8-1-2024

Intrinsic Universality in Tile Automata and Related Results

Elise C. Grizzell
The University of Texas Rio Grande Valley

INTRINSIC UNIVERSALITY IN TILE AUTOMATA AND RELATED RESULTS

A Thesis

by

ELISE GRIZZELL

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: Computer Science

The University of Texas Rio Grande Valley

August 2024

INTRINSIC UNIVERSALITY IN TILE AUTOMATA AND RELATED RESULTS

A Thesis
by
ELISE GRIZZELL

COMMITTEE MEMBERS

Dr. Tim Wylie
Chair of Committee

Dr. Robert Schweller
Committee Member

Dr. Bin Fu
Committee Member

Dr. Qi Lu
Committee Member

August 2024

ABSTRACT

Grizzell, Elise, Intrinsic Universality in Tile Automata and Related Results. Master of Science (MS), August, 2024, 235 pp., 1 table, 37 figures, 30 references.

The Tile Automata (TA) model describes self-assembly systems in which monomers can build structures and transition with an adjacent monomer to change their states. This paper shows that seeded TA is a non-committal intrinsically universal model of self-assembly. We present a single universal Tile Automata system containing approximately 4600 states that can simulate (a) the output assemblies created by any other Tile Automata system Γ, (b) the dynamics involved in building Γ's assemblies, and (c) Γ's internal state transitions. It does so in a non-committal way: it preserves the full non-deterministic dynamics of a tile's potential attachment or transition by selecting its state in a single step, considering all possible outcomes until the moment of selection.

The system uses supertiles, each encoding the complete system being simulated. The universal system builds supertiles from its seed, each representing a single tile in Γ, transferring the information to simulate Γ to each new tile. Supertiles may also asynchronously transition states according to the rules of Γ. This result directly transfers to a restricted version of asynchronous Cellular Automata: pairwise Cellular Automata.

DEDICATION

To my mother for seeing me through the darkest times and supporting my successes today.

ACKNOWLEDGMENTS

First and most importantly, I want to thank my parents for all their love and support. Especially my mother, Dr. Saara Grizzell. Without her, I never would have come to UTRGV and had the chance to pursue research.

None of this would have been possible without my advisors Tim Wylie, Robert Schweller, and Bin Fu. Wylie, I will always be grateful that you recruited me into research and have been such an incredible advisor. Schweller your guidance and assistance have been invaluable. Dr. Fu, not only have you been an asset to my research, but you also awarded me the G.A.A.N.N. Grant, enabling me to do research full-time. I could not be more grateful.

There were incredible fellow students along the way. Tim, you were everything I could have asked for in a colleague and friend and more. Thank you for sharing your brilliance, mentorship, and support. Micheal, not only were you a significant support and friend in the lab you also spearheaded the creation of the tool that made this thesis possible. This wouldn't have been possible without either of you. Andrew, your calm rationalism, great ideas, and solid friendship were one of the high points of my time at ASARG. Ryan, you've been a fantastic co-author and rubiks cube competitor.

Tom, you're the best cheerleader a woman could have. I will always be grateful for your love, support, and substantial rewrite of the attachment section.

I have been lucky to work with the following other wonderful co-authors: Rachel Anderson, Alberto Avila, Josh Brunner, David Caballero, Sonya C. Cirlos, Michael Coulombe, Erik D. Demaine, Jenny Diomidova, Markus Hecher, Austin Luchsinger, Jayson Lynch, Aiden Massie, Gourab Mukhopadhyay, Adrian Salinas, Ahmed Shalaby, Armando Tenorio, Evan Tomai, and Damien Woods.

I am so grateful for every member of the Algorithmic Self-Assembly Research group and Computer Science department for creating a supportive and friendly environment.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The molecular programming field is a new frontier of research in which we explore the possibilities of using molecules to compute. In particular the area of the field known as self-assembly studies how individual monomers may come together to form larger structures.

My thesis primarily focuses on the topic of Intrinsic Universality in the self-assembly model Tile Automata but also covers other work in the aTAM, Chemical Reaction Networks, and surface Chemical Reaction Networks models. The majority of results are in the design of an intrinsically universal tile set.

Shared Work. All of this work was done in conjunction with Dr. Robert Schweller and Dr. Timothy Wylie. Additionally, I have worked with other authors in these papers such as Micheal Alaniz, Rachel Anderson, Alberto Avila, Josh Brunner, David Caballero, Sonya C. Cirlos, Michael Coulombe, Erik D. Demaine, Jenny Diomidova, Bin Fu, Tim Gomez, Markus Hecher, Ryan Knobel, Jayson Lynch, Aiden Massie, Gourab Mukhopadhyay, Tom Peters, Andrew Rodriguez, Adrian Salinas, Armando Tenorio, and Evan Tomai.

