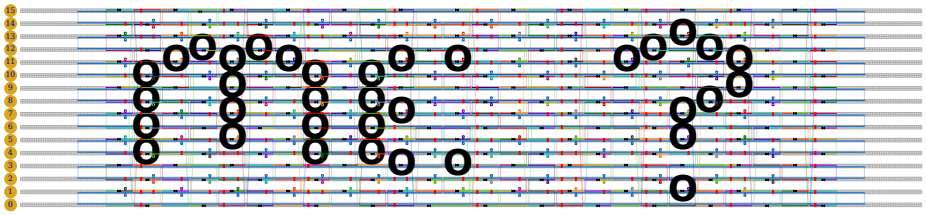
## 3.7 Unused fields

In order to maximize interoperability with other tools, scadnano allows arbitrary fields to be included in a scadnano `.sc` file. Any fields that it does not recognize are simply ignored. However, they are stored and written back out when the file is saved. Thus, "light" editing of scadnano files is possible that will preserve fields used by other programs. For example, codenano [5] allows an optional field `label` on each strand, which will be preserved for each strand by scadnano while editing other aspects of the design.

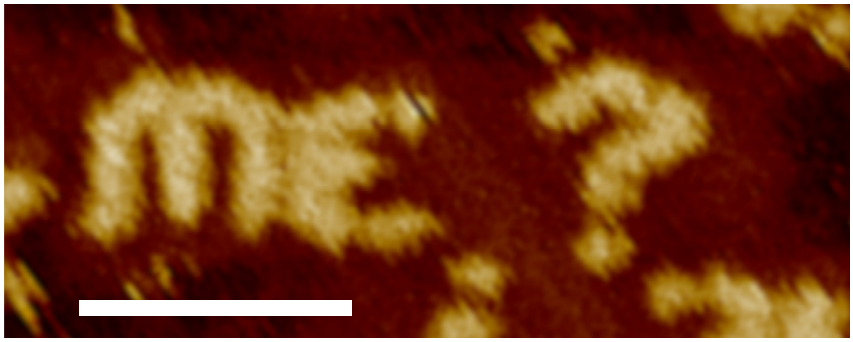
## 3.8 Gridless helix placement

scadnano includes the option to use no grid; see Figure 3c. This allows more flexible helix placement, where helix centers can be placed at any real-valued (i.e., floating-point) $(z, y)$ coordinate. This feature is useful for some designs that do not align nicely with the standard square or honeycomb lattice. In the absence of a grid, coordinates of helices are specified in nanometers. By default, the distance between each DNA helix center is 3 nm.[6]

[6] The accepted measurement of the DNA double-helix diameter is $\approx 2$ nm. However, AFM images show that in 2D square-lattice DNA origami designs, an origami with $n$ helices will have height in nanometers of approximately $3 \cdot n$ due to electrostatic repulsion between neighboring helices.



**(a)** biotin DNA modifications on the 5' end of some staples, displayed in scadnano.



**(b)** The same design imaged with atomic force microscopy (AFM), with streptavidin added to visualize the biotin locations. (scale bar: 50 nm) (image source: `https://web.cs.ucdavis.edu/~doty/papers/#proposal`)

**Figure 7** An example of a design containing biotin modifications.

## 3.9 Interoperability with cadnano

Interoperability with cadnano (version 2) is an important goal of the project. Both the scadnano GUI and Python module provide functionality that allows users to import/export a design from/to cadnano. All cadnano (version 2) designs can be imported in scadnano. However, because of fundamental differences between the way cadnano and scadnano encode designs, some scadnano designs cannot be converted cadnano (see Section 2.3).[7]

# 4 Software architecture

## 4.1 Two codebases

The codebase for scadnano is split into two pieces: the Python scripting library, and the web interface. Unfortunately, some algorithmic functionality is duplicated between them. We chose Python as the scripting language because it is easy to learn and already familiar to many physical scientists likely to use scadnano. However (despite innovations such as Pyodide [11], Skulpt [15], and Brython [1]), Python is not well-suited for *front-end* web programming, where the code is executed in the browser rather than on a server. A design goal of scadnano is to do as much work as possible in the browser.

The web interface is instead implemented using the Dart programming language [6], a modern, strongly-typed, object-oriented language that can be compiled to Javascript, the *lingua franca* of web browsers. In order to make the Python scripting library as easy to use as possible (no dependence on Dart libraries) and to keep the web interface as fast as possible

[7] These constraints are described in the documentation: `https://scadnano-python-package.readthedocs.io/en/latest/index.html#interoperability-cadnano-v2`

and avoid the need to farm out computation to a server, some algorithms (e.g., computing complementary DNA sequences of strands when they are bound to another strand that has had a DNA sequence assigned to it) are implemented in both libraries.

However, we intend for the file format to be decoupled from the scripting and web-based programs that manipulate it. Indeed, another tool called codenano [5] uses essentially the same file format as scadnano, although that program is written in Rust and has the user specify the design by writing Rust code.

## 4.2 Unidirectional data flow in graphical user interface code

Graphical user interface software, inherently asynchronous and non-sequential, is notoriously difficult to reason about. Whole classes of bugs exist that do not plague programs with only sequential logic. The open-source software community has developed many tools to aid in such design. The *model-view-controller* (MVC) architecture is almost as old as graphical interfaces themselves, dating to the 1970s [29]. However, MVC is not very well-defined, particularly the controller part, and still lends itself to common bugs.

A more recent innovation, originating within the past decade, goes under a few names, such as *model-view-update*, *the Elm architecture* [7], or *unidirectional data flow* [16]. Several variants exist implementing the idea. We chose a popular pair of technologies, React [12] and Redux [14]. They are designed for Javascript, but since Dart compiles to Javascript, they can be used with Dart with appropriate wrapping libraries [10, 13].

The cited links go into detail about the architecture; we summarize it briefly here for the curious. Briefly, all application *state* is stored in a single immutable object. (In scadnano, this includes the entire DNA design, as well as more ephemeral UI state, such as which strands are currently selected.) Immutability is a powerful concept in programming, allowing one to share an object between many concurrent processes without worrying that one process will modify it in ways unexpected by the other processes. The global state object is a tree (cycles are difficult to handle with immutable objects). The *view* (what the user sees on the screen) is specified as a deterministic function of the state. This greatly reduces the "surface area" where bugs can (and reliably do) occur: the application does not have to contain code stating how to *modify* the view in response to any possible change in the state. It merely says what the *entire* view should be, as a function of the *entire* state.

Changes to the application state are expressed using the Command pattern [25] by dispatching an *action* describing that the state should change. The application responds to the action by computing the new state as a deterministic function of the old state and the action. The view redraws itself, but optimizations ensure only the parts that depend on changed state will actually be redrawn.

This decoupling of actions that change state (and the sometimes complex logic behind them), and views that draw themselves as a function of a single state, is the key to making it straightforward to implement new features without introducing bugs. It's not foolproof; bugs do occur. There is also a nontrivial computational cost: the React library compares the old state to the new to determine which subtrees actually changed (determining which parts of the view actually need to re-render), a potentially expensive operation.

However, we find it is worth the computational cost for the benefit of reliability. We believe it will make it easier to maintain scadnano, fix bugs, and add features in the future.

Both the Python package and the Dart web interface are open-source software to which anyone can contribute. Both repositories have a CONTRIBUTING document explaining how to contribute to the projects, following the git model of making a separate branch, adding the change, and doing a pull request to merge the changes. Both repositories are currently maintained by the first author, who reviews all pull requests.

## 5 Conclusion

The goal of scadnano is to reproduce the usefulness of cadnano for designing large-scale DNA structures in a web app with a well-documented, easy-to-use scripting library. It is ready to use for designing DNA structures, although some work remains to bring it up to a more polished state. The issues page of each repository (see first page) shows many bugs and feature enhancements that have not yet been addressed.

scadnano excels where cadnano excels: in describing DNA structures where all DNA helices are in parallel. A broader range of DNA nanostructures exists, such as wireframe designs [19, 44] and curved DNA origami shapes [22, 27]. A 2D projected view can describe these, but more awkwardly than a 3D view. Since the chief goal of scadnano is to remain easy to use and responsive to bug reports and feature requests within the current scope of scadnano, it will remain for the near-term future as a tool primarily for designs that are straightforward to visualize in 2D. We outline possible future work:

**export to other file formats.** Currently, scadnano can export to the cadnano v2 file format, and it can export DNA sequences in either a comma-separated value (CSV) file, which can be processed by the user's custom scripts, or in a few formats recognized by the DNA synthesis company IDT (Integrated DNA Technologies, Coralville, IA, `https://www.idtdna.com`). It should be straightforward to export to formats recognized by other DNA synthesis companies (e.g., Bioneer), or other DNA nanotech software (e.g., oxDNA).

**helices rotated in the main view plane.** Some 2D structures do not have all helices in parallel, for example DNA origami implementations of 4-sided tiles [37], or flat origami "stiffened" by a second layer of perpendicular helices [36]. We are exploring design ideas for supporting this in a way "natural" for editing in the 2D view. In particular, copy/paste and moving of strands spanning multiple helices makes most sense for groups of helices that are parallel. One idea is to let a design specify several helix *groups*, where all helices within a group are parallel, but the groups have different rotations and translations. (For example, there would be two groups for [36] and two or four groups for [37].)

**3D visualization.** cadnano has never been ideal for visualizing arbitrary 3D structures, and neither is scadnano currently. It may remain the case that the ideal way to visualize 3D structures is to export the design to another tool specialized for the job, such as codenano [5], CanDo [4], or oxDNA [35]. However, WebGL provides a powerful platform for visualizing 3D structures, used by other software such as oxDNA and codenano. In fact, since codenano is itself implemented as a web app (written in Rust that is compiled to WebAssembly, which is itself callable from Javascript), it should be possible to implement the 3D visualization features of codenano as a library that scadnano can call.

**DNA design database.** Communication of DNA designs through the Supplementary Information of a journal remains an ad hoc method. A centralized database of DNA designs would benefit the community. We hope that the scadnano/codenano file format is sufficiently expressive to describe any such design. However, such a database need not have anything to do with the scadnano website itself.

**collaborative editing.** Collaborative editing tools such as Google Docs make use of a recently developed technique known as a *conflict-free replicated data type* (CRDT) [34]. It is conceivable that a CRDT representation of a DNA design could enable remote collaborators to simultaneously view and edit a DNA design.

## References


1 Brython. https://brython.info/.
2 cadnano v2.5. https://github.com/cadnano/cadnano2.5.
3 cadnano v2.5 Python API. https://cadnano.readthedocs.io/en/master/scripting.html.
4 Cando. https://cando-dna-origami.org/.
5 codenano. https://dna.hamilton.ie/2019-07-18-codenano.html.
6 Dart programming language. https://dart.dev/.
7 Elm programming language. https://elm-lang.org/.
8 IDT DNA modifications. https://www.idtdna.com/pages/products/custom-dna-rna/oligo-modifications.
9 Json (javascript object notation). https://www.json.org/json-en.html.
10 Overreact Dart library. https://pub.dev/packages/over_react.
11 Pyodide. https://github.com/iodide-project/pyodide.
12 React Javascript library. https://reactjs.org/.
13 Redux Dart library. https://pub.dev/packages/redux.
14 Redux Javascript library. https://redux.js.org/.
15 Skulpt. https://skulpt.org/.
16 Unidirectional data flow in Redux. https://redux.js.org/basics/data-flow.
17 SAMSON, the open molecular modeling platform. https://www.samson-connect.net, 2019.
18 Erik Benson, Abdulmelik Mohammed, Johan Gardell, Sergej Masich, Eugen Czeizler, Pekka Orponen, and Björn Högberg. DNA rendering of polyhedral meshes at the nanoscale. *Nature*, 523(7561):441–444, July 2015. doi:10.1038/nature14586.
19 Erik Benson, Abdulmelik Mohammed, Johan Gardell, Sergej Masich, Eugen Czeizler, Pekka Orponen, and Björn Högberg. DNA rendering of polyhedral meshes at the nanoscale. *Nature*, 523(7561):441–444, 2015.
20 Gourab Chatterjee, Neil Dalchau, Richard A Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature nanotechnology*, 12(9):920, 2017.
21 Elisa de Llano, Haichao Miao, Yasaman Ahmadi, Amanda J. Wilson, Morgan Beeby, Ivan Viola, and Ivan Barisic. Adenita: Interactive 3D modeling and visualization of DNA nanostructures. Technical report, bioRxiv, 2019. doi:10.1101/849976.
22 Hendrik Dietz, Shawn M Douglas, and William M Shih. Folding DNA into twisted and curved nanoscale shapes. *Science*, 325(5941):725–730, 2009.
23 Shawn M Douglas, Hendrik Dietz, Tim Liedl, Björn Högberg, Franziska Graf, and William M Shih. Self-assembly of DNA into nanoscale three-dimensional shapes. *Nature*, 459(7245):414–418, 2009.
24 Shawn M Douglas, Adam H Marblestone, Surat Teerapittayanon, Alejandro Vazquez, George M Church, and William M Shih. Rapid prototyping of 3D DNA-origami shapes with caDNAno. *Nucleic Acids Research*, 37(15):5001–5006, 2009. URL: https://cadnano.org/.
25 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Pearson Education India, 1995.
26 Hongzhou Gu, Jie Chao, Shou-Jun Xiao, and Nadrian C Seeman. A proximity-based programmable DNA nanoscale assembly line. *Nature*, 465(7295):202–205, 2010.
27 Dongran Han, Suchetan Pal, Jeanette Nangreave, Zhengtao Deng, Yan Liu, and Hao Yan. DNA origami with complex curvatures in three-dimensional space. *Science*, 332(6027):342–346, 2011.
28 Hyungmin Jun, Xiao Wang, William Bricker, Steve Jackson, and Mark Bathe. Rapid prototyping of wireframe scaffolded DNA origami using ATHENA. Technical report, bioRxiv, 2020. doi:10.1101/2020.02.09.940320.
29 Glenn Krasner and Stephen Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of object-oriented programming*, 1, 1988.

**30** Ronny Lorenz, Stephan H Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1), November 2011. doi:10.1186/1748-7188-6-26.

**31** Christopher Maffeo and Aleksei Aksimentiev. MrDNA: A multi-resolution model for predicting the structure and dynamics of nanoscale dna objects. *bioRxiv*, 2019. doi:10.1101/865733.

**32** Dionis Minev, Christopher M. Wintersinger, Anastasia Ershova, and William M Shih. Robust nucleation control via crisscross polymerization of DNA slats. Technical report, biorXiv, 2019. URL: https://www.biorxiv.org/content/10.1101/2019.12.11.873349v1.

**33** Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

**34** Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS 2011: Symposium on self-stabilizing systems*, pages 386–400, 2011.

**35** Benedict EK Snodin, Ferdinando Randisi, Majid Mosayebi, Petr Šulc, John S Schreck, Flavio Romano, Thomas E Ouldridge, Roman Tsukanov, Eyal Nir, Ard A Louis, and Jonathan P. K. Doye. Introducing improved structural properties and salt dependence into a coarse-grained model of DNA. *The Journal of chemical physics*, 142(23):234901, 2015.

**36** Anupama J Thubagere, Wei Li, Robert F Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjan Srinivas, Damien Woods, Erik Winfree, and Lulu Qian. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017.

**37** Grigory Tikhomirov, Philip Petersen, and Lulu Qian. Programmable disorder in random DNA tilings. *Nature nanotechnology*, 12(3):251, 2017.

**38** Petr Šulc, Flavio Romano, Thomas E. Ouldridge, Lorenzo Rovigatti, Jonathan P. K. Doye, and Ard A. Louis. Sequence-dependent thermodynamics of a coarse-grained DNA model. *The Journal of Chemical Physics*, 137(13):135101, 2012. doi:10.1063/1.4754132.

**39** Bryan Wei, Mingjie Dai, and Peng Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485(7400):623–626, 2012.

**40** Erik Winfree, Furong Liu, Lisa A Wenzler, and Nadrian C Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–544, 1998.

**41** Sungwook Woo and Paul WK Rothemund. Programmable molecular recognition based on the geometry of DNA nanostructures. *Nature chemistry*, 3(8):620, 2011.

**42** Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567:366–372, 2019. doi:10.1038/s41586-019-1014-9.

**43** Joseph N. Zadeh, Conrad D. Steenberg, Justin S. Bois, Brian R. Wolfe, Marshall B. Pierce, Asif R. Khan, Robert M. Dirks, and Niles A. Pierce. Nupack: Analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32(1):170–173, 2011. doi:10.1002/jcc.21596.

**44** Fei Zhang, Shuoxing Jiang, Siyu Wu, Yulin Li, Chengde Mao, Yan Liu, and Hao Yan. Complex wireframe DNA origami nanostructures with multi-arm junction vertices. *Nature nanotechnology*, 10(9):779, 2015.

# Verification and Computation in Restricted Tile Automata

**David Caballero**
Department of Computer Science, University of Texas, Rio Grande Valley, TX, USA
david.caballero01@utrgv.edu

**Timothy Gomez**
Department of Computer Science, University of Texas, Rio Grande Valley, TX, USA
timothy.gomez01@utrgv.edu

**Robert Schweller**
Department of Computer Science, University of Texas, Rio Grande Valley, TX, USA
robert.schweller@utrgv.edu

**Tim Wylie**
Department of Computer Science, University of Texas, Rio Grande Valley, TX, USA
timothy.wylie@utrgv.edu

## Abstract

Many models of self-assembly have been shown to be capable of performing computation. Tile Automata was recently introduced combining features of both Celluar Automata and the 2-Handed Model of self-assembly both capable of universal computation. In this work we study the complexity of Tile Automata utilizing features inherited from the two models mentioned above. We first present a construction for simulating Turing Machines that performs both covert and fuel efficient computation. We then explore the capabilities of limited Tile Automata systems such as 1-Dimensional systems (all assemblies are of height 1) and freezing Systems (tiles may not repeat states). Using these results we provide a connection between the problem of finding the largest uniquely producible assembly using $n$ states and the busy beaver problem for non-freezing systems and provide a freezing system capable of uniquely assembling an assembly whose length is exponential in the number of states of the system. We finish by exploring the complexity of the Unique Assembly Verification problem in Tile Automata with different limitations such as freezing and systems without the power of detachment.

**2012 ACM Subject Classification** Theory of computation → Turing machines; Computer systems organization → Molecular computing; Theory of computation → Problems, reductions and completeness

**Keywords and phrases** Tile Automata, Turing Machines, Unique Assembly Verification

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.10

**Funding** This research was supported in part by National Science Foundation Grant CCF-1817602.

## 1 Introduction

Self-assembly systems have quickly become an intense area of research due to fabrication simplicity [13], the ability to create systems at the DNA level [16], the control of nanobots [14], and the maturity of experimental techniques [12]. Self-assembly is a naturally occurring process where simple particles come together to form complex structures. These are computationally of interest since computing at the molecular level yields a lot of power.

There are several models of tile self-assembly, and they each strive to capture some aspect of self-assembling systems. A few of the better known models are the Abstract Tile Assembly Model (aTAM) [24], the 2-Handed Assembly Model (2HAM) [3], the Staged self-assembly model [10], and the Signal-passing Tile Assembly Model (STAM) [19]. There


are several other models designed to model different aspects of DNA/RNA or laboratory conditions. A recent model of tile self-assembly, called *Tile Automata* [5], was introduced as an intentional mathematical abstraction designed to implement the key features of active algorithmic self-assembly while avoiding specifics tied to any one particular implementation (using state change rules and tile attachments/detachments based on local affinities between states). By abstracting away implementation details, TA strives to serve as a proving ground for exploring the power of active algorithmic self-assembly, along with providing a central *hub* through which various disparate models of self-assembly can be related by way of comparison to TA. One recent example of this type of application includes [2] in which TA is shown capable of simulating the *Amoebots* model [8] of programmable matter.

Given the goal of TA to connect many models of self assembly, in this paper we explore the computational power of limited Tile Automata systems such as versions of TA that do not allow detachment (not possible in some models). To facilitate this, we first show how to create general Turing Machines, and then we explore the complexity of a common question within self-assembly models: the unique assembly verification problem. If given a system, can the output be guaranteed? This is a natural problem that is polynomial in some models, yet uncomputable in others.

## 1.1 Previous Work

In his Ph.D. thesis, Winfree presented the Abstract Tile Assembly model (aTAM) and showed it was capable of universal computation by simulating a Turing Machine [24], and the computational power is explored in depth in other works such as [15]. The 2-Handed Assembly Model (2HAM) [3] introduced a more powerful model and is capable of fuel efficient computation [20] along with the Signal-passing Tile Assembly Model [19] which has tiles that can interact to turn glues on or off.

In [10, 25], the authors show a connection between finding the smallest Context Free Grammar and optimization problems in the Staged Assembly model. In the staged assembly model, it was show that while only using a constant number of tile types, a system can construct length-$n$ lines using $\mathcal{O}(\log n)$ bins and mixes [9]. Repulsive forces have been shown to aid in constructing shapes at constant scale [18]. Further, by utilizing the temperature to encode information, shapes can be constructed with constant (or nearly) tile types [6, 22].

The Unique Assembly Verification problem asks if a given system uniquely produces a given assembly. In the aTAM this problem was shown to be solvable in polynomial time [1]. In the 2HAM this problem was shown to be in coNP with certain generalizations being coNP-Complete [3, 21]. In the staged assembly model, this problem is known to be $\text{coNP}^{\text{NP}}$-hard and conjectured to be PSPACE-Complete [23]. Adding the power of negative glues also vastly changes the complexity of this problem making in uncomputable in models that include it due to the ability for pieces of assemblies to break off [11]. However, adding negative glues but restricting the ability for assemblies to detach we still see an increase in difficulty with UAV in aTAM without detachment being coNP-complete [4].

The Tile Automata model was introduced in [5] merging ideas from Cellular Automata and Tile Self-Assembly. The authors showed that freezing tile automata (where a tile cannot repeat states) is capable of simulating non-freezing systems. This powerful model has also been shown to be capable of simulating models of programmable matter [2]. Cellular Automata has been shown to be Turing Complete even in 1-dimension [7].

**Table 1** Given a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_a, q_r, q_s)$, simulating Tile Automata systems are given in Theorems 3.4 and 3.5, respectively.

| Turing Machine | Tile Automata System | States | Transition Rules |
|---|---|---|---|
| Determinisic | Non-Freezing 1D | $\mathcal{O}(\|Q\|\|\Gamma\|)$ | $\mathcal{O}(\|\delta\|)$ |
| Bounded Time | Freezing 1D | $\mathcal{O}(\|Q\|\|\Gamma\|TIME(M))$ | $\mathcal{O}(\|\delta\|TIME(M)^2)$ |

**Table 2** Results for the Unique Assembly Verification in Tile Automata. **Transition Rules** describes the types of transition rules allowed in the system. In Affinity Strengthening Systems all transition rules increase affinity so no detachment may occur. **Freezing** indicates whether the system is freezing where tiles cannot repeat states. **Result 1D** is the complexity of UAV in 1 Dimension and **Result 2D** is the complexity of 2 Dimensions. **Theorem** is where these can be found. *This result is only true when cycles in the production graph are allowed. All other results are true regardless of which definition is used.

| Transition Rules | Freezing | 1D Result | 2D Result | Theorem |
|---|---|---|---|---|
| Affinity Strengthening | Freezing | coNP-hard | $\text{coNP}^{\text{NP}}$-Complete | Thms. 6.8, 6.7 |
| Affinity Strengthening | Non-freezing | PSPACE-Complete | PSPACE-Complete | Thm. 6.3 |
| General | Freezing | Open | Undecidable | Thm. 5.2* |
| General | Non-freezing | Undecidable | Undecidable | Thm. 5.1 |

## 1.2 Our Contributions

In Tile Automata, cases may occur where systems contain one terminal assembly but exhibit behavior that does not naturally seem to uniquely produce that assembly. We define unique assembly later, but note that the final requirement addresses a feature of Tile Automata and other models with detachment where there exist assemblies that are not terminal but are never part of the final assembly. Cycles in the production graph are not possible in many self-assembly models so we add this restriction. However many of our results work with or without this restriction, so we explore both cases.

In this work we explore Tile Automata systems that uniquely assemble $n$-length lines and the complexity of determining whether a system uniquely assembles a given assembly. We first present a Turing Machine simulation capable of covert and fuel-efficient computation. We use this construction to show a connection between the largest finite assembly problem and Busy Beaver Machines (Turing Machines that print a certain number of symbols using a minimum number of states). In the more restricted case of Freezing Systems we show we can construct $n$-length lines using $\mathcal{O}(n)$ states. Results are shown in Table 1.

We then explore the Unique Assembly Verification problem. An overview of the results are shown in Table 2. We show that UAV is uncomputable via Turing Machine simulation. We also extend this to 2-Dimensional freezing systems (this reduction results in a system with cycles). By removing the ability for assemblies to break apart we achieve a model closer to traditionally studied models. We restrict this by studying what we call *Affinity-Strengthening* systems where a state can never lose affinity by a transition. In this case, we show the UAV problem is PSPACE-Complete utilizing bounded-space Turing Machine simulation. When restricting the model to both Affinity Strengthening and Freezing we show membership in $\text{coNP}^{\text{NP}}$. We then provide reductions to show $\text{coNP}^{\text{NP}}$-completeness for 2-dimensional UAV and coNP-hardness in 1 dimension.

## 2 Model and Definitions

A Tile Automata system is a marriage between cellular automata and 2-handed self-assembly. Systems consist of a set of monomer tile states, along with local affinities between states denoting the strength of attraction between adjacent monomer tiles in those states. A set of local state-change rules are included for pairs of adjacent states. Assemblies (collections of edge-connected tiles) in the model are created from an initial set of starting assemblies by combining previously built assemblies given sufficient binding strength from the affinity function. Further, existing assemblies may change states of internal monomer tiles according to any applicable state change rules. An example system is shown in Figure 1.

### 2.1 States, tiles, and assemblies

**Tiles and States.** Consider an alphabet of *state types*[1] $\Sigma$. A tile $t$ is an axis-aligned unit square centered at a point $L(t) \in \mathbb{Z}^2$. Further, tiles are assigned a state type from $\Sigma$, where $S(t)$ denotes the state type for a given tile $t$. We say two tiles $t_1$ and $t_2$ are of the same *tile type* if $S(t_1) = S(t_2)$.

**Affinity Function.** An *affinity function* takes as input an element in $\Sigma^2 \times D$, where $D = \{\perp, \vdash\}$, and outputs an element in $\mathbb{N}$. This output is referred to as the *affinity strength* between two states, given direction $d \in D$. Directions $\perp$ and $\vdash$ indicate above-below and side-by-side orientations of states, respectively.

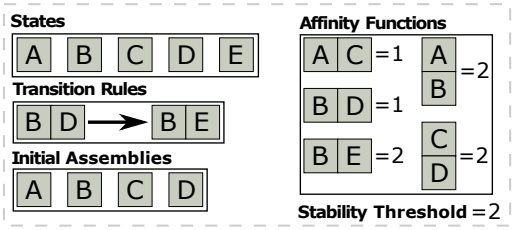
**Transition Rules.** Transition rules allow states to change based on their neighbors. A *transition rule* is a 5-tuple $(S_{1a}, S_{2a}, S_{1b}, S_{2b}, d)$ with each $S_{1a}, S_{2a}, S_{1b}, S_{2b} \in \Sigma$ and $d \in D = \{\perp, \vdash\}$. ($S_{1a}$ and $S_{1b}$ being the left state or the top state.) Essentially, a transition rule says that if states $S_{1a}$ and $S_{2a}$ are adjacent to each other, with a given orientation $d$, they can transition to states $S_{1b}$ and $S_{2b}$ respectively.

**Assemblies.** A *positioned shape* is any subset of $\mathbb{Z}^2$. A *positioned assembly* is a set of tiles at unique coordinates in $\mathbb{Z}^2$, and the positioned shape of a positioned assembly $\mathcal{A}$ is the set of coordinates of those tiles, denoted as $\texttt{SHAPE}_{\mathcal{A}}$. For a positioned assembly $\mathcal{A}$, let $\mathcal{A}(x, y)$ denote the state type of the tile with location $(x, y) \in \mathbb{Z}^2$ in $\mathcal{A}$.

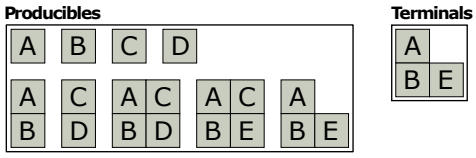
For a given positioned assembly $\mathcal{A}$ and affinity function $\Pi$, define the *bond graph* $G_{\mathcal{A}}$ to be the weighted grid graph in which:

- each tile of $\mathcal{A}$ is a vertex,
- no edge exists between non-adjacent tiles,
- the weight of an edge between adjacent tiles $T_1$ and $T_2$ with locations $(x_1, y_1)$ and $(x_2, y_2)$, respectively, is
  - $\Pi(S(T_1), S(T_2), \perp)$ if $y_1 > y_2$,
  - $\Pi(S(T_2), S(T_1), \perp)$ if $y_1 < y_2$,
  - $\Pi(S(T_1), S(T_2), \vdash)$ if $x_1 < x_2$,
  - $\Pi(S(T_2), S(T_1), \vdash)$ if $x_1 > x_2$.

[1] We note that $\Sigma$ does not include an "empty" state. In tile self-assembly, unlike cellular automata, positions in $\mathbb{Z}^2$ may have no tile (and thus no state).




**(a)** Tile Automata System Γ.




**(b)** The producibles and terminals of Γ.

**Figure 1** An example of a tile automata system Γ. Recursively applying the transition rules and affinity functions to the initial assemblies of a system yields a set of producible assemblies. Any producibles that cannot combine with, break into, or transition to another assembly are considered to be terminal.

A positioned assembly $\mathcal{A}$ is said to be *$\tau$-stable* for positive integer $\tau$ provided the bond graph $G_{\mathcal{A}}$ has min-cut at least $\tau$.

For a positioned assembly $\mathcal{A}$ and integer vector $\vec{v} = (v_1, v_2)$, let $\mathcal{A}_{\vec{v}}$ denote the positioned assembly obtained by translating each tile in $\mathcal{A}$ by vector $\vec{v}$. An *assembly* is a set of all translations $\mathcal{A}_{\vec{v}}$ of a positioned assembly $\mathcal{A}$. A *shape* is the set of all integer translations for some subset of $\mathbb{Z}^2$, and the shape of an assembly $A$ is defined to be the set of the positioned shapes of all positioned assemblies in $A$. The *size* of either an assembly or shape $X$, denoted as $|X|$, refers to the number of elements of any positioned assembly of $X$.

**Breakable Assemblies.** An assembly is *$\tau$-breakable* if it can be split into two assemblies along a cut whose total affinity strength sums to less than $\tau$. Formally, an assembly $C$ is *breakable* into assemblies $A$ and $B$ if the bond graph $G_{\mathcal{C}}$ for some positioned assembly $\mathcal{C} \in C$ has a cut $(\mathcal{A}, \mathcal{B})$ for positioned assemblies $\mathcal{A} \in A$ and $\mathcal{B} \in B$ of affinity strength less than $\tau$. We call assemblies $A$ and $B$ *pieces* of the breakable assembly $C$.

**Combinable Assemblies.** Two assemblies are *$\tau$-combinable* provided they may attach along a border whose strength sums to at least $\tau$. Formally, two assemblies $A$ and $B$ are *$\tau$-combinable* into an assembly $C$ provided $G_{\mathcal{C}}$ for any $\mathcal{C} \in C$ has a cut $(\mathcal{A}, \mathcal{B})$ of strength at least $\tau$ for some positioned assemblies $\mathcal{A} \in A$ and $\mathcal{B} \in B$. $C$ is a *combination* of $A$ and $B$.

**Transitionable Assemblies.** Consider some set of transition rules $\Delta$. An assembly $A$ is *transitionable*, with respect to $\Delta$, into assembly $B$ if and only if there exist $\mathcal{A} \in A$ and $\mathcal{B} \in B$ such that for some pair of adjacent tiles $t_i, t_j \in \mathcal{A}$:

- $\exists$ a pair of adjacent tiles $t_h, t_k \in \mathcal{B}$ with $L(t_i) = L(t_h)$ and $L(t_j) = L(t_k)$
- $\exists$ a transition rule $\delta \in \Delta$ s.t. $\delta = (S(t_i), S(t_j), S(t_h), S(t_k), \perp)$ or $\delta = (S(t_i), S(t_j), S(t_h), S(t_k), \vdash)$
- $\mathcal{A} - \{t_i, t_j\} = \mathcal{B} - \{t_h, t_k\}$

## 2.2 Tile Automata model (TA)

A *tile automata system* is a 5-tuple $(\Sigma, \Pi, \Lambda, \Delta, \tau)$ where $\Sigma$ is an alphabet of state types, $\Pi$ is an affinity function, $\Lambda$ is a set of initial assemblies with each tile assigned a state from $\Sigma$, $\Delta$ is a set of transition rules for states in $\Sigma$, and $\tau \in \mathbb{N}$ is the *stability threshold*. When the affinity function and state types are implied, let $(\Lambda, \Delta, \tau)$ denote a tile automata system. An example tile automata system can be seen in Figure 1.

▶ **Definition 2.1** (Tile Automata Producibility). *For a given tile automata system* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, \tau)$*, the set of* producible assemblies *of* $\Gamma$*, denoted* $\texttt{PROD}_\Gamma$*, is defined recursively:*

- *(Base)* $\Lambda \subseteq \texttt{PROD}_\Gamma$
- *(Recursion) Any of the following:*
  - *(Combinations) For any* $A, B \in \texttt{PROD}_\Gamma$ *such that* $A$ *and* $B$ *are* $\tau$*-combinable into* $C$*, then* $C \in \texttt{PROD}_\Gamma$*.*
  - *(Breaks) For any* $C \in \texttt{PROD}_\Gamma$ *such that* $C$ *is* $\tau$*-breakable into* $A$ *and* $B$*, then* $A, B \in \texttt{PROD}_\Gamma$*.*
  - *(Transitions) For any* $A \in \texttt{PROD}_\Gamma$ *such that* $A$ *is transitionable into* $B$ *(with respect to* $\Delta$*), then* $B \in \texttt{PROD}_\Gamma$*.*

For a system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, \tau)$, we say $A \rightarrow_1^\Gamma B$ for assemblies $A$ and $B$ if $A$ is $\tau$-combinable with some producible assembly to form $B$, if $A$ is transitionable into $B$ (with respect to $\Delta$), if $A$ is $\tau$-breakable into assembly $B$ and some other assembly, or if $A = B$. Intuitively this means that $A$ may grow into assembly $B$ through one or fewer combinations, transitions, and breaks. We define the relation $\rightarrow^\Gamma$ to be the transitive closure of $\rightarrow_1^\Gamma$, i.e., $A \rightarrow^\Gamma B$ means that $A$ may grow into $B$ through a sequence of combinations, transitions, and/or breaks.

▶ **Definition 2.2** (Production Graph). *The production graph of a Tile Automata system* $\Gamma$ *is a directed graph where each vertex corresponds to an assembly in* $\texttt{PROD}_\Gamma$ *and there exists a directed edge between assemblies* $A$ *and* $B$ *if* $A \rightarrow^\Gamma B$*.*

▶ **Definition 2.3** (Terminal Assemblies). *A producible assembly* $A$ *of a tile automata system* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, \tau)$ *is* terminal *provided* $A$ *is not* $\tau$*-combinable with any producible assembly of* $\Gamma$*,* $A$ *is not* $\tau$*-breakable, and* $A$ *is not transitionable to any producible assembly of* $\Gamma$*. Let* $\texttt{TERM}_\Gamma \subseteq \texttt{PROD}_\Gamma$ *denote the set of producible assemblies of* $\Gamma$ *which are terminal.*

▶ **Definition 2.4** (Freezing). *Consider a tile automata system* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, \tau)$ *and a directed graph* $G$ *constructed as follows:*

- *each state type* $\sigma \in \Sigma$ *is a vertex*
- *for any two state types* $\alpha, \beta \in \Sigma$*, an edge from* $\alpha$ *to* $\beta$ *exists if and only if there exists a transition rule in* $\Delta$ *s.t.* $\alpha$ *transitions to* $\beta$

$\Gamma$ *is said to be* freezing *if* $G$ *is acyclic and* non-freezing *otherwise. Intuitively, a tile automata system is freezing if any one tile in the system can never return to a state which it held previously. This implies that any given tile in the system can only undergo a finite number of state transitions.*

▶ **Definition 2.5** (Affinity Strengthening). *An Affinity-Strengthening system is a Tile Automata system where all transition rules can only increase a states affinity with all other states so no detachments ever occur. Formally a tile automata system* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, \tau)$ *is an Affinity Strengthening system if for each* $s, s' \in \Sigma$ *where* $s$ *transitions to* $s'$*,* $\Delta(s, t) \leq \Delta(s', t) \forall t \in \Sigma$*.*

▶ **Definition 2.6** (Bounded). *A tile automata system* $\Gamma$ *is* bounded *if and only if there exists a* $k \in \mathbb{Z}_{>0}$ *such that for all* $A \in \texttt{PROD}_\Gamma$*,* $|A| < k$*.*

▶ **Definition 2.7** (Unique Assembly). *A Tile Automata system* $\Gamma$ uniquely produces *an assembly* $A$ *if*

- $A$ *is the only assembly in* $\texttt{TERM}_\Gamma$
- *for all* $B \in \texttt{PROD}_\Gamma$*,* $B \rightarrow^\Gamma A$*.*
- $\Gamma$ *is bounded.*
- *there does not exist a pair of assemblies* $B, C \in \texttt{PROD}_\Gamma$*, such that* $B \rightarrow^\Gamma C \rightarrow^\Gamma B$*.*[2]

[2] When we refer to Unique Assembly allowing cycles, this requirement is omitted.

## 3 One Dimensional Turing Machine

Since Tile Automata is a generalization of 2HAM and borrows from Cellular Automata it is expected that it is as powerful as both of these models. Here we present a construction that is capable of both covert and fuel-efficient computation. We present informal definitions of each of these. For rigorous definitions, we refer the reader to [20, 19] for fuel-efficiency, and [4] for covert computation.

▶ **Definition 3.1** (Simulation). *A Tile Automata system $T$ is said to simulate a Turing Machine $M$, if for every producible assembly $a$ of $T$ can be mapped to a configuration $m$ of $M$ and any other producible assembly $b$ such that $a \rightarrow_1^\Gamma b$, $b$ either also maps to $m$ or maps to another configuration $m'$ such that $m'$ is the next step of $m$. Finally, each terminal assembly of $T$ maps to an output of $M$.*

▶ **Definition 3.2** (Covert Computation). *Given a Tile Automata system $T$ that simulates a Turing Machine $M$, $T$ covertly simulates $M$ if for each output of $M$, there exits a single terminal assembly that maps to it.*

▶ **Definition 3.3** (Fuel Efficient Computation). *A fuel efficient Turing machine simulation in Tile Automata represents the tape of a Turing machine as one assembly, and requires that each computational step of the Turing machine occurs by way of the attachment of at most a constant number of assemblies of at most constant size. Thus, the simulation of $n$ steps of a computation "uses up" at most $O(n)$ tiles worth of fuel.*

▶ **Theorem 3.4.** *For any Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_a, q_r, q_s)$, there exists a covert, fuel-efficient, 1-dimensional Tile Automata system $T = (\Sigma_{TA}, \Pi, \Lambda, \Delta)$[3] that can simulate $M$ such that $|\Sigma_{TA}| = \mathcal{O}(|Q||\Gamma|)$ and $|\Delta| = \mathcal{O}(|\delta|)$.*

**Proof.** Given a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_a, q_r, q_s)$, we construct the Tile Automata system $T = (\Sigma_{TA}, \Pi, \Lambda, \Delta)$ as follows.