1.1 Overview

Chapter II includes the current manuscript of the Intrinsic Universality in Active Tile Self-Assembly paper which is set to be submitted to a conference within the next month, as such it is currently incomplete and a final version will be uploaded with the final manuscript of this thesis. Here, we build an intrinsically universal tile set for the seeded Tile Automata model. Chapter III covers other work in the Tile Automata model. Chapter IV covers the construction of covert tile

assembly computers using the abstract Tile Assembly Model (aTAM). Next, Chapter V covers Chemical Reaction Networks (CRNs) particularly. Finally, VI covers my work in the surface Chemical Reaction Networks model.

CHAPTER II

INTRINSIC UNIVERSALITY IN ACTIVE TILE SELF-ASSEMBLY

The concept of intrinsic universality is that we not only want to simulate computation to receive the output of another system but also do what the system we are simulating does in the way that it does it. In the case of tile self-assembly this means having a single universal tile set that can not only build anything any other tile set can build but also with the same construction process. To do so we use *supertiles*, built from the universal tile set and programmed with the necessary information to simulate another tile set or itself. We store that information, the *affinities* and *transitions* of the other system, inside of a lookup table within the supertile.

My Contributions. I worked on this project alone for two years. During my first year on this project, I created an initial overview of a supertile and attempted to program those gadgets into AutoTile, spending a substantial number of hours doing so; however, I had to return to the drawing board on many of the gadgets later when I began to actually write the paper. In terms of the initial design, outside of the state transmission wire configuration, I did all of the work including macrocells, active state column, table door edge, doors, the exact function of the lookup table, how affinities are stored and transmitted, etc. I wrote the rough drafts of the abstract, introduction, temperature simulation, attachment, transition, metrics, and conclusion sections. I did assist in the writing of the seeded results and asynchronous cellular automata sections. I also wrote overviews for several sections in preliminaries as well as compiled and merged them from the [5] and [20] papers. I made the original versions of all of the figures and counted states. I also did the previous work reading and large portions of the original version of the table. I also coordinated the later work

with my co-authors and conducted meetings.

Abstract. The Tile Automata (TA) model describes self-assembly systems in which monomers can build structures and transition with an adjacent monomer to change their states. This paper shows that seeded TA is a non-committal intrinsically universal model of self-assembly. We present a single universal Tile Automata system containing approximately 4600 states that can simulate (a) the output assemblies created by any other Tile Automata system Γ, (b) the dynamics involved in building Γ's assemblies, and (c) Γ's internal state transitions. It does so in a non-committal way: it preserves the full non-deterministic dynamics of a tile's potential attachment or transition by selecting its state in a single step, considering all possible outcomes until the moment of selection.

The system uses supertiles, each encoding the complete system being simulated. The universal system builds supertiles from its seed, each representing a single tile in Γ, transferring the information to simulate Γ to each new tile. Supertiles may also asynchronously transition states according to the rules of Γ. This result directly transfers to a restricted version of asynchronous Cellular Automata: pairwise Cellular Automata.

2.1 Introduction

Tile self-assembly is a model that attempts to exploit the computational capabilities of nucleic acids. DNA molecules can form complex structures, and in controlling the growth of those structures, we can utilize their powers to perform computations. In recent years, a diverse set of new abstractions and models have been conceived, the most prominent of which has been the (two-dimensional) *abstract Tile Assembly Model* (aTAM) [28]. In this model, a *tile* is a non-rotatable unit square with specified *glues* on each side, modeling a single monomer. Two tiles can attach if their glues match. A *tile assembly system* is a set of these tile types and a temperature τ. Research into these models usually revolves around the types of *assemblies* that can be created with specific sets of tile types.

In this paper, we work in a related model, derived by combining elements of tile self-assembly and the local state changes of asynchronous Cellular Automata: *seeded Tile Automata (TA)* [5]. A Tile Automata *system* Γ has a set of *states* Σ. These states contain no glues, contrary to the aTAMs tile types. Instead, tiles with an *initial state* $\sigma \in \Lambda$ ($\Lambda \subseteq \Sigma$) can attach to the *seed s* if the system contains an *affinity rule* for their respective tile types that has an equal or higher strength than the system *temperature* τ. Should a single pair of tiles lack sufficient strength to bind to the assembly, they may bind *cooperatively* by adding the strengths of affinities of neighboring tiles to reach τ. Contrary to the *passive* aTAM, tiles in the *active* TA system can change their state. More restricted than most Cellular Automata systems, only *two* tiles directly adjacent to one another can transition their states if the system contains the corresponding *transition rule*.