**States.** Conceptually, we partition the set of states ($\Sigma_{TA}$) into three subsets for clarity: *head* states $\mathcal{H}$, *symbol* states $\mathcal{S}$, and *utility* states $\mathcal{W}$. Let $\mathcal{H} = \{h_{(q,s)} | q \in Q, s \in \Sigma\}$ and let $\mathcal{S} = \{\sigma_s | s \in \Sigma\}$ (Figure 2a). All states in $\mathcal{H}$ and $\mathcal{S}$ have affinity with all states in $\Sigma_{TA}$. There are eight states in $\mathcal{W}$: signal accept states, final accept states, signal reject states, final reject states, and four buffer states $B_L$, $B'_L$, $B_R$, and $B'_R$. The *signal accept state* has affinity with all states in $\Sigma_{TA}$, and the *final accept state* has affinity with all states other than itself and the four buffer states. The two reject states have corresponding affinity rules as those of the accept states. The buffer states ensure that no two assemblies attach during the computation. Each of the four buffer states have affinity with each state in $\mathcal{H}$ and $\mathcal{S}$. $B_L$ and $B_R$ have affinity with $B'_L$ or $B'_R$ respectively.

**Transitions.** We create a transition rule such that for each Tile Automata state $h_{(q,s)} \in \mathcal{H}$ and $\sigma_i \in \mathcal{S}$, the rule represents a step in $M$ (Figure 2b). WLOG, assume an assembly $A$ representing the a configuration of a Turing Machine $M$ has the state $h_{(q,s)}$ with states, $\sigma_L, \sigma_R \in \mathcal{S}$ to the left and right of $h_{(q,s)}$, respectively. If the head of $M$ moves right then the transition rule will take place between $h_{(q,s)}$ and $\sigma_R$. If the TM head moves left then the transition rule will be between $\sigma_L$ and $h_{(q,s)}$. $h_{(q,s)}$ will transition into the state representing

[3] 1-Dimensional Tile Automata systems always have $\tau = 1$ so we omit that parameter from $T$

the symbol that is to be written on the tape in $M$ after a state $q$ reads symbol $s$. Either $\sigma_L$ or $\sigma_R$ would then transition into the state $h_{(q',\sigma_L)}$ or $h_{(q',\sigma_R)}$ respectively where $q'$ is the new state of the head of $M$ after reading $s$ from state $q$. There also exists an additional transition rule if $\sigma_L$ or $\sigma_R$ is a buffer state. This will transition $B_L$ or $B_R$ to state $B'_L$ or $B'_R$ respectively. $B'_L/B'_R$ transitions into the symbol state representing the blank symbol when it is to attached to state $B_L/B_R$.

**Accept/Reject.** For transitions where $M$ enters the accept state, we create transition rules where both tiles enter the signal accept state. This state has transition rules with each other state transitioning that state into the signal accept state as well. If it transitions with a buffer state or the final accept state, both tiles enter the final accept state. The final accept state also transitions with every other state and both tiles become the final accept state. The reject states follow the same rules.

**Input.** We construct a Tile Automata system that runs $M$ on a string $x$. We construct the system as described and create an initial assembly $A$ that represents $x$. $A$ will have a length of $|x| + 2$. The left most state of $A$ will be $B_L$. (WLOG assume the head of $M$ starts on the left most cell.) The next state of $A$ will be $s_{(q,s)}$ where $q$ is the initial state of $M$ and $s$ is the first symbol in $x$. The next states of $A$ each represent the symbols in the string $x$ in order. The rightmost state of $A$ is $B_R$ (Figures 2c, 2d).

The buffer states $B_L$ and $B_R$ are always an initial assembly and are used to extend the tape if the head attempts to move past the right edge. First, the head state causes $B_R$ to transition to $B'_R$. With $B'_R$ on the edge of the assembly a new $B_R$ tile will attach. Once this attachment occurs $B'_R$ transitions to the symbol state representing the blank symbol on the tape. Then the head state may transition with the blank symbol if needed. The same process occurs with $B_L$ when the head attempts to move off the left end of the tape.

**Terminal Assemblies.** If $M$ accepts the input $x$, then by the rules of our system the accept states will appear in our assembly. The signal accept state will be the first to appear and will propagate to the edges of the assembly. Once the signal accept state reaches the buffer states on the edge of the assembly they will transition into the final accept states. Any final accept state that is attached to any other state will make that tile into a final accept state. Any two final accept states that are next to each other do not have affinity and will detach. After the accept state appears in an assembly the only terminal assemblies that will exist are single final accept states. The same will occur if the machine rejects.

Since there are only two possible terminal assemblies, the final accept state and the final reject state, this construction performs covert computation. This computation is also fuel efficient since the only time a new assembly is attached is when the Turing Machine writes on a blank symbol at the edge of the tape, which can only occur once per computation step. ◀

## 3.1 Freezing Systems

Here we present modifications to the construction above for freezing 1-dimensional systems to perform bounded time computation.

▶ **Theorem 3.5.** *For any bounded-time Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_a, q_r, q_s)$, there exists a covert, fuel-efficient, 1-dimensional freezing Tile Automata system $T = (\Sigma_{TA}, \Pi, \Lambda, \Delta)$ that can simulate $M$ such that. $|\Sigma_{TA}| = \mathcal{O}(|Q||\Gamma|TIME(M))$ and $|\Delta| = \mathcal{O}(|\delta|TIME(M)^2)$.*

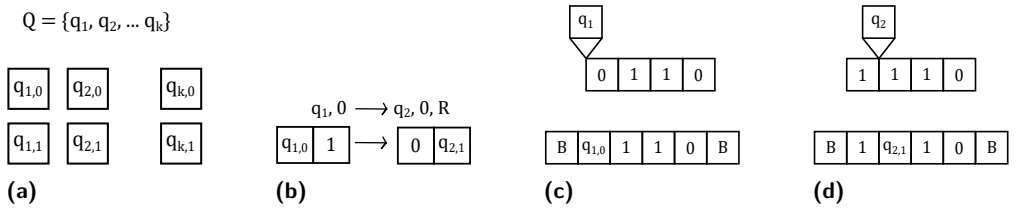



**Figure 2** (a) Tile automata states (Below) created from the states of Turing Machine (Above) over a binary alphabet. (b) State change rules (Below) created from the Turing Machine transition rules (Above). (c) A Turing Machine (Above) configuration and the representative TA assembly (Below) . (d) The same Turing Machine (Above) after making one step and the assembly (Below) after the same step.

**Proof.** We modify the construction from Theorem 3.4. We have $\Sigma_{TA}$ partitioned into three sets $\mathcal{H}$, $\mathcal{S}$, and $\mathcal{W}$. In a freezing system states can not be repeated, so for each state in $\mathcal{H}$ and $\mathcal{S}$ we create a number of states equal to the number of steps the Turing Machine $M$ can take. Each head state will not only represent the state of the Turing machine and the symbol on the tape, but it will also represent how many steps the Turing Machine has taken. Each symbol state will represent the symbol on the tape and also the last step that it was modified. The head states will have a transition rule with each symbol state regardless of the last step that symbol was modified. When a head state transitions into a symbol state it will represent the step that the transition took place.

This increase in state-space ensures no tile will ever become the same state twice. Symbol states written at step $x$ can only transition into a head state. The head state will always represent a step $y > x$. When the head state transitions back to a symbol state it will go to a symbol state written at state $y$. Since $x < y$, no tile will ever repeat states. ◀

## 4 Shapebuilding and the Largest Assembly Problem

Given a Tile Automata system with limited states, we examine how large of an assembly may be constructed. We first consider the case of one-dimensional assemblies and leverage Theorems 4.2 and 4.3 to show that the longest buildable line's length is related to the Busy Beaver function in general, and exponential in the case of freezing systems. We then consider the Largest Assembly problem, and apply Theorem 4.3 to show that this problem is uncomputable for general TA even in one-dimension.

### 4.1 General

The Busy Beaver function $BB(n)$, for any positive integer $n$, is the maximum number of symbols printable by a Turing Machine using $n$ states.[4]

▶ **Definition 4.1** (String Representation). *An assembly $A$ is said to represent a string $x$ if there exists a mapping of the states in $A$ to the symbols in $x$ such that the $n^{th}$ state of $A$ maps to the $n^{th}$ symbol of $x$ for all $0 < n \leq |x|$*

▶ **Lemma 4.2.** *For any $n$-state 2-symbol (not including the blank symbol) Turing Machine $M$ which produces an output $x$, there exists a $\mathcal{O}(n)$-state Tile Automata System $T$ which uniquely assembles an assembly $A$, such that $A$ represents $x$.*

[4] For this definition we consider Turing Machines using a binary alphabet.

**Proof.** We modify the construction from Theorem 3.4 so that once $M$ halts the head state transitions into a symbol state. The resulting assembly will be terminal since symbol states do not transition with each other. This final assembly will consist of symbol states that each represent the symbols in $x$. The number of states used by $T$ is $2n$ head states, 2 symbol states, and 4 buffer states which is bounded by $\mathcal{O}(n)$. Note there is no need for accept/reject states since the head state just turns into a symbol state when the TM halts. ◀

▶ **Theorem 4.3.** *For any positive integer $n$, there exists a 1-dimensional Tile Automata system that uniquely assembles a $BB(n)$-length line using $\mathcal{O}(n)$ states.*

**Proof.** Using Lemma 4.2 we can take any Busy Beaver Machine and create a Tile Automata system which uniquely produces an assembly the same size as the number of symbols printed on the tape. ◀

## 4.2 Freezing

For freezing Tile Automata systems, we can create systems that uniquely produce $n$-length lines and only require states that are logarithmic in the length of the line. For clarity we begin with a helping lemma.

▶ **Lemma 4.4.** *For all $n = 2^x$ for $x \in \mathbb{N}$, there exists a 1-dimensional freezing Tile Automata system that uniquely assembles an $n$ length line using $\mathcal{O}(\log n)$ states.*

**Proof.** The cases for $x = 0, 1, 2$ are trivial. A system that uniquely builds a length $2^3$ line is shown in Figure 3. The only initial states are $1_A$ and $1_B$. The affinities are between adjacent states. The transition rules are highlighted in red which transition to make the next producible assembly depicted. Our unique terminal assembly is a length $2^3$ line. We will show that by adding a constant number of states, transitions, and affinities to this system the length of the uniquely assembled line will double, and that this process can be repeated to uniquely assemble any length $2^n$ line.

For $n > 3$, Let $T_n$ be the system that uniquely assembles a length $2^n$ line derived by recursively applying the following process to $T_3$ $n - 3$ times. Assuming that $T_n$ uniquely assembles a length $2^n$ line of the form $(1_A, n_D, \ldots, n_D, n_A, n_B, n_F, \ldots, n_F, 1_B)$, $T_{n+1}$ is constructed as follows. First we add the non-initial states $n+1_A, \ldots, n+1_F$, and a transition from $(n_A, n_B)$ to both $(n + 1_E, n_B)$ and $(n_A, n + 1_C)$. We add six new transitions involving $n + 1_C$ or $n + 1_E$ which allow that state to propagate left/right respectively and transition to $n + 1_D$ and $n + 1_F$ respectively when the end to the line assembly is reached. There will be 6 additional transition rules added to allow states $n + 1_D$ and $n + 1_F$ to propagate in the opposite direction and eventually transition $1_A$ and $1_B$ to $n + 1_B$ and $n + 1_A$ respectively. Adding the affinity rule $(n + 1_A, n + 1_B)$ will allow the two length $2^n$ lines to bond uniquely assembling a length $2^{n+1}$ line. This new system uniquely produces a length $2^{n+1}$ line of the same form previously described, to which the process can be repeated to once again double the length of the unique assembly. ◀

▶ **Theorem 4.5.** *For all positive integers $n$, there exists a 1-dimensional freezing Tile Automata system that uniquely assembles an $n$ length line using $\mathcal{O}(\log n)$ states.*

**Proof.** We modify the construction from Lemma 4.4 to build arbitrary length-$n$ lines.

To build any length-$n$ line using $\mathcal{O}(\log n)$ states we modify $T = T_{\lceil \log_2 n \rceil}$. Let $b_i$ indicate the $i^{th}$ least significant bit of $n$'s binary expansion. For all $i > 2$ such that $b_i$ is equal to 1 we add a transition rule from $(i_A, i_B)$ to $(i_L, i_L)$ in $T$. When these two states are adjacent

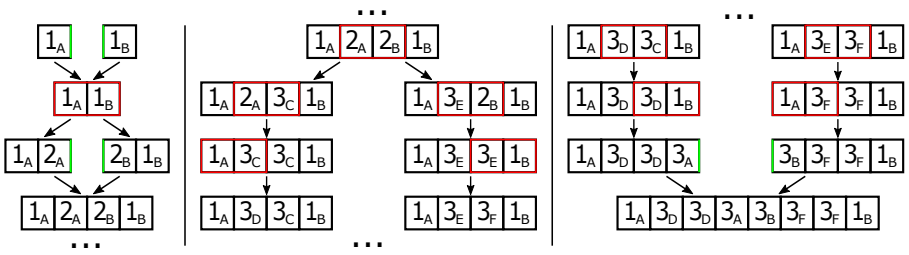


**Figure 3** A system that uniquely builds a length $2^3$ line. The only initial states are $1_A$ and $1_B$. The affinities are between adjacent states. The transition rules are highlighted in red which transition to make the next producible depicted.

they exist in an assembled line of length $2^i$. This transition "locks" this producible, stopping it from growing. Four more transition rules are added to allow this state to propagate to the ends of the line. Finally, we add a transitions between all $i_L$ states and the states $1_B$ and $1_A$, which are the endpoints of the lines. These endpoints transition to states that have affinity with the next largest locked producible on one side. If $b_1$ or $b_2$ is equal to 1 we add in an assembly of size $b_1 \times 1 + b_2 \times 2$ that connects to the last locked producible. ◀

## 4.3 Largest Finite Assembly Problem

Given a positive integer $n$, the Largest Finite Assembly Problem asks what is the largest assembly that can be uniquely assembled in a Tile Automata system using $n$ states.

▶ **Theorem 4.6.** *The Largest Finite Assembly problem in Tile Automata is uncomputable.*

**Proof.** Let $\sigma_n$ be the size of the largest assembly that can be constructed using $n$ states. From Theorem 4.3, there must exists a system that can construct a line of length $BB(n)$ using $\mathcal{O}(n)$ states so $\sigma_{\mathcal{O}(n)} \geq BB(n)$. This means $\sigma_n$ grows asymptotically as fast as the Busy Beaver function, which grows faster than any computable function. Thus, $\sigma_n$ is uncomputable. ◀

# 5 Unique Assembly Verification

A well-studied problem in self-assembly is the Unique Assembly Verification problem. This asks whether a given system uniquely produces a given assembly. We show that the general problem is undecidable. Again, we consider two definitions of Unique Assembly one where systems with cycles are allowed in the production graph, and the other where they are not.

## 5.1 Undecidability

▶ **Theorem 5.1.** *Tile Automata Unique Assembly Verification is undecidable even in one dimension.*

**Proof.** Using Theorem 3.4 we reduce from the halting problem. Given a Turing Machine $M$ we can construct a Tile Automata system $\Gamma$ that simulates $M$. If $M$ halts then there exists a single terminal assembly which is the final accept state tile. If $M$ does not halt then there exists no terminal assemblies. This is true under both definitions of Uniquely Assembly since the only time there would exist a cycle in the production graph of $\Gamma$ is if $M$ ever revisited a configuration. If $M$ revisits a configuration then $M$ will not halt so our system will not uniquely assemble the final accept state tile. ◀

▶ **Theorem 5.2.** *Freezing 2-Dimensional Tile Automata Unique Assembly Verification is undecidable under the definition of Unique Assembly allowing cycles even when all assemblies are of constant height.*

**Proof.** To prove undecidability we reduce from UAV for 1-Dimensional Tile Automata systems (Theorem 5.1). Given an instance of UAV asking if a system $\Gamma$ uniquely produces an assembly $A$ we use the simulation provided in [5] to create a freezing Tile Automata system $\Gamma'$. By the definition of $\Gamma'$ simulating $\Gamma$ if $\texttt{TERM}_\Gamma$ only contains one terminal assembly $A$ then $\texttt{TERM}'_\Gamma$ will only contain one assembly $A'$ that maps to $A$.

The simulation utilizes constant scale macroblocks to represent tiles so the height of the assemblies in $T$ will be constant height. This simulation also uses a token passing scheme that results in cycles in the production graph so this system will not uniquely produce assemblies if cycles are not allowed. ◀

## 6 Affinity Strengthening UAV

Many self-assembly models where UAV is well-studied do not have detachment (and are thus decidable). Here, we investigate versions of TA without this power and show hardness. We do this by exploring Affinity-Strengthening Tile Automata (ASTA). We start by considering the non-freezing case, then consider the added restriction of freezing.

### 6.1 Non-Freezing

▶ **Lemma 6.1.** *The Unique Assembly Verification problem in Affinity-Strengthening Tile Automata is in PSPACE.*

**Proof.** The UAV problem can be solved by the following co-nondeterministic algorithm. Given an Assembly $A$ and an ASTA system $T$, nondeterministically build an assembly $B$ of less than size $2|A|$ where $|A|$ is the size of the given assembly. We now have a branch for every producible assembly and we check the following about $B$ in order. If any branch rejects, the whole algorithm rejects.

- If $B = A$, accept.
- If $|B| \geq |A|$, reject.
- If $B \neq A$ and $B$ is terminal, reject.
- Continue nondeterministically performing construction steps (attachments and transitions) on $B$. If $B$ is reached again, reject. If $A$ is reached, accept.

Only assemblies up to size $2|A|$ can be checked since if any assembly exists larger than $2|A|$, it would have been built using at least one assembly of size greater than $|A|$, which would have already been rejected. We can also check if $B$ is terminal using a nondeterministic subroutine by non-deterministically building a second assembly and checking if it can attach to $B$. Checking if an assembly is breakable or if it is transitionable can be done in polynomial time and space. The final step of the algorithm checks for cycles in the production graph. By the definition of unique assembly, $B \rightarrow^\Gamma A$, by continuing to perform construction steps on $B$ we will eventually reach $A$. If we ever end up reaching $B$ again we know that there exists a cycle in the production graph (cycle checking in a directed graph is in P).

This algorithm shows the UAV problem for Affinity-Strengthening Tile Automata is in coNPSPACE which equals PSPACE. For the case of unique assembly where cycles in the production graph are allowed, the last step of the algorithm is skipped. ◀

▶ **Lemma 6.2.** *The Unique Assembly Verification problem in Affinity-Strengthening Tile Automata is PSPACE-hard.*

**Proof.** We show UAV in Affinity-Strengthening TA is PSPACE-hard by describing how to reduce from any problem $L \in$ PSPACE. Consider a Turing Machine $M$ that decides $L$. The construction from Theorem 3.4 can be modified to be an Affinity-Strengthening system that results in a system capable of performing bounded space computation (a Linear Bounded Automata, which is equivalent to parsing a context-sensitive grammar and is PSPACE-complete [17]). The only transition where a state loses affinity is from the signal accept and reject state to the final accept and reject state. We remove the final states from the system. This will result in two possible terminal assemblies one consisting of a buffer state, then accept states, then another buffer state, and the other being the same with reject states. We remove the buffer state from the set of initial assemblies. We change the length of the assembly representing the input to be the amount of space used by $M$.

Given a bounded space deterministic Turing machine and its input, construct a Tile Automata system that uniquely produces the assembly with accept states if and only if the Turing machine accepts. If the Turing Machine rejects, then the reject assembly will be the only terminal assembly. If the TM ever enters an infinite loop then there will exist a cycle in our system and there will not exist any terminal assemblies, so the TA system will not uniquely produce any assembly regardless of whether there exists a restriction on cycles. ◀

▶ **Theorem 6.3.** *The Unique Assembly Verification problem in Affinity-Strengthening Tile Automata is PSPACE-complete.*

**Proof.** Follows from Lemmas 6.1 and 6.2. ◀

## 6.2 Freezing

In this section we show the complexity of Unique Assembly Verification in a freezing Affinity-Strengthening Tile Automata system. In 2-dimensions, we show UAV is coNP$^{\text{NP}}$-Complete. We utilize the same reduction strategy as in [23]. We conclude by showing coNP-hardness for UAV in one dimension. Note that cycles cannot occur in Freezing Affinity-Strengthening Tile Automata, so we only consider one definition of Unique Assembly.

▶ **Definition 6.4** (∀∃3SAT). *Given a 3SAT formula $\phi(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$, is it true that for every assignment to variables $x_1, \ldots, x_k$, there exists an assignment to $x_{k+1}, \ldots, x_n$ such that $\phi(x_1, \ldots, x_n)$ is satisfied?*

▶ **Lemma 6.5.** *The Unique Assembly Verification problem in freezing Affinity-Strengthening Tile Automata is in coNP$^{NP}$.*

**Proof.** Take the construction and algorithm from Lemma 6.1, we prove that the running time is polynomial. When building an assembly $B$, since the system is freezing we know the time to build $B$ is $|\Sigma||B|$ where $|\Sigma|$ is the number of states in the system. Since we reject if one branch rejects, this is a coNP algorithm.

We utilize one subroutine that is in coNP to check if $B$ is terminal. This is done in polynomial time by nondeterministically building a second assembly and checking if they can attach. If there is an assembly that can attach to $B$, then the assembly is not terminal. Using the coNP algorithm and using the subroutines as oracles, this problem is in coNP$^{\text{NP}}$ ◀

▶ **Lemma 6.6.** *The Unique Assembly Verification problem in freezing Affinity-Strengthening Tile Automata is coNP$^{NP}$-Hard.*

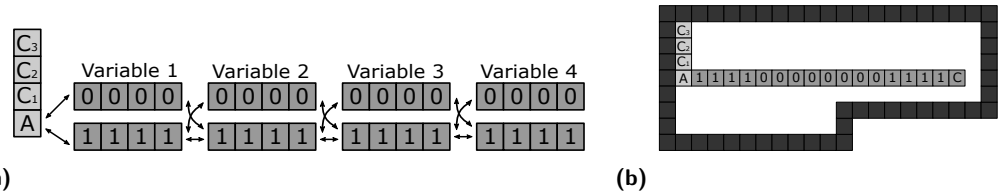



**Figure 4** Part of the construction for Theorem 6.6. (a) The base assemblies are constructed nondeterministically. One is constructed for every possible variable assignment. (b) An example of a base assembly fitting into a frame. $C_x$ binds cooperatively to $C_{x-1}$ and the frame states.

**Proof.** Given an instance of ∀∃3-SAT, this reduction produces a $\tau = 2$ freezing ASTA system which uniquely assembles a target assembly if and only if the instance of ∀∃3-SAT is true. This system has stability threshold 2 to allow for *cooperative binding* in which two assemblies attach using affinities at two separate points, when one of the affinities alone would not be strong enough for this attachment to be stable.

**Overview.** We first create an 'L'-shaped base assembly contained in a larger frame (Figure 4b) that encodes a variable assignment. Rows of this assembly represent clauses and columns represent variables. Each clause is evaluated by cooperatively placing tiles that represent the assignment of the variable in its column, and whether the clause of its row is currently satisfied. Once the assignments are evaluated, additional tiles fill out the rest of the frame. If the assignment evaluates to false, then frame will be filled. If the assignment evaluates to true, then there will be remaining spaces representing the assignment to the variables in the first quantifier. We construct a test assembly for every possible assignment to thee variables that can attach into that space. Once an assembly has completely filled out its frame, all states inside transition into a target state and create our target assembly.

**Base Assemblies.** We construct a rectangular base assembly for every possible variable assignment to $x_1, \ldots, x_n$, with the rows of this assembly representing clauses and columns representing variables. There are two sets of initial states for each variable: one for 0, and one for 1. These sets of states attach to form length-4 line assemblies. The line assemblies have affinities with both the 0 and 1 line assemblies of the next variable. The nondeterministic nature of the model will ensure the creation of all possible combinations of these 0 and 1 line assemblies (Figure 4a). Given $m$ clauses in our 3SAT formula, the TA system includes tiles with initial states $C_1, \ldots, C_m$. These states cooperatively attach to state $A$ and a frame (Figure 4b). The frame ensures there is no unbounded growth. Tiles then cooperatively bind to fill out this structure. The affinities between these states and the variable line assemblies are encoded such that they evaluate if the variable assignment, represented by the base assembly, satisfies the 3SAT formula (Figure 5a). The row containing $C_i$ evaluates whether the $i^{th}$ clause is satisfied by the variable assignment of the base. $U$ and $S$ states cooperatively attach to fill out a row- $U$ indicating the clause has not yet been satisfied, and $S$ indicating that it has. This is done by "passing" the assignment of the variable line upwards with a specific encoding of the affinities. When an $S$ state attaches, only $S$ states can attach to its right side. This allows a $Y$ state to attach at the end of the row if a previous clause was not already evaluated to be unsatisfied. If it is not satisfied, the rightmost state of that row will be $N$, which does not allow a $Y$ state to attach above it.

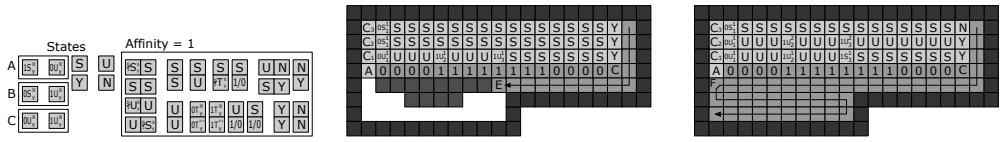



**Figure 5** (a) Initial states needed to evaluate if the variable assignment satisfies the 3SAT formula. Choose 1 from A/B/C for each clause/variable combination. Choose A if 1 assigned to variable $y$ satisfies the $x^{th}$ clause, B if 0 satisfies, and C if the variable does not appear in that clause. $T$ is a placeholder for $U$ or $S$, depending on which was chosen for each clause/variable combination. (b) Example of a 4-variable, 3-clause base assembly that is marked as true (top right "Y"). The assembly grows downward, but interacts with the variable tile line to encode their variable assignment in the assembly's geometry. (c) Example of a 4-variable 3-clause base assembly marked as false (top right "N"). The assembly grows to fill out the entire frame.

Once the rectangle is filled out an assembly will be marked as "True" or "False", represented by the top right $Y/N$ state in the construction. (Figure 5b, 5c). True assemblies grow downward, leaving a space between the base assembly and the frame. The shape of this space is an encoding of this assembly's original variable assignment of $x_1, \ldots, x_k$ (Figure 5b). False assemblies also grow downward, but entirely fill out the frame of the base construction.

**Test Assemblies.** A set of test assemblies are also built using the same nondeterministic method used to create the base assemblies' variable assignments. A test assembly is created for each assignment to variables $x_1, \ldots, x_k$ (Figure 6a). The geometry of a test assembly encodes this variable assignment in a complementary fashion to that of a "True" base assembly representing the same assignment to $x_1, \ldots, x_k$. This allows a test assembly to attach to a "True" base assembly with the same variable assignment to $x_1, \ldots, x_k$, but not to any other due to that causing overlapping geometry. The test assemblies cooperatively bind with two strength-1 affinities at two points (Figure 6b). A test assembly will only be terminal if there is no base assembly matching its variable assignment that was marked as "True".

**Transition to Uniform Assembly.** If the solution to the instance of $\forall\exists$3SAT is true, all assemblies eventually grow/transition to one unique target assembly. To achieve this, there are state transitions which allow every "True"/"False" flagged base assembly to grow into one uniform assembly. For base assemblies marked "True", to which a test assembly attached, the states needed to cooperatively bind these test assemblies to base assemblies having a transition rule to transition to state $T$. For assemblies marked "False", a transition to state $T$ occurs when $A$ and $F$ (Figure 5c) are adjacent. Additional transition rules between state $T$ and all other states (excluding the frame states) allow this state to propagate throughout the entire assembly. The transitions used are shown in Figure 7a. These transitions will change every state besides the frame states to state $T$. This is the target assembly for our created instance of ASTA UAV (Figure 7b).

The only terminal assembly possibly produced that is not the target assembly is a test assembly representing a specific assignment to $x_1, \ldots, x_k$ that could not attach to an assignment assembly marked "True", which represents the same variable assignment. Thus, the system only uniquely assembles the target assembly if the instance of $\forall\exists$3SAT is true. ◀

▶ **Theorem 6.7.** *The Unique Assembly Verification problem in freezing Affinity-Strengthening Tile Automata is coNP$^{NP}$-Complete.*

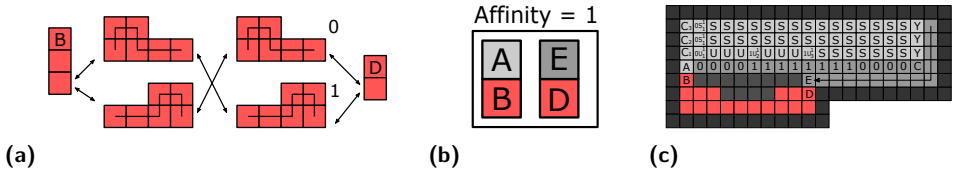



**Figure 6** (a) Test assemblies are nondeterministically built by allowing the possibility for each assignment of one variable construction to attach to either assignment of the next variable construction. (b) Affinities between test assemblies and base assemblies. (c) Example of a test assembly binding to a base assembly that encodes the same variable assignment of $x_1, \ldots, x_k$.

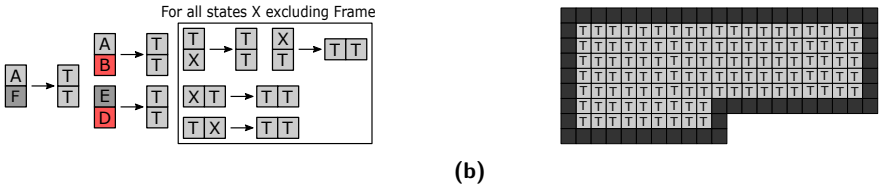



**Figure 7** (a) Transitions Utilized. All states will take the place of $X$, excluding those that are part of the frame. (b) Target Assembly after the $T$ state has fully propagated through the assembly.

**Proof.** Follows from Lemmas 6.5 and 6.6. ◀

▶ **Theorem 6.8.** *The Unique Assembly Verification problem in freezing Affinity-Strengthening Tile Automata is coNP-hard in one dimension.*

**Proof.** We show Affinity Strengthening Freezing UAV is coNP-hard by describing how to reduce from any problem in coNP. Given a problem $L \in$ coNP take a nondeterministic Turing Machine $M$ that decides $L$. From Theorem 3.5, we construct systems that simulate bounded-time Turing Machines. Since we are considering polynomial-time machines, the size of this Tile Automata system is also polynomial. We change the system to be Affinity Strengthening in the same way as in Lemma 6.2. Further, since the Tile Automata model includes nondeterminism in selecting possible transitions for an assembly, we can simulate nondeterministic Turing Machines. We simply have transition rules for each possible outcome.

Using the method described above we can simulate $M$ on $x$. If any of the possible computation paths lead to $M$ accepting, the assembly with the accept states will appear as a terminal assembly. If all possible computations path reject, the only terminal assembly will be the assembly with the reject states. ◀

## 7 Conclusion

In this paper we looked at a powerful new model of self-assembly that combines properties of both cellular automata and hierarchical self-assembly models. We showed that even extremely limited and simple constructions in Tile Automata are powerful and capable of arbitrary computation. We also showed how difficult it is to determine the output of these limited systems. This opens several directions for future work.

One direction is further exploring the assembly of length-$n$ lines in freezing systems. Does there exist a bound on buildable length? Is the finite assembly problem in freezing or other restricted system decidable? Also attempting to construct lines in systems with additional restrictions such as limits on the number of transition rules per state.

For the UAV problem, we show that the general case is undecidable. However, the complexity of the problem in freezing 1-dimensional systems is open. If the problem of asking whether a system is bounded is decidable, then UAV is decidable by first identifying whether a system is bounded and then constructing the production graph and finding the terminal assemblies. The problem for freezing 2-dimensional systems with no cycles is also open.

Since Tile Automata can be seen as a generalization of 2HAM, our results can be compared to the open problem of UAV in that model which is known to be in coNP. The most restricted version of Tile Automata we explore is Affinity Strengthening and freezing, which is only one level of the polynomial hierarchy above other generalizations of 2HAM such as allowing tiles to go into 3-dimensions or allowing a variable temperature. Further limiting Tile Automata may provide more insight into the hardness of these problems.

## References


1 Leonard M. Adleman, Qi Cheng, Ashish Goel, Ming-Deh A. Huang, David Kempe, Pablo Moisset de Espanés, and Paul W. K. Rothemund. Combinatorial optimization problems in self-assembly. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 23–32, 2002.

2 John Calvin Alumbaugh, Joshua J. Daymude, Erik D. Demaine, Matthew J. Patitz, and Andréa W. Richa. Simulation of programmable matter systems using active tile-based self-assembly. In Chris Thachuk and Yan Liu, editors, *DNA Computing and Molecular Programming*, pages 140–158, Cham, 2019. Springer International Publishing.

3 Sarah Cannon, Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Matthew J. Patitz, Robert T. Schweller, Scott M Summers, and Andrew Winslow. Two Hands Are Better Than One (up to constant factors): Self-Assembly In The 2HAM vs. aTAM. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 172–184. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

4 Angel A. Cantu, Austin Luchsinger, Robert Schweller, and Tim Wylie. Covert Computation in Self-Assembled Circuits. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:14, 2019.

5 Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In *International Conference on DNA Computing and Molecular Programming*, pages 155–172. Springer, 2018.

6 Cameron Chalk, Austin Luchsinger, Robert Schweller, and Tim Wylie. Self-assembly of any shape with constant tile types using high temperature. In *Proc. of the 26th Annual European Symposium on Algorithms*, ESA'18, 2018.

7 Matthew Cook. Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.

8 Joshua J. Daymude, Kristian Hinnenthal, Andréa W. Richa, and Christian Scheideler. Computing by programmable particles. In *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, pages 615–681. Springer, Cham, 2019.

9 Erik D Demaine, Martin L Demaine, Sándor P Fekete, Mashhood Ishaque, Eynat Rafalin, Robert T Schweller, and Diane L Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with o (1) glues. *Natural Computing*, 7(3):347–370, 2008.

10 Erik D. Demaine, Sarah Eisenstat, Mashhood Ishaque, and Andrew Winslow. One-dimensional staged self-assembly. In *Proceedings of the 17th international conference on DNA computing and molecular programming*, DNA'11, pages 100–114, 2011.

11 David Doty, Lila Kari, and Benoît Masson. Negative interactions in irreversible self-assembly. *Algorithmica*, 66(1):153–172, 2013.

12 Constantine Evans. *Crystals that Count! Physical Principles and Experimental Investigations of DNA Tile Self-Assembly*. PhD thesis, California Inst. of Tech., 2014.

13 Antonios G Kanaras, Zhenxin Wang, Andrew D Bates, Richard Cosstick, and Mathias Brust. Towards multistep nanostructure synthesis: Programmed enzymatic self-assembly of dna/gold systems. *Angewandte Chemie International Edition*, 42(2):191–194, 2003.

14 Ryuji Kawano. Synthetic ion channels and dna logic gates as components of molecular robots. *ChemPhysChem*, 19(4):359–366, 2018. doi:10.1002/cphc.201700982.

15 Alexandra Keenan, Robert Schweller, Michael Sherman, and Xingsi Zhong. Fast arithmetic in algorithmic self-assembly. *Natural Computing*, 15(1):115–128, March 2016.

16 Ceren Kimna and Oliver Lieleg. Engineering an orchestrated release avalanche from hydrogels using dna-nanotechnology. *Journal of Controlled Release*, April 2019. doi:10.1016/j.jconrel.2019.04.028.

17 Sige-Yuki Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7(2):207–223, 1964. doi:10.1016/S0019-9958(64)90120-2.

18 Austin Luchsinger, Robert Schweller, and Tim Wylie. Self-assembly of shapes at constant scale using repulsive forces. *Natural Computing*, August 2018. doi:10.1007/s11047-018-9707-9.

19 Jennifer E. Padilla, Matthew J. Patitz, Raul Pena, Robert T. Schweller, Nadrian C. Seeman, Robert Sheline, Scott M. Summers, and Xingsi Zhong. Asynchronous signal passing for tile self-assembly: Fuel efficient computation and efficient assembly of shapes. In *Unconventional Computation and Natural Computation*, pages 174–185. Springer, 2013.

20 Robert Schweller and Michael Sherman. Fuel efficient computation in passive self-assembly. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'13, pages 1513–1525. SIAM, 2013.

21 Robert Schweller, Andrew Winslow, and Tim Wylie. Complexities for high-temperature two-handed tile self-assembly. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, pages 98–109, Cham, 2017. Springer International Publishing.

22 Robert Schweller, Andrew Winslow, and Tim Wylie. Nearly constant tile complexity for any shape in two-handed tile assembly. *Algorithmica*, 81(8):3114–3135, 2019.

23 Robert Schweller, Andrew Winslow, and Tim Wylie. Verification in staged tile self-assembly. *Natural Computing*, 18(1):107–117, 2019.

24 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.

25 Andrew Winslow. Staged self-assembly and polyomino context-free grammars. *Natural Computing*, 14(2):293–302, 2015.

# Turning Machines

**Irina Kostitsyna**
Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands
https://www.win.tue.nl/~ikostits/
i.kostitsyna@tue.nl

**Cai Wood**
Hamilton Institute and Department of Theoretical Physics, Maynooth University, Ireland
https://dna.hamilton.ie
cai.wood.2017@mumail.ie

**Damien Woods**
Hamilton Institute and Department of Computer Science, Maynooth University, Ireland
https://dna.hamilton.ie/woods/
damien.woods@mu.ie

## Abstract

Molecular robotics is challenging, so it seems best to keep it simple. We consider an abstract molecular robotics model based on simple folding instructions that execute asynchronously. Turning Machines are a simple 1D to 2D folding model, also easily generalisable to 2D to 3D folding. A Turning Machine starts out as a line of connected monomers in the discrete plane, each with an associated turning number. A monomer turns relative to its neighbours, executing a unit-distance translation that drags other monomers along with it, and through collective motion the initial set of monomers eventually folds into a programmed shape. We fully characterise the ability of Turning Machines to execute line rotations, and to do so efficiently: computing an almost-full line rotation of $5\pi/3$ radians is possible, yet a full $2\pi$ rotation is impossible. We show that such line-rotations represent a fundamental primitive in the model, by using them to efficiently and asynchronously fold arbitrarily large zig-zag-rastered squares and $y$-monotone shapes.

**2012 ACM Subject Classification** Theory of computation → Models of computation

**Keywords and phrases** model of computation, molecular robotics, self-assembly, nubot, reconfiguration

**Digital Object Identifier** 10.4230/LIPIcs.DNA.2020.11

**Funding** Authors C. Wood and D. Woods are supported by European Research Council (ERC) award number 772766 and Science foundation Ireland (SFI) grant 18/ERCS/5746 (this manuscript reflects only the authors' view and the ERC is not responsible for any use that may be made of the information it contains).

**Acknowledgements** We thank Vera Sacristán and Suneeta Ramaswami for insightful ideas and important input. This work began at the 29th Bellairs Winter Workshop on Computational Geometry (March 21-28, 2014 in Holetown, Barbados), we thank Erik Demaine for organising a wonderful workshop and providing valuable feedback, and the rest of the participants for providing a stimulating environment. We also thank Dave Doty and Nicolas Schabanel for helpful comments.

## 1 Introduction

The challenge of building molecular robots has many moving parts, as the saying goes. These include molecular parts that move relative to each other; units needing some sort of memory state; the ability to transition between states; and perhaps even the ability to use computation to drive robotic movements. Here we consider a simple robotic model of reconfiguration called Turning Machines.

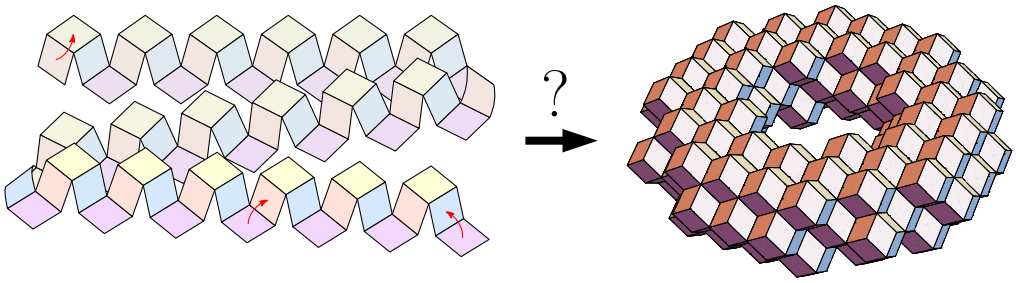




**Figure 1** Turning Machine motivation: what shapes can be made by autonomously folding structures using simple local turning rules that effect non-local movement? Finding suitable abstract models and characterising their ability helps us to step back and create a vision of where we can go.