Here, we study the creation of an *intrinsically universal* (IU) Tile Automata system Γ_U, a system with a finite state set capable of creating not only the final assemblies of any other arbitrary Tile Automata system Γ but also replicating the exact assembly process and any additional computations achieved via transitions. Our universal tile assembly system can simulate systems that contain more states than Σ_U does and even simulate itself. To do this, we sacrifice scale. We

use many tiles to create a *supertile*, that simulates a single tile in Γ.

In this paper, we show that *non-committal intrinsic universality* is impossible in any *passive* system, such as the aTAM. This means that the dynamics of attachment and transitions of a tile assembly system cannot be faithfully simulated by achieving the final determinations of each in a single step. Instead, they are *committal* intrinsically universal, meaning that they need multiple attachment and or transition steps to replicate the decision process of a single step in the target system. On first sight, this appears to contradict previous work showing the aTAM is intrinsically universal [8]. However, that paper contained a subtle error which was later addressed by making the definition of intrinsic universality (IU) slightly weaker [20]. We will refer to this weakened version as committal IU. Besides our negative result, we show that the seeded Tile Automata model with its infinite state changes is, in fact, non-committal intrinsically universal, using approximately 4600 states.

Intrinsic universality is motivated by creating a universal tile set small enough to be stored in a lab refrigerator for real-world experimentation. Although 4600 tiles is still a large number of states and is not optimal, 4600 tiles is about ten million tile types less than the previously stated committal intrinsic universality result for two-dimensional aTAM [8]. Importantly, our initial state set Λ is only a single tile type. While current laboratory capabilities lag the ability to implement this universal tile set as of today, there have been recent advancements in for example the ability to replace tiles experimentally [26, 27] and in the aTAM a tile set capable of universal 6-bit computation was created [29]. The aTAM has also been proven to be intrinsically universal in 3D [13], and synchronous Cellular Automata have been shown to be intrinsically universal in 1D, 2D, and 3D [2, 11, 19].

The question of whether 2D asynchronous Cellular Automata is intrinsically universal is currently open, though work towards a 1D version has been done [30]. Tile Automata can be viewed as a restricted version of asynchronous Cellular Automata in which the neighborhood size is 2, the radius is 1, the system is non-deterministic, and the updating is asynchronous. Therefore, our results

directly carry over to this restricted version of Cellular Automata.

2.1.1 Previous Work

Cellular Automata. The study of self-simulation, and new types of universalities is as old as the field of Cellular Automata itself, with von Neumann introducing the model to build a self-replicating machine [21]. Though it was Banks in 1970 who explicitly coined the term intrinsic universality [2], von Neumann's initial construction was later proven to be intrinsically universal. Conway's famous Game of Life cellular automaton was proven to be intrinsically universal [10]. Intrinsic universality in CA has been extensively studied [3, 11, 12, 15, 22–25, 30]. Specifically, four different updating schemes for Asynchronous CA were shown to be IU in [30]. These updating schemes restrict which cells can be updated at each time step. The closest related updating scheme to Tile Assembly is "fully asynchronous" where only one cell may update at a time.[1]

Passive Self-Assembly. Intrinsic universality first crossed into the self-assembly world in [9], where a universal tile set was introduced for systems with tiles that bond with exactly strength 2. Two years later, the first properly intrinsically universal tile set, one that can simulate the full aTAM at any temperature, was presented in [8]. These papers both used the definition of intrinsic universality that we call non-committal. However, these definitions were later corrected to the version that we call committal [20]. It was also shown that a single polygon tile type with the ability to flip, translate, and rotate can simulate any aTAM system through several intermediate simulations [6]. The aTAM was found not to be committal intrinsically universal at Temperature-1 [20], and in directed and non-directed planar systems [13]. Directed 3D and Spatial aTAM were proven to be IU [13]. The 2-handed self-assembly model is, in general, not intrinsically universal; however, there are intrinsically universal tile sets for each temperature [7]. Work towards a universal tile set in Wang Tiles, which studies whether a given tile set can infinitely, and potentially periodically, tile a plane, has also been investigated [16–18].

[1] For the case of Tile Automata and Surface Chemical Reaction Networks it better stated as "one rule" can be applied at a time because two cells can be updated in one update.