The main ethos behind our work is the notion of having a reconfigurable structure where component monomers actuate their position relative to their neighbours and governed by *simple* actuation rules. Volume exclusion applies (two monomers can not occupy the same position in space), almost for free we get massive parallelism and asynchronicity, and the complexity of allowable state changes is small: start with a natural number and decrement step-by-step to zero. The Turning Machine model embodies these concepts.

On the one hand, there are a number of senses in which molecular systems are *better* suited to robotic-style reconfiguration than macro-scale robotic systems: there is no gravity nor friction fighting against components' actuation, and should we know how to exploit them, randomness, freely diffusing fuel (robots need not carry all their fuel) and large numbers of components are all readily available as resources. On the other hand, building nanoscale components presents a number of challenges including implementing computational controllers at the nanoscale, as well as designing systems that self-assemble and interact in a regime where we can not easily send in human mechanics to diagnose and fix problems.

## 1.1 Turning machines

Monomers are the atomic components of a Turning machine and are arranged in a connected chain on the triangular/hexagonal grid, with each monomer along the chain pointing at the next. In an initial instance, the chain of monomers are sitting on the $x$-axis all pointing to the east. Each monomer has an initial integer turning number $s \in \mathbb{Z}$, the monomer's ultimate goal is to set that number to 0: if $s$ is positive, the monomer tries to simultaneously decrement $s$ and turn anti/counter-clockwise[1] by an angle of $\pi/3$, if $s$ is negative, it tries to increment and turn clockwise by $\pi/3$.[2] If $s = 0$ the monomer has reached its target orientation and does not turn again. Figures 2 and 3 give the idea, and Section 2 gives a full definition.

A key point is that although a monomer actuates by rotating the direction in which it points, when it does so it "drags" (translates) all monomers that come after it in the chain in the same way the rotation motion of an arm (around a shoulder) appears to translate a flag through the air, or the way a cam in an combustion engine converts rotational shaft motion to translational piston motion.

[1] We define counter-clockwise to be anticlockwise and use these terms interchangeably.

[2] Having the monomer turning angle be confined to the range $(0, \pi/2]$ seems to capture a range of interesting and important blocking behaviours that would otherwise be missed by the model. Having the angle be $\pi/3$, which leads us to the choice of triangular grid over the square grid, is a somewhat arbitrary choice in the model definition.

## 1.2 Turning machines: the main programming challenge

Programming the model simply requires annotating an east-pointing line of monomers with turning numbers; an incredibly simple programming syntax.

Locally, individual monomers exhibit a small rotation, but globally this effects a large translation, or dragging, of many monomers. Thus globally, the main challenge is how to effect global rotations – in other words how to use translation to simulate rotation. In particular, how to do this when lots of monomers are asynchronously moving and bumping into each other, potentially blocking each other from moving.

Blocking comes in two forms. *Temporary blocking* where one monomer is in the way of another, but eventually will get out of the way, and *permanent blocking* where all monomers block each other in a locked configuration that will never free itself. We say that a target structure is foldable if all possible system trajectories lead to that structure, i.e. permanent blocking does not occur. A foldable structure may exhibit temporary blocking on some trajectories, indeed most of the work for our positive results in this paper comes down to showing that for certain folding tasks any blockings that happen are merely temporary kinks in the chain that are eventually worked out. We measure the amount of blocking by considering the completion time: a foldable structure where temporarily blocked monomers can quickly become unblocked finishes faster than one where blocking takes a while to sort out. Our model of time assumes that the time to apply a turning rule to a given unblocked monomer is an exponential random variable with rate 1, and the system evolves as a continuous time Markov chain with the discrete events being rules applied asynchronously and in parallel.

## 1.3 Results

We fully characterise the line rotation capability of the Turning Machine model, in two senses. First, we show that for each of the angles $\theta \in \{\pi/3, 2\pi/3, \pi, 4\pi/3, 5\pi/3\}$, and any number of monomers $n \in \mathbb{N}$ there is a Turning Machine with $n$ monomers that starts on the $x$-axis and ends rotated by $\theta$ radians. We show this is the best one can do, that is, that rotation of $\theta \geq 2\pi$ is impossible (for any $n > 7$, there are always some trajectories that are permanently blocked). Second, line rotation is fast. Up to constant factors the speed is optimal, completing in expected time $O(\log n)$. This shows that despite the fact that line rotations in the range $\pi \leq \theta \leq 5\pi/3$ experience large number of blockings along their trajectories, these blockings are all temporary, and do not conspire to slow the system down by more than a constant factor on average.

To illustrate that line rotation results are indeed a fundamental primitive in the model, as an application, we show how to fold any $n \times n$ square, rastered in a zig-zag fashion (Theorem 17). More generally, this allows us to fold any shape from a wide class called $y$-monotone shapes (see Figure 9), all in optimal expected time $O(\log n)$.

## 1.4 Related and future work

Besides finding insights at the interface of computation and geometry, another ultimate aim of this kind of work is bridge the gap between what we can imagine in theory and what we can engineer in the lab [19]. Biological systems actuated at the molecular scale provide inspiration: in the gastrulation phase of embryonic development of the model organism *Drosophila melanogaster*, large-scale rearrangements of the embryo are effected by thousands of (nanoscale) molecular motors working together to rapidly push and pull the embryo into a target shape [9, 17].

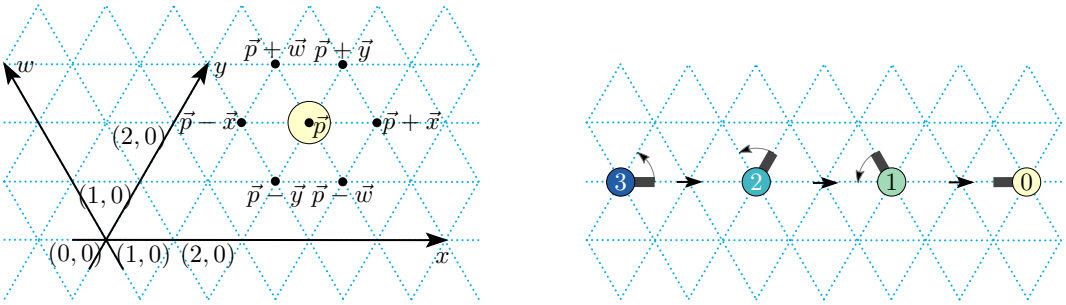



**Figure 2** Turning machine model. Left: Triangular grid conventions. A configuration showing a single monomer on the triangular grid, along with axes $x$, $y$ and $w$. Right: A monomer in state 3 pointing to the east undergoes three turning rule applications finishing in state 0 and no more rules are applicable. Locally, the monomer effects a rotation motion, subsequent figures show the induced global translational, or dragging, motion.

Our Turning Machine model is a restriction of the nubot model [20], a molecular robotic model with many features including self-assembly capabilities, random agitation (jiggling) of monomers, the ability to execute cellular automata style rules, and floppy/rigid molecular bonds. The parallel computing capabilities [4], and construction using random agitation and self-assembly [3] have been studied. Dabby and Chen consider related (experimental and theoretical) systems that use an insertion primitive to quickly grow long (possibly floppy) linear structures [8], later tightly characterised by Hescott, Malchik and Winslow [15, 14] in terms of number of monomer types and time. Hou and Chen [16] show that the nubot model can display exponential growth without needing to exploit state changes. Chin, Tsai and Chen [6] look at both minimising numbers of state changes and number of '2D layers' to assembly 1D structures. There are a number related autonomous self-folding models, both 1D to 2D [5] and 2D to 3D [7], and reconfigurable robotic/programmable matter systems, e.g. [1, 2, 10, 11, 12, 18].

There are several avenues for future work. In this paper, we study model instances with natural number states, leading to anti-clockwise rotation motion (that is, anti-clockwise translation about the origin). Does the combination of clockwise and anti-clockwise turning rules increase the expressivity of the model? Using a variant [20, 3] of the model with random agitation of monomers would side-step our main negative result about the impossibility of full $2\pi$ line rotation by allowing reversible movement out of blocked configurations. Indeed, the analysis of such systems would provide intellectual fruit by mixing probability, geometry and computation. As indicated in Figure 1, it is straightforward to generalise the model to (say) 2D trees folding into 3D shapes, this provides an interesting avenue for exploration. In all of these cases fully characterising the class of shapes that can be folded, and characterising the time to fold such classes of structures, provides a number of questions whose answers would expand our understanding of the capabilities of simple reconfigurable robotic systems.

## 2 Turning machine model definition

In this section we define the Turning Machine model. Formally speaking, the model is a restriction of the Nubot model [20], for simplicity we instead use a custom formalism.

**Grid.** Positions are pairs in $\mathbb{Z}^2$ defined on a two-dimensional triangular grid using $x$ and $y$ axes as shown in Figure 2. For convenience, we define a third axis, $w$, centred on the origin and running through the point $(x, y) = (-1, 1)$. We let $\pm\overrightarrow{x}, \pm\overrightarrow{y}, \pm\overrightarrow{w}$ denote the unit vectors along the $x$, $y$ and $w$ axes.

**Monomer, configuration, trajectory.** A monomer is a pair $m = (s(m), \text{pos}(m))$ where $s(m) \in \mathbb{Z}$ is a state and $\text{pos}(m_i) \in \mathbb{Z}^2$ is a position. A *configuration*, of length $n \in \mathbb{N}$, is a tuple of monomers $c = (m_0, m_1, \ldots, m_{n-1})$ whose positions $\sigma(c) = \text{pos}(m_0), \text{pos}(m_1), \ldots, \text{pos}(m_{n-1})$ define a length $n-1$ simple directed path (or non-self-intersecting chain) in $\mathbb{Z}^2$ (on the triangular grid) and where $\text{pos}(m_0) = (0,0)$.[3]

A configuration is a tuple of $n \in \mathbb{N}$ monomers $(m_0, m_1, \ldots, m_{n-1})$. A *final configuration* has all monomers in state 0. A pair of configurations $(c_i, c_{i+1})$ is said to be a *step* if $c_i$ yields $c_{i+1}$ via a single *rule application* (defined below) which we write as $c_i \to c_{i+1}$. A trajectory, of length $k$, is a sequence of configurations $c_0, c_1, \ldots, c_{k-1}$ where, for each $i \in \{0, 1, \ldots, k-2\}$ the pair $(c_i, c_{i+1})$ is a step $c_i \to c_{i+1}$. A Turning machine *initial configuration* $c_0$ is said to *compute the target configuration* $c_t$ if all trajectories that start at $c_0$ lead to $c_t$, and is said to compute its target configuration if it reaches the configuration with all monomers in state 0. A *Turning machine* instance is an initial configuration. For a monomer $m_i$, we let $s_0(m_i)$ denote its state in the initial configuration.

**Turning rule: state decrement.** Let $S_{\text{init}} \subsetneq \mathrm{Z}$ be the set of states that appear in the initial configuration.[4] Let $s_{\min} = \min(S_{\text{init}} \cup \{0\})$ and $s_{\max} = \max(S_{\text{init}} \cup \{0\})$, and let $S = \{s_{\min}, s_{\min}+1, \ldots, s_{\max}\}$ be the called the Turning machine *state set*. The *turning rules* of a turning machine are defined by a function $r$ such that for all states $s \in (S \setminus \{0\})$:

$$r(s) = \begin{cases} s-1 & \text{if } s > 0\,, \\ s+1 & \text{if } s < 0\,. \end{cases} \tag{1}$$

Let $\mathcal{C}$ be the set of all configurations. The turning rule $R : \mathcal{C} \times \mathbb{Z} \to \mathcal{C}$ is a function and $R(c,i)$ is said to be *applicable* to monomer $m_i$ in configuration $c$ if $s(m_i) \neq 0$ and the rule is not blocked (defined below). If the rule is applicable, we write $R(c,i) = c'$ and say that $R(c,i)$ yields the new configuration $c'$, and we say that $(c, c')$ is a step.

**Turning rule: blocking.** For $i \in \{0, 1, \ldots, n-1\}$, we define the head and tail of monomer $m_i$ as $\text{head}(m_i) = m_{i+1}, m_{i+2}, \ldots, m_{n-1}$ and $\text{tail}(m_i) = m_0, m_1, \ldots, m_i$.

Consider the following tuple of unit vectors: $\vec{d} = (\vec{x}, \vec{y}, \vec{w}, -\vec{x}, -\vec{y}, -\vec{w})$, and let $\vec{d}_k$ denote the $k$th element of that tuple. Let $\vec{d}_i = \text{pos}(m_{i+1}) - \text{pos}(m_i)$, i.e. the unit vector from monomer $m_i$ to $m_{i+1}$, and then let $i' = (i+2) \mod 6$. For a vector $\vec{v} \in \mathbb{Z}^2$ we write $m_i + \vec{v}$ to mean the monomer $m_i$ translated by $\vec{v}$. Define[5] $\text{head}^{\rightarrow}(m_i) = m_{i+1} + \vec{d}_{i'}, m_{i+2} + \vec{d}_{i'}, \ldots, m_{n-1} + \vec{d}_{i'}$. If the set of positions of $\text{tail}(m_i)$ has a non-empty intersection with the set of positions of $\text{head}^{\rightarrow}(m_i)$ we say that the rule is blocked, and the rule is not applicable. If the rule is not blocked, it is applicable and the resulting next configuration is $c' = \text{tail}(m_i), \text{head}^{\rightarrow}(m_i) = m_0, m_1, \ldots, m_i, m_{i+1} + \vec{d}_{i'}, m_{i+2} + \vec{d}_{i'}, \ldots, m_{n-1} + \vec{d}_{i'}$.

A configuration $c$ is said to be *permanently blocked* if (a) not all states are 0, and (b) none of the monomers in $c$ has an applicable rule. A monomer $m$ within a configuration $c$ is said to be *temporarily blocked* if (a) $m$ is not in state 0, and (b) there is no rule applicable to $m$, and (c) there is a trajectory starting at $c$ that reaches a configuration $c'$ where there is a rule applicable to $m$.

---

[3] In the language of [20], one can imagine that for all $i \in \{0, 1, \ldots, n-2\}$, there is a rigid bond between monomer $m_i$ and monomer $m_{i+1}$, and otherwise there are no bonds.

[4] Throughout this paper, only natural number states are used. However, for generality, symmetry and potential future work, we intentionally define the model to have integer states.

[5] Another way to state this is that when a monomer $m_i$ moves, $\text{head}(m_i)$ translates in the direction corresponding to the current direction of $m_i$ rotated by the angle $2\pi/3$.

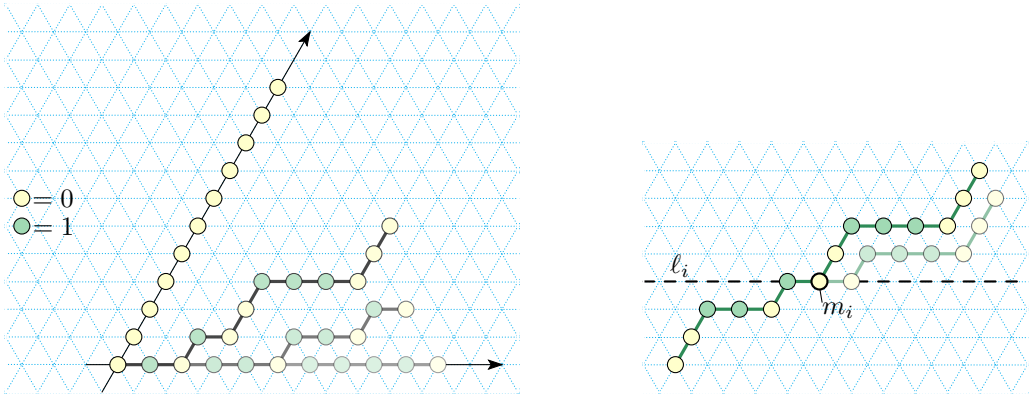



**Figure 3** Left: The Turning Machine $L_n^1$ that rotates a line of $n = 11$ monomers by $\pi/3$; illustration for Lemma 5. Four configurations are shown. The initial configuration has all monomers in state 1 sitting on the $x$-axis, in the final configuration all are in state 0 and sitting on the $\pi/3$ line. Two intermediate configurations are shown, respectively after 2, and then after 5, turning rules applications. Right: A configuration of some Turning Machine from the class $\mathcal{M}_{11}^3$ with the chain running from bottom left to top right. Lemmas 5 and 6 uses the fact that $\text{tail}(m_i)$ sits on or below $\ell_i$, $\text{head}(m_i)$ sits on or above $\ell_i$, and $\text{head}^{\rightarrow}(m_i)$ sits strictly above $\ell_i$.

**Time.** A Turning Machine evolves as a continuous time Markov process. The rate for each rule application is 1. If there are $k$ applicable transitions for a configuration $c_i$ (i.e. $k$ is the sum of the number of rule applications that can be applied to all monomers in $c_i$), then the probability of any given transition being applied is $1/k$, and the time until the next transition is applied is an exponential random variable with rate $k$ (i.e. the expected time is $1/k$). The probability of a trajectory is then the product of the probabilities of each of the transitions along the trajectory, and the expected time of a trajectory is the sum of the expected times of each transition in the trajectory. Thus, $\sum_{t\in\mathcal{T}} \Pr[t] \cdot \text{time}(t)$ is the expected time for the system to evolve from configuration $c_i$ to configuration $c_j$, where $\mathcal{T}$ is the set of all trajectories from $c_i$ to $c_j$, and $\text{time}(t)$ is the expected time for trajectory $t$.

▶ **Example.** *The proof of Lemma 5 in Appendix A, and Figure 3, illustrate these concepts.*

## 3 Classes of Turning Machines: line rotation and square

Every Turning Machine analysed in this paper starts with $n \in \mathbb{N}$ monomers, sitting on the $x$-axis, as formalised in the following definition.

▶ **Definition 1** ($\mathcal{M}_n^{\leq\sigma}$). *Let $n, \sigma \in \mathbb{N}$. We let $\mathcal{M}_n^{\leq\sigma}$ denote the set of $n$-monomer Turning Machines with initial configuration $c_0 = m_0, m_1, \ldots, m_{n-1}$ having all monomers positioned on the $x$-axis ($\text{pos}(m_i) = (i, 0) \in \mathbb{Z}^2$) and pointing to the east, and with initial states $s_0(m_i)$ bounded by $\sigma$, i.e. $s_0(m_i) \leq \sigma$ for all $0 \leq i \leq n-2$, and $s_0(m_{n-1}) = 0$.*

We next define a sub class of $\mathcal{M}_n^{\leq\sigma}$ machines, called "line rotation" Turning Machines.

▶ **Definition 2** (Line rotation Turning Machine). *Let $n \in \mathbb{N}$ and let $L_n^\sigma$ be the Turning Machine with initial configuration of $n$ monomers $c_0 = m_0, m_1, \ldots, m_{n-1}$ all pointing to the east, positioned on the $x$-axis ($\text{pos}(m_i) = (i, 0) \in \mathbb{Z}^2$), and for $0 \leq i \leq n-2$ all monomers in the same state $s_0(m_i) = \sigma \in \mathbb{N}^+$ and $s_0(m_{n-1}) = 0$.*

▶ Remark 3. The initial monomer state $\sigma \geq 0$ dictates that each monomer wishes to turn (have a rule applied) a total $\sigma$ times, i.e. be rotated through an angle of $\sigma\pi/3$.

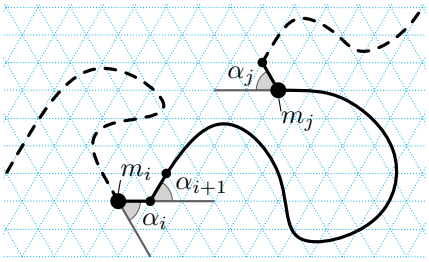



**Figure 4** Illustration of turn angle (Definition 7). The turn angles $\alpha_i$ and $\alpha_{i+1}$ are positive (and to the left), and $\alpha_j$ is negative (and to the right).