Simulation between Tile Assembly and CA. The aTAM can simulate some versions of CA. In particular, it was found that the aTAM can simulate only finite CA [14]. The TA model does not have this restriction, as we can infinitely tile the plane with our seed assembly and use transitions to simulate infinite CA. Where the aTAM is asynchronous, nondeterministic, and finite, Cellular Automata is potentially generally synchronous, deterministic or nondeterministic, and infinite. Tile Automata is asynchronous, deterministic or nondeterministic, and finite. Additionally, Tile Automata is restricted to a neighborhood size of two.

Notable, IU in CA is usually possible with systems that contain a very limited number of states. However, in self-assembly, the simulating system does not only need to simulate the local interactions between existing states, but importantly also build new tiles in valid locations. Therefore, IU systems in tile self-assembly tend to use a lot more states.

2.1.2 Our Contributions

In this paper, we push forward the study of IU systems in a few ways. First, we prove that any passive self-assembly model (such as the aTAM) and variants of active self-assembly with bounded state changes cannot adhere to the stronger non-committal definition of intrinsic universality for self-assembly initially presented in [8]. However, this was later corrected and since then, a slightly more permissive definition for the simulation of dynamics for intrinsically universal systems has been used within self-assembly [20]. Although this is indication that the problem with modeling dynamics within passive models is known, to our knowledge, this has not been formally proven before.

Then, we show that in 2D, the seeded Tile Automata model, with unbounded state changes, does indeed adhere to this stronger non-committal definition of intrinsic universality. We do this by presenting a temperature-1 seeded TA system, and configuration of an initialized seed assembly, that is IU for all seeded temperature-1 systems in approximately 4600 states. We then show that any temperature TA system can be simulated by a temperature-1 TA system. We also prove that the

effect of temperature simulation on the scale of the system's supertiles is bounded. No additional states in the IU system's state set are required to simulate systems greater than temperature-1, extending our result to all seeded TA systems. Following this, we show that, due to the mechanics of TA, our construction can be adapted to prove that 2D Asynchronous Cellular Automata, with a cardinal radius of 1 and neighborhood size of 2, is also IU in approximately 2600 states, which although inefficient, is the first 2D ACA IU result. These positive results are summarized in Table 2.1 together with other known IU results.

Section 2.2 starts by giving precise definitions of the model. Then, we show that bounded state change systems can never be IU in Section 2.3. Opposing this negative result, we continue to show that Tile Automata systems with their unlimited state changes are IU. Due to the volume of necessary details, the paper first gives a high-level overview in Section 2.4, that discusses the main gadgets and the framework of how the pieces work together. We reference the more detailed later sections that follow the overview.

Section 2.5 then covers the temperature simulation part of the IU framework in depth. Next, sections 2.6, 2.7, and 2.8 detail the supertiles, their construction and how they transition respectively in full detail. We analyse the number of states in Section 2.9 and proof the correctness of the simulation in Section 2.10. We continue to show how our result transfers over to Cellular Automata in Section 2.11. We then summarize the conclusion with Section 2.12.

Intrinsic Universality Across Models					
Model	**D**	N	$\lvert T\rvert$ / $\lvert\Sigma\rvert$	**Scale (S)**	**Reference**
aTAM	2D	5	>10M	$O(n^4\log(n))$	[8]
aTAM	3D	7	152 000	$O(n^2\log(n\tau))$	[13]
Seeded TA Temp-1	**2D**	**5**	**4600**	$O(n^3)$	**Theorem 2.4.1**
Seeded TA	**2D**	**5**	**4600**	$O(\min((\tau n)^3, n^9))$	**Theorem 2.4.4**
Async. Cellular Automata	1D	3	$O(1)$	unknown	[30]
Block-Pairwise ACA	**2D**	**2**	**2600**	$O(n^3)$	**Theorem 2.11.1**
Pairwise ACA	**2D**	**2**	$O(1)$	$O(n^3)$	**Theorem 2.11.3**

Table 2.1: An overview of known and new simulation results for types of asynchronous cellular automata including tile assembly models. D is the dimension, N is neighborhood size of the input system, $|T|$ and $|\Sigma|$ are, respectively, the number of tile or state types in the universal system, S is the scale factor, n is the number of states in the input system, and τ is the system's temperature.

2.2 Preliminaries

In this section, we cover the basics of the Tile Automata model. We use many of the same definitions as in [1, 5]. An example of a Tile Automata system can be seen in Figure 2.1.