▶ Remark 4 (Target configuration). For intuition, if there was no notion of blocking in the Turning Machine model, that is, if the model permitted self-intersecting configurations (which it does not), then the final configuration $c$ of the Turning Machine in Definition 2 is a straight line of monomers sitting along the ray that starts at the origin and is at an angle of $\sigma\frac{\pi}{3}$, i.e. at positions $(0,0),(0,-1),\ldots,(0,-(n-1))$ and all pointing to the west. We call $c$ the desired *target configuration* of the line rotation Turning Machine $L_n^\sigma$. Also, if there was no notion of blocking: expected time to completion would be fast, $O(\log n)$ (by a generalisation of the analysis used in the proof of Lemma 5). However, a model with no blocking would be rather uninteresting.

Figure 3 (left) illustrates Lemma 5 and Appendix A contains its straightforward, yet instructive, proof.

▶ **Lemma 5.** *For each $n \in \mathbb{N}$, the line-rotating Turning Machine $L_n^1$ computes its target configuration, and does so in expected $O(\log n)$ time.*

Lemma 6 is illustrated in Figure 3 (right).

▶ **Lemma 6.** *Let $n \in \mathbb{N}$ and let $L_n^{\leq 3}$ be a Turning Machine in $\mathcal{M}_n^{\leq 3}$ (Definition 1). Let $m_i$ for $0 \leq i \leq n-1$ be a monomer in some reachable configuration $c$ of $L_n^{\leq 3}$. The monomers* $\mathrm{head}(m_i)$ *are positioned on or above $\ell_i$, and* $\mathrm{tail}(m_i)$ *are positioned on or below $\ell_i$.*

**Proof.** The claim follows from the fact that in any configuration of $L_n^{\leq 3}$, and for any $j \in \{0,1,\ldots,n-2\}$ the angle of the vector $\overrightarrow{\mathrm{pos}(m_j)\mathrm{pos}(m_{j+1})}$ (from monomer $m_j$ to $m_{i+1}$) is either 0°, 60°, 120°, or 180° (and, in particular, is not strictly between 180° and 360°). ◀

## 4 Tools for reasoning about Turning machines

The notion of turn angle of a monomer is crucial to our analysis and is illustrated in Figure 4.

▶ **Definition 7** (Turn angle). *Let $c$ be the configuration of an $n$-monomer Turning Machine and let $0 \leq i < n-1$. The* turn angle $\alpha_i$ at monomer $m_i$ *is the angle between $\overrightarrow{\mathrm{pos}(m_{i-1})\mathrm{pos}(m_i)}$ and $\overrightarrow{\mathrm{pos}(m_i)\mathrm{pos}(m_{i+1})}$, and it is the positive counterclockwise angle if the points $\mathrm{pos}(m_{i-1}),\mathrm{pos}(m_i),\mathrm{pos}(m_{i+1})$ make a left turn*[6]*, and the negative clockwise angle otherwise.*

[6] The notion of left or right turn along the three points $\mathrm{pos}(m_{i-1}),\mathrm{pos}(m_i),\mathrm{pos}(m_{i+1})$ can be formalised by considering the line $\ell_i$ running through $\mathrm{pos}(m_i)$, in the direction $\overrightarrow{\mathrm{pos}(m_{i-1})\mathrm{pos}(m_i)}$, noting that $\ell_i$ cuts the plane in two, and defining the left- and right-hand side of the plane with respect to the vector along $\ell_i$.

For a monomer $m_i$, the following definition gives a measure, $\Delta s(m_i)$, of how its state $s(m_i)$ has progressed since the initial configuration.

▶ **Definition 8.** *Let $c$ be a reachable configuration of an $n$-monomer Turning Machine. Define $\Delta s(m_i)$ to be the number of rule applications to (moves of) the monomer $m_i$ from the initial configuration to $c$. That is, $\Delta s(m_i) = s_0(m_i) - s(m_i)$, where $s_0(m_i)$ is the initial state of $m_i$, and $s(m_i)$ is the state of $m_i$ in configuration $c$.*

▶ **Lemma 9** (Difference of State is $\leq 2$)**.** *Let $n \in \mathbb{N}$, and let $c$ be any reachable configuration of an $n$-monomer Turning Machine $T_n$ with non-negative initial states, then*

$$|\Delta s(m_i) - \Delta s(m_{i+1})| \leq 2\,,$$

*for all $0 \leq i < n-1$.*

**Proof.** Let $m_k^t$, for $t \in \mathbb{N}$ and $k \in \{0, 1, \ldots, n-1\}$, denote the $k^{\text{th}}$ monomer in the $t^{\text{th}}$ configuration $c_t$. Initially, $\Delta s(m_j^0) = 0$ for all monomers $m_j$, and thus $|\Delta s(m_i^0) - \Delta s(m_{i+1}^0)| = 0$.

Observe, that $|\Delta s(m_i) - \Delta s(m_{i+1})| \neq 3$ because otherwise $\text{pos}(m_i) = \text{pos}(m_{i+2})$ making $c$ a self-intersecting (non-simple) configuration, contradicting its definition.

By Equation (1), when a rule is applied to one of $m_i^t$ or $m_{i+1}^t$ its state decreases by 1 and its $\Delta s(\cdot)$ increases by 1. Then $|\Delta s(m_i^t) - \Delta s(m_{i+1}^t)| = |\Delta s(m_i^{t-1}) - \Delta s(m_{i+1}^{t-1})| \pm 1$. When a rule is applied to some other monomer $m_k$ with $i \neq k \neq j$, then $|\Delta s(m_i^t) - \Delta s(m_{i+1}^t)| = |\Delta s(m_i^{t-1}) - \Delta s(m_{i+1}^{t-1})| \pm 0$. Thus, after each rule application the value of $|\Delta s(m_i) - \Delta s(m_{i+1})|$ changes by at most 1, and as it cannot be equal to 3, we have that $|\Delta s(m_i) - \Delta s(m_{i+1})| \leq 2$. ◀

We can now show the following lemma, which proves a relation between the states of any two monomers of a Turning Machine and the geometry of the current configuration.

▶ **Lemma 10.** *Let $c$ be any reachable configuration of an $n$-monomer Turning Machine $T_n$, whose initial configuration $c_0$ has all monomers pointing in the same direction, and let $m_i$ and $m_j$ be two monomers of $c$ such that $i < j < n-1$, then*

$$\Delta s(m_j) - \Delta s(m_i) = \frac{3}{\pi} \sum_{k=i+1}^{j} \alpha_k\,,$$

*where $\alpha_k$ is the turn angle at monomer $m_k$.*

**Proof.** For any intermediate configuration, the turn angle $\alpha_{i+1}$ between monomers $m_i$ and $m_{i+1}$ depends only on the number of moves each monomer has made. Initially, $\alpha_{i+1} = 0$, and it increases by $\pi/3$ each time monomer $m_i$ moves, and decreases by $\pi/3$ every time monomer $m_{i+1}$ moves. By Lemma 9, for two consecutive monomers $m_i$ and $m_{i+1}$, in any configuration, $|\Delta s(m_i) - \Delta s(m_{i+1})| \leq 2$. Hence, for a pair of consecutive monomers $m_i$ and $m_{i+1}$, the turn angle $\alpha_{i+1}$ is in the range $[-2\frac{\pi}{3}, 2\frac{\pi}{3}]$, and thus $\alpha_{i+1} = \Delta s(m_{i+1}) - \Delta s(m_i)$. Summing over all $i$ gives the lemma conclusion. ◀

The following technical lemma is used extensively for our main results. Intuitively, it tells us that high-state monomers are not blocked.

▶ **Lemma 11.** *Let $T_n \in \mathcal{M}_n^s$ be a Turning Machine with maximum state $s \leq 5$. In any reachable configuration $c$ of $T_n$ no monomer $m_i$ with $\Delta s(m_i) \leq 1$ is blocked (neither temporarily blocked nor permanently blocked).*

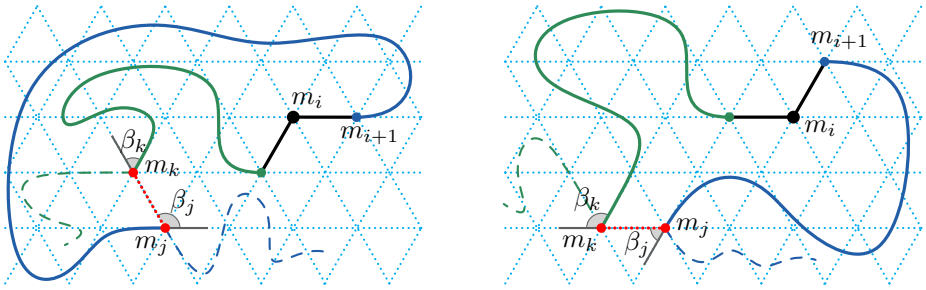



**Figure 5** Illustration for Lemma 11. Monomer $m_i$ is shown in black, head($m_i$) is shown in blue and tail($m_i$) is shown as the green curve plus the black monomer $m_i$. Left: monomer $m_i$ is in its initial state ($\Delta s(m_i) = 0$), and polygon $P$ is traversed counter-clockwise. Right: monomer $m_i$ has moved once ($\Delta s(m_i) = 1$), and polygon $P$ is traversed clockwise.

**Proof.** Suppose, for the sake of contradiction, there is a blocked monomer $m_i$ with $\Delta s(m_i) \leq 1$. Then there exist two monomers $m_j \in \text{head}(m_i)$ and $m_k \in \text{tail}(m_i)$ such that $\text{pos}(m_k) = \text{pos}'(m_j)$, where $\text{pos}'(m_j)$ is the position of $m_j$ in $\overrightarrow{\text{head}}(m_i)$ (see Figure 5).

By definition of head and tail we know that $k \leq i < j$. Consider the closed chain $P = \text{pos}(m_k), \text{pos}(m_{k+1}), \ldots, \text{pos}(m_{j-1}), \text{pos}(m_j), \text{pos}(m_k)$. Since configurations are simple, $P$ defines a simple polygon. The turn angles of a simple polygon sum to $2\pi$ if the polygon is traversed anticlockwise (interior of $P$ is on the left-hand side while traversing), or $-2\pi$ if the polygon is traversed clockwise (interior of $P$ is on the right-hand side). For $P$, this sum is defined as:

$$\alpha_P = \sum_{\ell=k+1}^{j-1} \alpha_\ell + \beta_j + \beta_k = \pm 2\pi \,,$$

where $\alpha_\ell$ is the turn angle at monomer $m_\ell$, and $\beta_j$ and $\beta_k$ are the turn angles of the polygon at vertices $\text{pos}(m_j)$ and $\text{pos}(m_k)$ respectively (see Figure 5). More precisely,

$$\begin{aligned}
\alpha_\ell &= \angle(\overrightarrow{\text{pos}(m_\ell)} - \overrightarrow{\text{pos}(m_{\ell-1})}, \overrightarrow{\text{pos}(m_{\ell+1})} - \overrightarrow{\text{pos}(m_\ell)}) \,, \\
\beta_j &= \angle(\overrightarrow{\text{pos}(m_j)} - \overrightarrow{\text{pos}(m_{j-1})}, \overrightarrow{\text{pos}(m_k)} - \overrightarrow{\text{pos}(m_j)}) \,, \text{ and} \\
\beta_k &= \angle(\overrightarrow{\text{pos}(m_k)} - \overrightarrow{\text{pos}(m_j)}, \overrightarrow{\text{pos}(m_{k+1})} - \overrightarrow{\text{pos}(m_k)}) \,.
\end{aligned}$$

Furthermore, by Lemma 10,

$$\Delta s(m_{j-1}) - \Delta s(m_k) = \frac{3}{\pi} \sum_{\ell=k+1}^{j-1} \alpha_\ell \,.$$

Thus,

$$\Delta s(m_{j-1}) = \Delta s(m_k) + \frac{3}{\pi} \sum_{\ell=k+1}^{j-1} \alpha_\ell = \Delta s(m_k) + \frac{3}{\pi}(\pm 2\pi - \beta_j - \beta_k) = \Delta s(m_k) \pm 6 - \frac{3}{\pi}(\beta_j + \beta_k) \,.$$

Observe that when a monomer $m_i$ moves, its head translates in the direction corresponding to the current direction of $m_i$ rotated by angle $2\pi/3$. Therefore, the state of $m_k$ can be represented as a function of the state of $m_i$ and the angle $\beta_k$, more precisely

$$\Delta s(m_k) = \Delta s(m_i) + 2 + \frac{3}{\pi}\beta_k \,.$$

(See Figure 5 for an example.) Therefore, by the previous two equalities

$$\Delta s(m_{j-1}) = \Delta s(m_i) + 2 \pm 6 - \frac{3}{\pi}\beta_j \,.$$

Recall, that the angle $\beta_j \in [-2\pi/3, 2\pi/3]$, that $0 \leq \Delta s(m_i) \leq 1$ by the assumption of the lemma, and that $\Delta s(m_{j-1}) \leq s$. If the polygon defined by $P$ is traversed counter-clockwise, then

$$\Delta s(m_{j-1}) = \Delta s(m_i) + 8 - \frac{3}{\pi}\beta_j \geq 0 + 8 - 2 = 6\,,$$

which implies that $s(m_{j-1})$ is out of the range of valid states, as $m_{j-1}$ must have moved more times as its initial state. Else, if the polygon $P$ is traversed clockwise, then

$$\Delta s(m_{j-1}) = \Delta s(m_i) - 4 - \frac{3}{\pi}\beta_j \leq 1 - 4 + 2 = -1\,,$$

which again implies that $s(m_{j-1})$ is out of the range of valid states, as $m_{j-1}$ must have moved in the wrong direction. In either case we contradict that the state $s(m_{j-1})$ is in the range of valid states, and, therefore, the monomer $m_i$ is not blocked. ◀

▶ **Lemma 12.** *Let $L_n^s$ be a line-rotating Turning Machine with $s \leq 5$. Let $c$ be a reachable configuration of $L_n^s$ where each monomer $m_i$ in $c$ has $s_c(m_i) < s$. Then the line-rotating Turning Machine $L_n^{s-1}$ has a reachable configuration $c'$ such that for every $m_i$, $s_{c'}(m_i) = s_c(m_i)$ and the geometry (chain of positions) of $c$ is equal to that of the rotation of $c'$ by $\pi/3$ around the origin.*

**Proof.** Consider the sequence $\rho_c$ rule applications (moves) that brings the initial configuration of $L_n^s$ to configuration $c$. We claim that $\rho_c$ can be converted into another sequence $\rho_{c'}$, of the same length, in which the first $n-1$ moves are by monomers in state $s$.

First, we claim: for any two consecutive moves, where the second move is applied to a monomer in state $s$, swapping the two moves results in a valid sequence of moves transforming the Turning Machine into the same configuration. Let the first move be applied to monomer $m_i$ which transitions from state $s'$ to $s'-1$, and the second move be applied to monomer $m_j$ which transitions from state $s$ to $s-1$. Suppose for the sake of contradiction that swapping the moves results in at least one of the monomers $m_i$ or $m_j$ being blocked. We begin by attempting to apply the move to monomer $m_j$, but, by Lemma 11, that move is not blocked. Then we attempt to apply a move to monomer $m_i$, but that is not blocked either since the coordinates of all monomers before and after swapping the two moves are exactly the same; i.e. the resulting configuration is a valid (non-self-intersecting) configuration in both cases. Hence neither monomer is blocked.

Thus, the original sequence of moves resulting in configuration $c$, can be converted into another sequence where the first $n-1$ moves are applied to monomers in state $s$. Then, after the first $n-1$ moves the configuration of $L_n^s$ is equivalent to the initial configuration of $L_n^s$ but rotated by $\pi/3$ and with all monomers in state $s-1$. Hence equivalent to the initial configuration of $L_n^{s-1}$ rotated by $\pi/3$.

Applying the remaining moves to $L_n^{s-1}$ will transform it into configuration $c'$. ◀

## 5 Line rotation to $5\pi/3$

In this section we show that for $1 \leq s \leq 5$ the line-rotation Turning Machine $L_n^s$ computes its target configuration of a $s\pi/3$ rotated line (Theorem 13), and does so in expected time $O(\log n)$ (Theorem 14). In addition to those results for any state $s \leq 5$, in Appendix A we

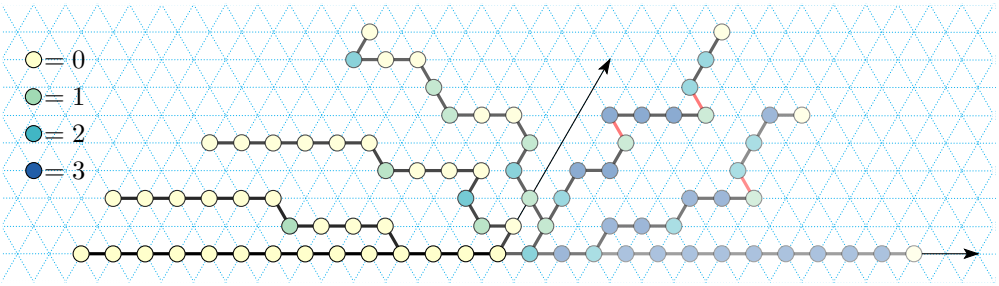



**Figure 6** Example trajectory of the Turning Machine $L_n^3$ that rotates a line of east-pointing monomers by an angle of $\pi$. Illustration for Theorem 14 with $s = 3$ (and for Lemma 20 and Theorem 21 in Appendix A). Seven configurations are shown, the initial configuration has all monomers in state 3 (blue), final in state 0 (yellow). Darker shading indicates later in time. A red bond (edge) indicates a blocked monomer. The proof of Lemma 20 shows that only monomers in state 1 are ever blocked and only when they are adjacent to a monomer in state 3, and that all such blockings are temporary – if we wait long enough they become unblocked.

include stand-alone proofs for each of $s = 1$, $s = 3$, and $s = 4$ which showcase a variety of geometric techniques for analysing Turning Machine movement, but are not needed to prove our main results. Also, the cases of $s = 1$ and $s = 3$ are illustrated in Figures 3 and 6.

▶ **Theorem 13.** *For each $n \in \mathbb{N}$ and $1 \leq s \leq 5$, the line-rotation Turning Machine $L_n^s$ computes its target configuration.*

**Proof.** We prove by induction on $1 \leq s \leq 5$ that any reachable configuration $c$ of $L_n^s$ is not permanently blocked.

Base case $s = 1$. In any configuration reachable by $L_n^1$, monomers have either state $s = 1$ or 0. Monomers in state $s = 1$ cannot be permanently blocked by Lemma 11. Thus, any non-final configuration is not permanently blocked.

Assume for $s - 1$ the claim is true, i.e. it holds for $L_n^{s-1}$. We will prove that for $s$ it is also true, i.e. it holds for $L_n^s$. Suppose, for the sake of contradiction, there is a permanently blocked configuration $c$ of $L_n^s$ for some $n \in \mathbb{N}$ and $s \leq 5$. If there is no monomer in $c$ in state $s$, then by Lemma 12 there exists a corresponding configuration $c'$ in $L_n^{s-1}$ with monomers $m'_0, m'_1, \ldots, m'_{n-1}$, such that, for any monomer $m_i$ in $c$ with state $s_i < s$ the corresponding monomer $m'_i$ in $c'$ has the same state $s_i$. Configurations $c$ and $c'$ form chains equal up to rotation by angle $\pi/3$. Configuration $c'$ is not blocked by the induction hypothesis, thus configuration $c$ cannot be blocked either.

On the other hand, if there is a monomer $m_i$ in configuration $c$ in state $s$, then by Lemma 11 it is unblocked, and configuration $c$, again, is not blocked.

Hence the induction hypothesis holds for $s$, and $L_n^s$ does not have a reachable permanently blocked configuration. ◀

▶ **Theorem 14.** *For each $n \in \mathbb{N}$ and $1 \leq s \leq 5$, the line-rotation Turning Machine $L_n^s$ computes its target configuration in expected $O(\log n)$ time.*

**Proof.** By Theorem 13, $L_n^s$ computes its target configuration. For the time analysis we use a proof by induction on $u \in \{0, 1, \ldots, s\}$, in decreasing order.