Tiles. Let Σ be a set of *states*. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^2$ and has a state of $\sigma \in \Sigma$. Let $\sigma(t)$ be the state of t. Let ϕ denote a special state called the *empty* state.

Affinity Function. An *affinity function* Π over a set of states Σ takes an ordered pair of states $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$, and outputs an element of $\mathbb{Z}^{0+}$. The orientation d is the relative position of σ_1 to σ_2, with $\vdash$ meaning horizontal and $\perp$ meaning vertical. State σ_1 is the west or north state, respectively. We refer to the output as the *Affinity Strength* between these two states.

Transition Rules. A *Transition Rule* consists of two ordered pairs of states $(\sigma_1, \sigma_2), (\sigma_3, \sigma_4)$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$. The rule denotes that if the tiles with states (σ_1, σ_2) are next to each other in orientation d (σ_1 as the west/north state) they may be replaced by the states (σ_3, σ_4).

Assembly. An assembly A is a set of tiles (with states in Σ), such that no two tiles occupy

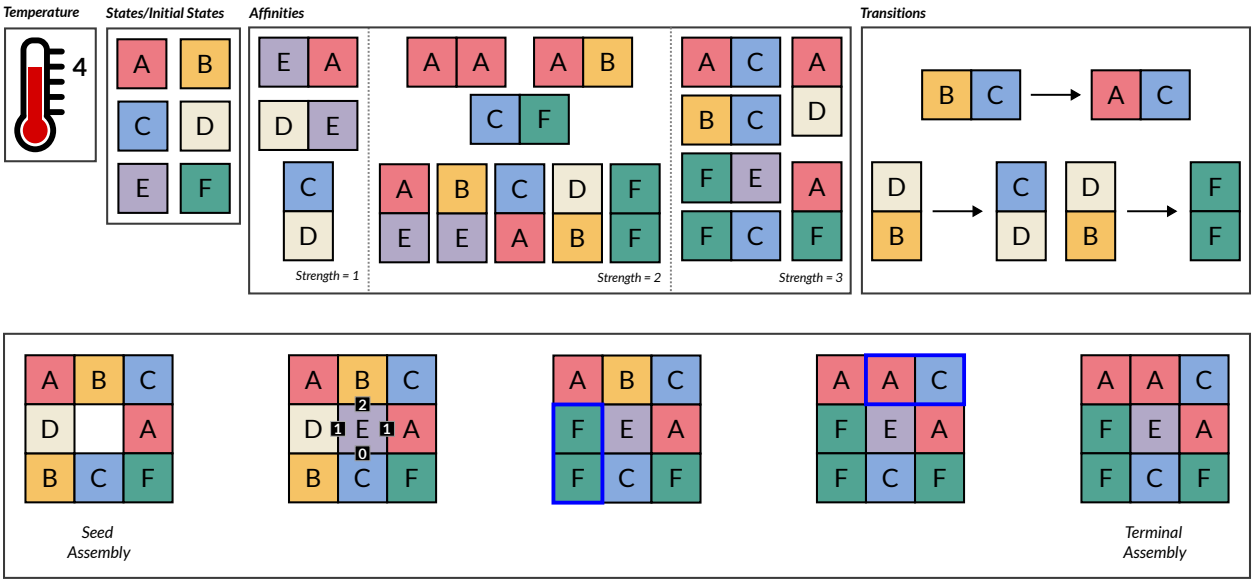


Figure 2.1: Example of a Tile Automata system with 6 states, a system temperature of 4, affinities of strengths 1, 2, and 3 vertical and horizontal transitions, and a seed assembly. The assembly sequence to a terminal assembly is also shown with the changes highlighted. Due to the affinity strengthening restriction, there is no detachment.

the same position, i.e., for every pair of tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2)$, it holds that $p_1 \neq p_2$. For an assembly A, let $A(x,y)$ denote the state of the tile with location $(x,y) \in \mathbb{Z}^2$ in A if such a tile exists and ϕ (the *empty* state) otherwise. For a set of states Σ, let A^{Σ} denote the set of all assemblies over state set Σ.

Let the *bond graph* $B_G(A)$ be formed by taking a node for each tile in A and adding an edge between neighboring tiles $t_1 = (\sigma_1, p_1)$ and $t_2 = (\sigma_2, p_2)$ in orientation d with a weight equal to $\Pi(\sigma_1, \sigma_2, d)$. We say an assembly A is τ-stable for some $\tau \in \mathbb{Z}^{0+}$ if the minimum cut through $B_G(A)$ is greater than or equal to τ.