The induction hypothesis is that for a reachable configuration $c_u$ of $L_n^s$ with maximum state value $u$ (there may be states $< u$ in the configuration), the expected time to reach a configuration $c_{u-1}$ with maximum state $u - 1$ is $O(\log n)$.

For the base case we let $u = s$ and assume $c$ is such that all monomers are in state $u$. Hence $c$ is an initial configuration and hence, by definition, is reachable. By Lemma 11, monomers in state $s$ are never blocked and hence we claim that the first configuration with maximum state $u-1$ appears after expected time $O(\log n)$. To see this claim, note that for each monomer $m_i$ in state $s(m_i) = u$ the rule application that sends $m_i$ to state $u-1$ occurs at rate 1, independently of the states and positions of the other monomers (by Lemma 11, there is no blocking of a monomer in state $u = s$). Since there are $n$ monomers in state $u$, the expected time for all $n$ to transition to $u-1$ is [13]:

$$\sum_{k=1}^{n} \frac{1}{k} = O(\log n)\,. \tag{2}$$

We assume the inductive hypothesis is true for $0 < u+1 \leq s$, and we will prove it holds for $u$. Thus, there exists a reachable configuration $c_u$ where the maximum state value is $u \leq s$, which is reachable from $c_{u+1}$ in expected $O(\log n)$ time. Let there be $n' \leq n$ monomers in state $u$ in $c_u$. By Lemma 12, there is a line-rotating Turning Machine $L_n^u$ that has a reachable configuration $c'_u$ such that for every $m_i$ in $c_u$, $s_{c'_u}(m_i) = s_{c_u}(m_i)$ and the positioning of $c_u$ is equal to the rotation of $c'_u$ by $\pi/3$ around the origin. By Lemma 11 monomers in state $u$ in $L_n^u$ are never blocked, hence monomers in state $u$ in $c_u$ are not blocked either. Setting $n = n'$ in Equation (2), and noting that $O(\log n') = O(\log n)$, proves the inductive hypothesis for $u$.

Since we need to apply the inductive argument at most $s \leq 5$ times, by linearity of expectation, the expected finishing time for the $s$ processes is their sum, $5 \cdot O(\log n) = O(\log n)$. ◀

## 6 Line rotation to $2\pi$ is impossible

▶ **Theorem 15.** *For all $n \in \mathbb{N}, n \geq 7$, the line-rotating Turning Machine $L_n^6$ does not compute its target configuration. In other words, there is a permanently blocked reachable configuration.*

**Proof.** Figure 7, looking only at blue monomers and edges, shows a valid trajectory of $L_7^6$ , then ends in a permanently blocked configuration, hence the lemma holds for $n = 7$.

Let $n > 7$, and in Figure 7 let the red line segment denote a straight line $\ell_{n-7}$ of $n-7$ monomers co-linear with the red line segment. By inspection, it can be verified that (a) in all 25 configurations the line $\ell$ does not intersect any blue monomer, and moreover (b) the transitions from configurations 1 through 14, configurations 17 through 23, and configuration 24 to 25 are all valid, meaning that the length $n-7$ line $\ell_{n-7}$ does not block the transition. The transitions for configurations 14 through 17 are valid by Theorem 13 (with $s = 3$) and the fact that the last blue monomer (the origin of $\ell_{n-7}$) is strictly above all other blue monomers (hence the 180° rotation of $\ell_{n-7}$ proceeds without permanent blocking by blue monomers). The transition for configuration 23 to 24 is valid by applying Lemma 5 (or Theorem 13, with $s = 1$) reflected through a horizontal line that runs through the last blue monomer, and the fact that the last blue monomer (the origin of $\ell_{n-7}$) is strictly below all other blue monomers (hence the 60° rotation of $\ell_{n-7}$ proceeds without permanent blocking). Thus all transitions are valid and the permanently blocked configuration is reachable, giving the lemma statement. ◀

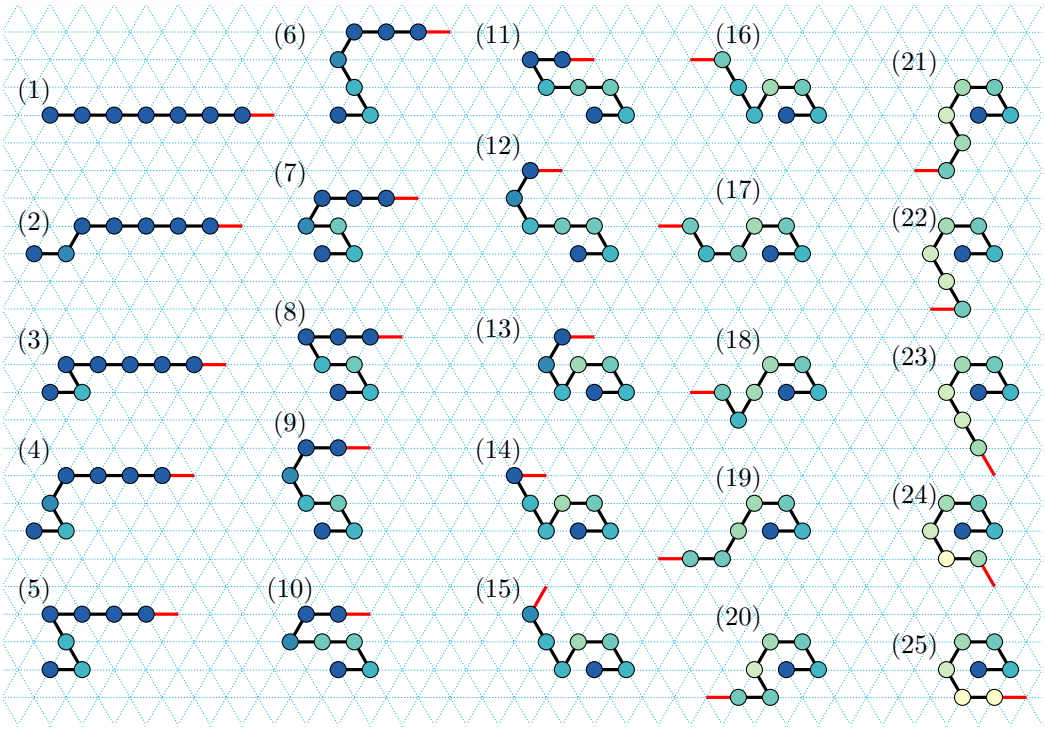



**Figure 7** Impossibility of 360° line rotation, by showing that for all $n \in \mathbb{N}$, the line-rotation Turning Machine $L_n^6$ has a reachable but permanently blocked configuration. Looking at the evolution of the first seven monomers (i.e. ignore the rotation of the red line segment) we see one trajectory of the Turning machine that exhibits permanent blocking in the final (bottom-right) configuration, which has respective states 6,4,3,2,1,0,0. We imagine the red line segment as representing an arbitrary long sequence of monomers running collinear with it, and transitions 14–16, 22–23, and 24–25, each representing the (many step) rotation of the red line by consecutive angles of 60°. These rotations of the red line can proceed by two applications of Theorem 13 (first with $s = 3$, then with $s = 1$) and the fact that the first monomer of the red line is strictly above, or below, the first seven monomers. Hence the final, permanently blocked, configuration is reachable no matter what length the red line is.

## 7 Folding zig-zag squares and $y$-monotone shapes

As a demonstration of our techniques, in this section we show how to build two shapes with Turning Machines: an $n \times n$ square, and any $y$-monotone shape.

We first define a specific curve which fills a square row by row in a zig-zag fashion. An example is shown in Figure 8 (left).

▶ **Definition 16** ($n \times n$ zig-zag square). *For any $n \in \mathbb{N}$, an $n \times n$* zig-zag square *is the length $n^2$ configuration such that the position of monomer $m_i$ is given by the following expression:*

$$\text{pos}(m_i) = \begin{cases} (i\%n, \lfloor \frac{i}{n} \rfloor), & \text{if } i\%(2n) < n, \\ (n-1-i\%n, \lfloor \frac{i}{n} \rfloor), & \text{if } i\%(2n) \geq n, \end{cases}$$

*where $i\%n$ denotes the remainder of $i$ divided by $n$.*

We now show that the zig-zag square can be built by a Turning Machine.

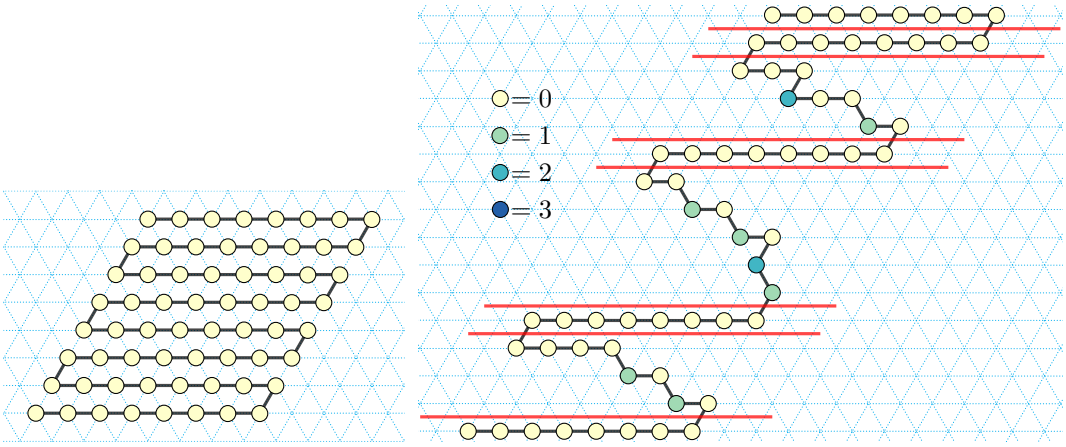



**Figure 8** Left: A target $n \times n$ zig-zag square, for $n = 8$. Right: an intermediate configuration $c$ after all 1-monomers have moved (for $0 \leq k \leq 3$, a $k$-monomer begins in state $k$). The horizontal lines (in red) subdivide the $T_n^{\text{zz}}$ into independent subchains equivalent to $n$ separate line-rotating Turning Machines $L_n^3$.

▶ **Theorem 17.** *For any $n \in \mathbb{N}$, let $T_n^{\text{zz}}$ be an $n^2$-monomer Turning Machine with initial configuration having all monomers positioned on the $x$-axis ($\text{pos}(m_i) = (i, 0) \in \mathbb{Z}^2$) and pointing to the east, with initial state sequence*

$$S = \begin{cases} (0^{n-1}13^{n-1}1)^{\frac{n}{2}-1}0^{n-1}13^{n-1}0, & \text{if } n \text{ is even}, \\ (0^{n-1}13^{n-1}1)^{(n-1)/2}0^n, & \text{if } n \text{ is odd}. \end{cases}$$

*Then, $T_n^{\text{zz}}$ computes the $n \times n$ zig-zag square (Definition 16) in expected time $O(\log n)$.*

**Proof.** For notation, we let $k$-monomers be the monomers whose initial state is $k$. Thus, the Turning Machine consists of sequences of 0- and 3-monomers, separated by single 1-monomers. Observe that all 1-monomers are never blocked. Thus, after expected $O(\log n)$ time they all move to their final orientation along the $y$-axis. Consider such a configuration $c$, in which all 1-monomers are in state 0. The remaining rules can only be applied to 3-monomers. Consider a set of horizontal lines passing through the midpoint of the unit-length line-segment that spans from the position $\text{pos}(m_i)$ of each 1-monomer $m_i$ to $\text{pos}(m_{i+1})$. These lines separate consecutive sequences of 0-monomers and sequences of 3-monomers from one another in the $\mathbb{R}^2$ plane. This implies, that after the two adjacent 1- monomers have moved, the full segment $M$ of 3-monomers in between them moves independently of the rest of the configuration. We claim that the evolution of these processes is modelled by the computation of a line-rotating Turning Machine $L_n^3$. Before its left-bordering 1-monomer has moved, the segment $M$ of 3-monomers acts as a length $n$ instance of $L_{n+1}^3$, with an additional 1-monomer, its first monomer, that simply has not moved yet. Since we know that monomer is first released after $O(\log n)$ time, this does not (asymptotically) change the expected time bound for the $L_{n+1}^3$ machine.

By Theorem 14, each of the sequences of 3-monomers will evolve into their target configuration in $O(\log n)$ expected time independent of one another, which would naively give an overall expected time of $O(\log^2 n)$ time. However, by Lemma 11 we know that no 3-monomer that is in state 3 or state 2, and no 1-monomer, is ever blocked. Hence, we can

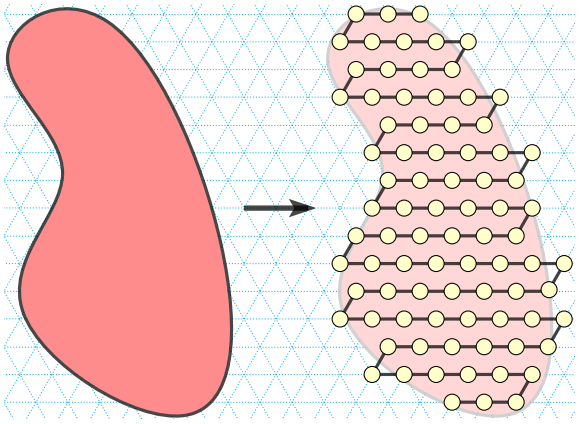


**Figure 9** A $y$-monotone shape in $\mathbb{R}^2$ approximated with a zig-zag chain on the triangular grid.

analyse all $n^2$ monomers as one system, noting that all such monomers complete in time $O(\log n)$, at which point we have a reachable configuration that has all 3-monomers in either state 0 and 1 (all others in state 0) which in turn finishes in $O(\log n)$ expected time.[7] ◀

▶ **Definition 18** ($y$-monotone shape). *A set $A \subset \mathbb{R}^2$ is $y$-monotone, if any horizontal line $h$ intersects $S$ along one continuous segment of $h$.*

Similarly to the construction of the zig-zag square presented above, we can build an approximation of any $y$-monotone shape $A$ by discretizing it and filling the resulting shape row by row in a zig-zag manner (refer to Figure 9). The resulting state sequence of the Turning Machine $T_n^{\text{zz}}$ consists of intervals of 0-monomers and 3-monomers of various lengths separated by single 1-monomers.

We conclude with the following theorem statement. In it we assume that the state sequence $S$ is such that the final configuration approximates some given $y$-monotone shape $A$. The proof is the same as that for Theorem 17 (but using a variety of horizontal segment lengths $n$).

▶ **Theorem 19.** *Let $T_n^{y-\text{mon}}$ be a Turning Machine with initial configuration having all monomers positioned on the $x$-axis ($\text{pos}(m_i) = (i, 0) \in \mathbb{Z}^2$) and pointing to the east, with initial state sequence $S$ consisting of intervals of 0- and 3-monomers separated by single 1-monomers. Then $T_n^{y-\text{mon}}$ computes its target configuration in $O(\log n)$ expected time.*

## References


1 Greg Aloupis, Sébastien Collette, Mirela Damian, Erik D Demaine, Robin Flatland, Stefan Langerman, Joseph O'Rourke, Suneeta Ramaswami, Vera Sacristán, and Stefanie Wuhrer. Linear reconfiguration of cube-style modular robots. *Computational Geometry*, 42(6-7):652–663, 2009.

2 Greg Aloupis, Sébastien Collette, Erik D. Demaine, Stefan Langerman, Vera Sacristán, and Stefanie Wuhrer. Reconfiguration of cube-style modular robots using O(log n) parallel moves. In *International Symposium on Algorithms and Computation*, pages 342–353. Springer, 2008.


[7] This is similar to the technique used in the proof of Theorem 14.

**3** Ho-Lin Chen, David Doty, Dhiraj Holden, Chris Thachuk, Damien Woods, and Chun-Tao Yang. Fast algorithmic self-assembly of simple shapes using random agitation. In *DNA20: The 20th International Conference on DNA Computing and Molecular Programming*, volume 8727 of *LNCS*, pages 20–36, Kyoto, Japan, September 2014. Springer. Full version: `arXiv:1409.4828`.

**4** Moya Chen, Doris Xin, and Damien Woods. Parallel computation using active self-assembly. *Natural Computing*, 14(2):225–250, 2014. arXiv version: `arXiv:1405.0527`.

**5** Kenneth C. Cheung, Erik D. Demaine, Jonathan R. Bachrach, and Saul Griffith. Programmable assembly with universally foldable strings (moteins). *IEEE Transactions on Robotics*, 27(4):718–729, 2011.

**6** Yen-Ru Chin, Jui-Ting Tsai, and Ho-Lin Chen. A minimal requirement for self-assembly of lines in polylogarithmic time. *Natural Computing*, 17(4):743–757, 2018.

**7** Robert Connelly, Erik D. Demaine, Martin L. Demaine, Sándor P. Fekete, Stefan Langerman, Joseph S.B. Mitchell, Ares Ribó, and Günter Rote. Locked and unlocked chains of planar shapes. *Discrete & Computational Geometry*, 44(2):439–462, 2010.

**8** Nadine Dabby and Ho-Lin Chen. Active self-assembly of simple units using an insertion primitive. In *SODA: The 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1526–1536, January 2012.

**9** Rachel E. Dawes-Hoang, Kush M. Parmar, Audrey E. Christiansen, Chris B. Phelps, Andrea H. Brand, and Eric F. Wieschaus. Folded gastrulation, cell shape change and the control of myosin localization. *Development*, 132(18):4165–4178, 2005.

**10** Erik D. Demaine, Jacob Hendricks, Meagan Olsen, Matthew J. Patitz, Trent A. Rogers, Nicolas Schabanel, Shinnosuke Seki, and Hadley Thomas. Know when to fold'em: self-assembly of shapes by folding in oritatami. In *DNA: International Conference on DNA Computing and Molecular Programming*, pages 19–36. Springer, 2018.

**11** Cody Geary, Pierre-Étienne Meunier, Nicolas Schabanel, and Shinnosuke Seki. Programming biomolecules that fold greedily during transcription. In *MFCS: The 41st International Symposium on Mathematical Foundations of Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

**12** Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Fabian Kuhn, Dorian Rudolph, Christian Scheideler, and Thim Strothmann. Forming tile shapes with simple robots. *Natural Computing*, pages 1–16, 2019.

**13** Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. Concrete mathematics, 1989.

**14** Benjamin Hescott, Caleb Malchik, and Andrew Winslow. Tight bounds for active self-assembly using an insertion primitive. *Algorithmica*, 77:537–554, 2017.

**15** Benjamin Hescott, Caleb Malchik, and Andrew Winslow. Non-determinism reduces construction time in active self-assembly using an insertion primitive. In *COCOON: The 24th International Computing and Combinatorics Conference*, pages 626–637. Springer, 2018.

**16** Chun-Ying Hou and Ho-Lin Chen. An exponentially growing nubot system without state changes. In *International Conference on Unconventional Computation and Natural Computation*, pages 122–135. Springer, 2019.

**17** Adam C Martin, Matthias Kaschube, and Eric F Wieschaus. Pulsed contractions of an actin–myosin network drive apical constriction. *Nature*, 457(7228):495–499, 2008.

**18** Othon Michail, George Skretas, and Paul G. Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019.

**19** Hamid Ramezani and Hendrik Dietz. Building machines with DNA molecules. *Nature Reviews Genetics*, pages 1–22, 2019.

**20** Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *ITCS: The 4th conference on Innovations in Theoretical Computer Science*, pages 353–354. ACM, 2013. Full version: `arXiv:1301.2626` [cs.DS].

# A Line rotation by $\pi/3$, $\pi$ and $4\pi/3$

In this appendix we present proofs that line-rotating Turning Machine for respective angles of $\pi/3$, $\pi$ and $4\pi/3$ terminates in expected time $O(\log n)$. These claims are superseded by the results in the main paper, but we include the proofs as they give a number of techniques to analyse the Turning Machine model.

## A.1 Line rotation by $\pi/3$: $L_n^1$

The following proof of line rotation by $\pi/3$ radians is intended to be a simple example worked out in detail. Let $L_n^1$ be the Turning Machine defined in Definition 2 with $\sigma = 1$, as illustrated in Figure 3 (left).

▶ **Lemma 5.** *For each $n \in \mathbb{N}$, the line-rotating Turning Machine $L_n^1$ computes its target configuration, and does so in expected $O(\log n)$ time.*

**Proof.** The initial configuration (Figure 3, left) of $L_n^1$ is a line of $n-1$ monomers in state 1 with an additional final monomer in state 0, i.e. at time 0 the $n$ states are $s(m_0)s(m_1)\cdots s(m_{n-1}) = 1^{n-1}0$. Since monomer states only change by decrementing from 1 to 0, any configuration on any trajectory of $L_n^1$ has its (composite) state of the form $\{0,1\}^{n-1}0$. Consider a configuration $c$ in a trajectory of evolution of $L_n^1$, and the corresponding state[8] $x \in \{0,1\}^{n-1}0$. Let $m_i^c$ denote the $i$th monomer of $L_n^1$ in configuration $c$. For any $i \in \{0,1,\ldots,n-2\}$ such that $s(m_i^c) = 1$, consider the unique configuration $c'$ where $c \to c'$ and $s(m_i^{c'}) = 0$ (and, by definition of next configuration step, $j \neq i$ implies $s(m_j^{c'}) = s(m_j^c)$).

We claim that $\text{tail}(m_i^c)$ does not share any positions with $\text{head}^{\rightarrow}(m_i^c)$, in other words, that $c'$ is a non-self-intersecting configuration. To show this, consider a horizontal line $\ell_i$ through monomer $m_i^c$ and observe that in $c'$ (and in $c$), the monomers $\text{tail}(m_i^c) = m_0^c, m_1^c, \ldots, m_i^c$ lie on or below $\ell_i$ (because the path $\text{pos}(m_0^c), \text{pos}(m_1^c), \ldots, \text{pos}(m_i^c)$ is connected and consists of unit length segments each at an angle of either $0°$ or $60°$ clockwise relative to the $x$-axis), but the monomers $\text{head}^{\rightarrow}(m_i^c) = m_{i+1}^{c'}, m_{i+2}^{c'}, \ldots, m_{n-1}^{c'}$ lie strictly above $\ell_i$ (because $\text{pos}(m_{i+1}^{c'})$ is strictly higher than $\text{pos}(m_i^c)$, and because the path $\text{pos}(m'_{i+1}), \text{pos}(m'_{i+2}), \ldots, \text{pos}(m_{n-1}^{c'})$ is connected and consists of unit length segments each at an angle of $0°$ or $60°$ to the $x$-axis). Hence there are no blocked configurations reachable by $L_n^1$ (neither permanent nor temporary blocking).

At each reachable configuration $c$, starting from the initial configuration, we can choose $i$ independently from the set of non-zero states. The expected time for the first rule application is $1/(n-1)$ since it is the expected time of the minimum of $n-1$ independent exponential random variables each with rate 1. The next is $1/(n-2)$, and so on. By linearity of expectation, the expected value of the total time $T$ is $E[T] = \sum_{k=1}^{n-1} \frac{1}{k} = O(\log n)$, where the sum is the $(n-1)^{\text{th}}$ partial sum of the harmonic series, known to have a $O(\log n)$ bound. Hence $L_n^1$ completes in expected $O(\log n)$ time. ◀

## A.2 Line rotation by $\pi$: $L_n^3$

Next, we analyse line rotation of $\pi$ radians.

▶ **Lemma 20.** *Let $L_n^3$ be a line-rotating Turning machine, then:*

**(i)** *any reachable configuration of $L_n^3$ has no more than $2n/3$ blocked monomers, and*

**(ii)** *there exists a configuration of $L_n^3$ that has exactly $2n/3$ blocked monomers.*

[8] In fact any $x \in \{0,1\}^{n-1}0$ is the state of a reachable configuration, but we don't need to prove that.

**Proof.** Consider any reachable configuration $c$ of $L_n^3$, and let monomer $m_i$ be blocked in $c$. By Lemma 11, monomers in state 2 and 3 are never blocked. By definition, monomers in state 0 are not blocked. Thus if $m_i$ is blocked it is in state 1, i.e. $s(m_i) = 1$. We claim that in this case either $s(m_{i-1}) = 3$ or $s(m_{i+1}) = 3$ (or both). Consider the following two cases for $s(m_{i+1})$:

1. If $s(m_{i+1}) \in \{1, 2\}$, then by Lemma 6 all monomers of head$^{\rightarrow}(m_i)$, except its first monomer $m'_{i+1}$, lie strictly above $\ell_i$, and since tail$(m_i)$ lies on or below $\ell_i$, we get that tail$(m_i)$ does not intersect head$^{\rightarrow}(m_i)$, except possibly at pos$(m'_{i+1})$. Whether pos$(m'_{i+1})$ intersects tail$(m_i)$ depends on the state of $m_{i-1}$:
   - **(a)** If $s(m_{i-1}) \in \{1, 2\}$, then all monomers of tail$(m_i)$ lie strictly below $\ell_i$ (except its first monomer $m_i$ which is not at position pos$(m'_{i+1})$), hence pos$(m'_{i+1})$ cannot intersect tail$(m_i)$. Then $m_i$ cannot be blocked.
   - **(b)** If $s(m_{i-1}) = 0$, then $m'_{i+1}$ does not intersect tail$(m_i)$: Indeed, pos$(m_{i-1}) =$ pos$(m_i) + \vec{x} =$ pos$(m'_{i+1}) + 2\vec{x} \neq$ pos$(m'_{i+1})$. Furthermore, let $m_j$, $m_{j+1}$, ..., $m_{i-1}$ be the longest consecutive subsequence of monomers in state 0 preceding monomer $m_i$. Then pos$(m_j)$, pos$(m_{j+1})$, ..., pos$(m_{i+1})$ are all strictly to the west of pos$(m_i)$. If $j - 1 \geq 0$, the non-zero-state[9] monomer $m_{j-1}$ enforces that the monomers $m_0$, $m_1$, ..., $m_{j-1}$ lie strictly below $\ell_i$. Thus $m_i$ is not blocked.

   Therefore, monomer $m_{i-1}$ can only be in state 3.
2. If $s(m_{i+1}) = 0$: Both head$^{\rightarrow}(m_i)$ and tail$(m_i)$ have monomers on $\ell_i$, but we claim the positions of head$^{\rightarrow}(m_i)$ do not intersect those of tail$(m_i)$. If $s(m_{i-1}) \in \{1, 2\}$, then all monomers of tail$(m_i)$ except $m_i$ lie strictly below $\ell_i$, and thus head$^{\rightarrow}(m_i)$ does not intersect tail$(m_i)$ (and recall that head$^{\rightarrow}(m_i)$ does not intersect pos$(m_i)$ because configurations are simple). If $s(m_{i-1}) = 0$ then the monomers $M = \{m_{i-1}, m_i, m'_{i+1}\}$ lie along $\ell_i$ (pointing west). Note that a prefix of $M$ is a suffix of tail$(m_i)$ and a (disjoint) suffix of $M$ is a prefix of head$^{\rightarrow}(m_i)$. Hence, in order for tail$(m_i)$ to intersect head$^{\rightarrow}(m_i)$, one or both must depart from $\ell_i$, but, by Lemma 6, tail$(m_i)$ can only do so by having monomers strictly below $\ell_i$, and head$^{\rightarrow}(m_i)$ can only do so by having monomers strictly above $\ell_i$. Thus, monomer $m_{i-1}$ can only be in state 3.

Therefore, if $m_i$ is blocked, then either $m_{i-1}$ or $m_{i+1}$ is in state 3, and thus is unblocked. Hence, there cannot be three monomers in a row which are blocked, resulting in Conclusion (i) of the lemma.

For Conclusion (ii), consider a line-rotating Turning Machine $L_n^3$ with $n = 3k$ for some $k$. The configuration $c$ with state sequence $S = (131)^{k-1}130$ has exactly $2n/3$ blocked monomers, as every monomer in state 1 is either blocked by a preceding monomer in state 3, or by a following monomer in state 3. ◀

▶ **Theorem 21** (Rotate a line by $\pi$)**.** *For each $n \in \mathbb{N}$, the line-rotating Turning Machine $L_n^3$ computes its target configuration, and does so in expected time $O(\log n)$.*

**Proof.** By Lemma 20, no configuration has a permanently blocked monomer, hence every trajectory of $L_n^3$ ends in the target configuration.

At the initial step, the rate of rule applications is $n - 1$ (there are $n - 1$ monomers in state 3). Over time, for successive configurations along a trajectory, the rate of rule applications may decrease for two reasons: (a) some monomers may be temporary blocked, and (b) after a monomer transitions to state 0 no more rules are applicable to it. We reason about both:

[9] Which must be in state 1 or 2, since 3 would give a self-intersection along the configuration.

(a) Lemma 20(ii) shows that a configuration with state sequence $s = (131)^{n/3-1}130$ has $2n/3$ blocked monomers, and Lemma 20(i) states that no configuration has more than $2n/3$ blocked monomers for $n$ divisible by 3. Using that fact, and in order to simplify the proof, we shall analyse a new, possibly slower, system where for any configuration $c$ that has $n' \leq n$ monomers in state $\neq 0$, we "artificially" block $2n'/3$ monomers.[10] Since this assumption merely serves to slow the system, it is sufficient to give an upper bound on the expected time to finish.

(b) A second "slowdown" assumption will be applied during the analysis and is justified as follows. Intuitively, the number of monomers transitioning to state 0 increases with time, and since monomers in state 0 have no applicable rules, this causes a *decrease* in the rate of rule applications. Consider a hypothetical continuous-time Markov system $M$, with $3n$ steps with rate decreasing by 1 every third step, that is, with successive rates $n, n, n, n-1, n-1, n-1, n-2, \ldots, 2, 1, 1, 1$. By linearity of expectation, the expected value of the finishing time $T$ is the sum of the expected times $E[t_i]$ for each of the individual steps $i \in \{1, 2, \ldots, 3n\}$:

$$E[T] = \sum_{i=1}^{3n} E[t_i] = \sum_{m=1}^{n} 3 \cdot \frac{1}{m} = 3 \sum_{m=1}^{n} \frac{1}{m} = 3H_n \leq 3(\ln(n) + 1) = O(\log n)\,, \tag{3}$$

where $H_n$ is the $n^{th}$ partial sum of the harmonic series $\sum_{m=1}^{\infty} \frac{1}{m}$ with $H_n \leq \ln(n) + 1$ (see [13]). Since, in $L_n^3$, it requires at least 3 steps to send a monomer from state 3 (the initial state) to state 0, no trajectory sends monomers to state 0 at a faster rate than a (hypothetical) trajectory where a transition to state 0 appears at every third configuration (step). Hence, if there were no blocking whatsoever, then the expected time for $L_n^3$ would be no larger than $3H_n$ (given by Equation (3)).

Taking the blocking "slowdown assumption" in (a) into account, if the rate at step $i$ is $r_i$, then the slowed down rate is $\frac{1}{3}r_i$ giving an expected time of

$$E[T] = \sum_{i=1}^{3n} E[t_i] = \sum_{m=1}^{n} 3 \cdot \frac{3}{1} \cdot \frac{1}{m} = 9 \sum_{m=1}^{n} \frac{1}{m} = 9H_n \leq 9(\ln(n) + 1) = O(\log n)\,. \tag{4}$$

Since our two assumptions merely serve to define a new system that is necessarily slower than $L_n^3$, we get the claimed expected time upper bound of $O(\log n)$ for $L_n^3$. ◀

## A.3 Line rotation by $4\pi/3$: $L_n^4$

▶ **Lemma 22.** *Let $m_i$ be a blocked monomer in some reachable configuration $c$ of a line rotation Turning Machine $L_n^s$ with $n \in \mathbb{N}$ and $1 \leq s \leq 4$, and let $m_j \in \text{head}(m_i)$ and $m_k \in \text{tail}(m_i)$ be a pair of monomers which block the movement of $m_i$, then in the subchain of $L_n^s$ from $m_k$ to $m_{j-1}$ the number of unblocked monomers is at least half the number of blocked monomers.*

**Proof.** Similarly to the proof of Lemma 11, consider the closed chain $P = \text{pos}(m_k)$, ..., $\text{pos}(m_j)$, $\text{pos}(m_k)$. Let $x(m_i)$ denote the $x$-coordinate of the position of monomer $m_i$, and $y(m_i)$ denote the $y$-coordinate of the position of monomer $m_i$. Note, that for any $\ell$,

[10] The monomers are not necessarily geometrically blocked, we are merely stopping any rule from being applied to them. No configuration in a trajectory of $L_n^3$ witnesses a larger slowdown due to blocking than the slowdown we have imposed on the configurations of $T_n'$.

- if $s(m_\ell) = s$, then $x(m_{\ell+1}) = x(m_\ell) + 1$ and $y(m_{\ell+1}) = y(m_\ell)$,
- if $s(m_\ell) = s-1$, then $x(m_{\ell+1}) = x(m_\ell)$ and $y(m_{\ell+1}) = y(m_\ell) + 1$,
- if $s(m_\ell) = s-2$, then $x(m_{\ell+1}) = x(m_\ell) - 1$ and $y(m_{\ell+1}) = y(m_\ell) + 1$,
- if $s(m_\ell) = s-3$, then $x(m_{\ell+1}) = x(m_\ell) - 1$ and $y(m_{\ell+1}) = y(m_\ell)$,
- if $s(m_\ell) = s-4$, then $x(m_{\ell+1}) = x(m_\ell)$ and $y(m_{\ell+1}) = y(m_\ell) - 1$.

Let $x(m_k) - x(m_j) = \varepsilon_x$ and $y(m_k) - y(m_j) = \varepsilon_y$, with $\varepsilon_x, \varepsilon_y \in \{-1, 0, 1\}$. The total change in $x$-coordinate and the total change in $y$-coordinate, when traversing $P$, is zero, that is,

$$\begin{aligned} &\sum_{\ell=k}^{j-1} (x(\ell+1) - x(\ell)) + \varepsilon_x = 0\,, \\ &\sum_{\ell=k}^{j-1} (y(\ell+1) - y(\ell)) + \varepsilon_y = 0\,. \end{aligned} \tag{5}$$

Considering the first part of Equation (5), and taking into account that the $x$-coordinate increases only when traversing monomers in state $s$, and the $x$-coordinate decreases only when traversing monomers in state $s-2$ or $s-3$, we get $\#(s) + \varepsilon_x = \#(s-2) + \#(s-3)$, where $\#(u)$ denotes the number of monomers with state $u$ in the subchain from $m_k$ to $m_{j-1}$. Observe, by Lemma 11, monomers in states $s$ and $s-1$ cannot be blocked, and since $s \leq 4$, only the monomers in states $s-2$ or $s-3$ can be blocked. This implies, that within the subchain from $m_k$ to $m_{j-1}$, the number of blocked monomers is at most within an additive factor 1 from the number of unblocked monomers.

Suppose, for a given subchain from $m_k$ to $m_{j-1}$, the number of monomers in state $s$ is strictly positive (that is, $\#(s) \geq 1$). Then, $\#(s) \geq \frac{1}{2}(\#(s-2) + \#(s-3))$, that is, in the subchain, the number of unblocked monomers is at least half the number of blocked monomers.

Now suppose that the number of monomers in state $s$ in the subchain is zero (that is, $\#(s) = 0$). As the blocked monomer $m_i$ has state either $s-2$ or $s-3$, the $x$-coordinate decreases by 1 when traversing it. The $x$-coordinate only increases when traversing monomers in state $s$. Therefore, if there are no monomers in state $s$, $\varepsilon_x$ has to be 1, and, besides the blocked monomer $m_i$, the subchain from $m_k$ to $m_{j-1}$ consists only of monomers in states $s-4$ and $s-1$.

Furthermore, as $\varepsilon_x = 1$, we have that $pos(m_k) = pos(m_j) - \vec{w}$ (that is, $m_i$ is in state $s-3$). We claim that there is at least one monomer in state $s-1$ in the subchain from $m_k$ to $m_{j-1}$. Indeed, consider the second part of Equation|5. Traversing the edge between monomers $m_k$ and $m_j$ changes the $y$-coordinate by $\varepsilon_y = y(m_j) - y(m_k) = y(-\vec{w}) = -1$. Thus there has to be at least one monomer traversing which increases the $y$-coordinate. This can only be a monomer in state $s-1$. Thus, in the subchain from $m_k$ to $m_{j-1}$, there is one blocked monomer $m_i$ and at least one unblocked monomer in state $s-1$, and the total number of unblocked monomers is at least the number of blocked monomers. ◀

▶ **Theorem 23.** *For each $n \in \mathbb{N}$ and $1 \leq s \leq 4$, the line rotation Turning Machine $L_n^s$ computes its target configuration in $O(\log n)$ steps.*

**Proof.** By Theorem 13 the Turning Machine $L_n^s$ computes its target configuration. That it computes the target configuration in $O(\log n)$ steps follows from the claim that in any intermediate configuration $c$, the number of blocked monomers is not greater than $3n/4$.

To prove this claim, consider a reachable configuration $c$ of $L_s^n$, and consider all blocked monomers $B = \{m_i : m_i \text{ is blocked}\}$. Let $e_{j,k}$ be the edge connecting the positions of two monomers $m_j$ and $m_k$ which block the movement of some monomer $m_i \in B$ (note, that $m_i$

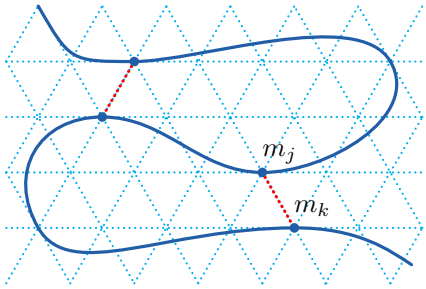



**Figure 10** Subdivision $D'$ of the plane consists of chain $L_s^n$ (shown in blue), and all edges (shown in red), connecting pairs of monomers blocking the movement of some monomer, such that these edges are incident to the outer face of $D'$.

can be blocked by more than one pair of monomers). Let $E = \{e_{j,k}\}$ be the set of all such edges for all pairs $m_j$ and $m_k$ which block some monomer in $L_s^n$. Observe, that no two edges in $E$ cross each other, as they are unit segments in the triangular graph, and for the same reason no edge in $E$ crosses the chain $L_s^n$. Let the chain $L_s^n$ together with the set of edges $E$ partition the plain into plane subdivision $D$ (refer to Figure 10). The bounded faces of $D$ are formed of subchains of $L_s^n$ and edges from $E$. Now, remove the edges of $E$ from $D$ which are not incident to the outer face, resulting in a plane subdivision $D'$. In it, every bounded face is formed by a single subchain of $L_s^n$ and a single edge from $E$.

Observe, that all monomers of $L_s^n$ which are blocked are incident to at least one bounded face. Otherwise, there would be two monomers $m_j$ and $m_k$ blocking the move with the edge $e_{j,k}$ not in $E$, thus contradicting the definition of $E$.

For each bounded face $f_i$ in $D'$, by Lemma 22, we have $\#_i(\text{unblocked}) \geq \frac{1}{2}\#_i(\text{blocked})$, where $\#_i(\text{unblocked})$ denotes the number of unblocked monomers incident to the face $f_i$, and $\#_i(\text{blocked})$ denotes the number of blocked monomers incident to the face $f_i$.

Note, that each unblocked monomer can be incident to at most two bounded faces of $D'$, and recall that each blocked monomer is incident to at least one bounded face of $D'$. Then,

$$\#(\text{unblocked}) \geq \frac{1}{2}\sum_{f_i \in D'} \#_i(\text{unblocked}) \geq \frac{1}{2}\left(\frac{1}{2}\sum_{f_i \in D'} \#_i(\text{blocked})\right) \geq \frac{1}{4}\#(\text{blocked}),$$

where the sums are over the bounded faces of $D'$, and $\#(\text{unblocked})$ denotes the total number of unblocked monomers in $L_s^n$, and $\#(\text{blocked})$ denotes the total number of blocked monomers in $L_s^n$.

Since there is a constant fraction of unblocked monomers in any configuration, the total expected time it takes $L_n^s$ to compute its target configuration is $O(\log n)$. ◀