2.2.1 The Seeded Tile Automata Model

In this paper, we investigate the *Seeded Tile Automata* model, which differs from the non-seeded Tile Automata model defined above, by only allowing single tile attachments to a growing seed, similar to the aTAM. Here we use many of the same definitions as in [1].

Seeded Tile Automata. A Seeded Tile Automata system Γ is a 6-tuple $\{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$

where Σ is a set of states, $\Lambda \subseteq \Sigma$ a set of *initial states*, Π is an *affinity function*, Δ is a set of *transition rules*, s is a stable assembly called the *seed* assembly consisting of tiles in states contained in Σ, and $\tau \in \mathbb{Z}^+$ is the *temperature* (or *threshold*). When we refer to a *tile set* (or equivalently *rule set*) we mean the four parameters $(\Sigma, \Pi, \Delta, \tau)$, that is, the states, the affinity function, the transition rules, and the temperature. A system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ is said to *use* rule/tile set $(\Sigma, \Pi, \Delta, \tau)$.

Attachment Step. A tile $t = (\sigma, p)$ may attach to an assembly A at temperature τ to build an assembly $A' = A \bigcup t$ if A' is τ-stable and $\sigma \in \Lambda$. We denote this as $A \rightarrow_{\Lambda,\tau} A'$.

Transition Step. An assembly A can transition to an assembly A' if there exist two neighboring tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2) \in A$ (where t_1 is the west or north tile) such that there exists a transition rule in Δ with the first pair being (σ_1, σ_2) and $A' = (A \setminus \{t_1, t_2\}) \bigcup \{t_3 = (\sigma_3, p_1), t_4 = (\sigma_4, p_2)\}$. We denote this as $A \rightarrow_{\Delta} A'$.

Affinity Strengthening. We only consider transition rules that are affinity strengthening, meaning for each transition rule $((\sigma_1, \sigma_2), (\sigma_3, \sigma_4), d)$, the bond between (σ_3, σ_4) must be at least the affinity strength of (σ_1, σ_2) and it must also maintain or increase any other neighbor affinities. Formally, $\Pi(\sigma_3, \sigma_4, d) \geq \Pi(\sigma_1, \sigma_2, d)$ and $\Pi(\sigma_3, \sigma_i, d) \geq \Pi(\sigma_1, \sigma_i, d)$ and $\Pi(\sigma_4, \sigma_j, d) \geq \Pi(\sigma_2, \sigma_j, d)$ $\forall i, j \in \Sigma$. This ensures that transitions may not induce cuts in the bond graph.

Producibles. We refer to both attachment and transition steps as production steps and say that $A \rightarrow_1^{\Gamma} A'$ if either $A \rightarrow_{\Lambda,\tau} A'$ or $A \rightarrow_{\Delta} A'$. For any sequence of assemblies $\{A_1, A_2, \ldots A_k\}$ such that $A_i \rightarrow_1^{\Gamma} A_{i+1}$ for all $1 \leq i < k$, we say that A_k is producible from A_1, and write $A_1 \rightarrow^{\Gamma} A_k$. Note that for any assembly A, $A \rightarrow^{\Gamma} A$. We say $A \rightarrow_{\geq 1}^{\Gamma} B$ if $A \rightarrow^{\Gamma} B$ and $A \neq B$. For a Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ we refer to the set $\mathbf{PROD}(\Gamma) = \{s\} \bigcup \{A | s \rightarrow^{\Gamma} A\}$ as the *producible assemblies* of Γ.

Terminal Assemblies. The set of terminal assemblies for a Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ is written as $TERM(\Gamma)$. This is the set of assemblies that cannot grow or transition any further. Formally, an assembly $A \in TERM(\Gamma)$ if $A \in \mathbf{PROD}(\Gamma)$ and there does not exists any assembly $A' \in \mathbf{PROD}(\Gamma)$ such that $A \rightarrow_1^{\Gamma} A'$.

Unique Assembly. A Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ *uniquely* assembles an assembly A if $A \in TERM(\Gamma)$, and for all $A' \in \mathbf{PROD}(\Gamma), A' \rightarrow^{\Gamma} A$.

2.2.2 Simulation

In this section, we formally define the concept of one tile automata system *non-committally* simulating another. We use a standard m-block simulation in which each tile of an assembly is simulated by a larger $m \times m$ block of tiles in the simulating system. The definition presented here is the same as that originally presented in [8], which we call non-committal IU. However, as stated before, that paper contained a subtle error that was later corrected to become committal IU.

The difference lies in the *models* concept, see Definition 2.2.3. In non-committal IU, this definition contains a universal quantifier, whereas the committal version contains a weaker statement. From here on, we focus on two tile Automata systems Γ_T and Γ_S, where Γ_S denotes a system that purports to *simulate* system Γ_T. Let Σ_T and Σ_S denote the set of states used in Γ_T and Γ_S, respectively.

***m*-block Supertiles.** An *m-block supertile* over a set of states Σ is a partial function $\lambda : \mathbb{Z}_m \times \mathbb{Z}_m \rightarrow \Sigma$, where $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. Let B_m^{Σ} be the set of all m-block supertiles over Σ. The m-block with no domain is said to be *empty*. For any assembly $\mathscr{A}$ over state space Σ, define $\mathscr{A}_{x,y}^{m}$ to be the m-block defined by $\mathscr{A}_{x,y}^{m}(i,j) = \mathscr{A}(mx+i, my+j)$ for $0 \leq i, j < m$.

Supertile representation and mapping. We refer to a function $R : B_m^{\Sigma_S} \rightarrow \Sigma_T \bigcup \{\phi\}$ as an *m-block representation function*. We require $R(B) = \phi$ for the empty m-block, and for any non-empty m-block B for which $R(B) = \phi$, we say B maps to a *ghost tile*. For a given m-block representation function R, define the partial function $R^* : A^{\Sigma_S} \rightarrow A^{\Sigma_T}$ such that $R^*(\mathscr{A}) = \mathscr{A}'$ if and only if $A'(x,y) = R(A_{x,y}^{m})$ for all $(x,y) \in \mathbb{Z}^2$.

c-Fuzz. The concept of c-fuzz is basically that a macroblock may have a bounded number of "extra" tiles attached to it without altering its mapping. This allows a simulating system to make minor intermediate attachments while enacting the simulation. Another way to think of c-fuzz is as

a reasonable allowance for limited-size non-empty macro-blocks (that map to an empty tile in the simulated system) to be used in the simulation process. Formally, a mapping $R^*(A) = A'$ is said to map to A with at most c-fuzz, for some $c \in \mathbb{Z}^+$, if and only if for all non-empty blocks $A^m_{x,y}$ it is the case that $R(A^m_{x+u,y+v}) \neq \phi$ for some integers u, v such that $|u| + |v| \leq c$. In other words, any non-empty macroblocks that map to ϕ (i.e., ghost tiles) are only at most c macroblocks away from a macroblock that maps to a real (non-empty) tile. We say a Tile Automata system achieves c-fuzz under mapping R^* if each producible assembly of the system achieves at most c-fuzz when mapped by R^*.

Definition 2.2.1 (Equivalent Productions)**.** *We say Γ_S has equivalent productions to Γ_T (under R) with up to c-fuzz, and write $\Gamma_S \Leftrightarrow_c \Gamma_T$, if the following hold:*

1. $\{R^*(A') | A' \in \boldsymbol{PROD}(\Gamma_S)\} = \boldsymbol{PROD}(\Gamma_T)$.
2. *Γ_S achieves c-fuzz under R^*.*

Definition 2.2.2 (Follows)**.** *We say that Γ_T follows Γ_S (under R), and write $\Gamma_T \dashv_R \Gamma_S$, if $A' \rightarrow^{\Gamma_S} B'$, for some $A', B' \in \boldsymbol{PROD}(\Gamma_S)$, implies that $R^*(A') \rightarrow^{\Gamma_T} R^*(B')$.*

Definition 2.2.3 (Non-Committally Models)**.** *We say that Γ_S (non-committally) models Γ_T, and write $\Gamma_S \models_R \Gamma_T$, if $A \rightarrow^{\Gamma_T} B$ for some $A, B \in \boldsymbol{PROD}(\Gamma_T)$, implies that for all A' such that $R^*(A') = A$, $A' \rightarrow^{\Gamma_S} B'$ for some $B' \in \boldsymbol{PROD}(\Gamma_S)$ with $R^*(B') = B$.*

Definition 2.2.4 (Non-Committal Simulation)**.** *We say Γ_S (non-committally) simulates Γ_T if for some $c \in \mathbb{Z}^+$, $\Gamma_S \Leftrightarrow_c \Gamma_T$ (equivalent productions), $\Gamma_T \dashv_R \Gamma_S$ and $\Gamma_S \models_R \Gamma_T$ (equivalent dynamics). We say the simulation is* clean *if it holds for $c = 1$, and we say the simulation achieves c-fuzz more generally.*

Definition 2.2.5 (Non-committal Intrinsic Universality.)**.** *A rule (tile) set $I = \{\Sigma, \Pi, \Delta, \tau\}$ is said to be intrinsically universal for a set of systems U if for all $\Gamma_T \in U$, there exists a system $\Gamma_S =$*

$\{\Sigma, \Lambda_T, \Pi, \Delta, s_T, \tau\}$ *that non-committally simulates* Γ_T. *The set U itself is said to be intrinsically universal if there exists a rule set I used by some system within U such that I is intrinsically universal for U. A model is said to be intrinsically universal if the set of all systems within that model is intrinsically universal.*

We use the term *non-committal* simulation to emphasize that the simulation definition we use is stronger than what is used in prior work, which we call *committal*. For the remainder of this paper, we deal exclusively with *non-committal* simulation and will just use the term *simulation* when not directly comparing with previous versions of simulation.

2.3 Impossibility for Passive or Bounded State Change Systems

In this section, we show that systems lacking the full state changing capability of the seeded Tile Automata model cannot achieve intrinsic universality under non-committal simulation. This includes well-studied models such as the Abstract Tile Assembly Model (aTAM) [28] and *freezing* variants of the seeded Tile Automata Model [5]. The key aspect of non-committal simulation that is impossible for these models is the non-committal modeling requirement of our simulation definition. In this section we show the impossibility of simulating a specific *passive* system, with the key use of the non-committal modeling requirement being used to prove Lemma 2.3.4.

Definition 2.3.1 (k-burnout, bounded, unbounded, passive, f reezing)**.** *For a non-negative integer k, a system is a k-burnout system if each tile in an assembly is restricted to only changing state at most k times. A system is called bounded if it is k-burnout for some k, and unbounded otherwise.* 0*-burnout systems are termed as passive. A freezing system [5] is one in which state change rules are such that a tile can never return to a previous held state.*

Observation 2.3.2. *Any aTAM system is a passive (*0*-burnout) system, and any freezing seeded TA system that uses* $|\Sigma|$ *states is bounded and a* $|\Sigma|$*-burnout system.*

Definition 2.3.3. *Define* X_n *to be the passive seeded Tile Automata system consisting of states* $\Sigma = \{S, a_1, a_2, \ldots, a_n\}$ *with seed tile in state S, and east-west affinity between S and each* a_i *of strength equal to the system temperature* τ*. Let* $s \cdot a_i$ *denote the producible assembly of this system obtained by attaching a tile of state* a_i *to the east of the seed tile.*

For the remainder of this section, let R denote a proposed m-block mapping function from macro blocks from a proposed simulator system to tiles from the system X_n.

Lemma 2.3.4. *For any system Y that simulates* X_n *under mapping R, and for any valid assembly sequence* $\langle A_{\pi_1}, \ldots, A_{\pi_m} \rangle$ *of Y such that for all* $1 \leq i \leq m$*,* $R^*(A_{\pi_i}) = s$*, either:*

1. *For all* $1 \leq i \leq n$ *there exists an assembly* A_i *such that* $A_{\pi_m} \to_1^Y A_i$ *and* $R^*(A_i) = s \cdot x_i$*, or*

2. *there exists an assembly* $A_{\pi_{m+1}}$ *such that* $A_{\pi_m} \to_1^Y A_{\pi_{m+1}}$ *and* $R^*(A_{\pi_{m+1}}) = s$.

Proof. Suppose constraint (1) does not hold for such a sequence $\langle A_{\pi_1}, \ldots A_{\pi_m} \rangle$ of Y, i.e. suppose that for some $1 \leq i \leq n$ there does not exists an assembly A_i such that $A_{\pi_m} \to_1^Y A_i$ and $R^*(A_i) = s \cdot a_i$. Since $Y \models_R X_n$ (Y non-committally models X_n), and $s \to^{X_n} s \cdot a_i$, it must be that there exists some assembly A_i such that $A_{\pi_m} \to^Y A_i$ and $R^*(A_i) = s \cdot a_i$, which by definition means there exists an assembly sequence $\langle A_{\pi_m}, B, \ldots, A_i \rangle$. Therefore, $A_{\pi_m} \to_1^Y B$, and since $X_n \dashv_R Y$ (X_n follows Y), we know that $R^*(B) = s$, which means the sequence $\langle A_{\pi_1}, \ldots, A_{\pi_m} \rangle$ can be extend with assembly B to satisfy constraint (2). □

Lemma 2.3.5. *For any bounded system Y that simulates X_n under mapping R there must exist $A, A_1, \ldots, A_n \in \boldsymbol{PROD}_Y$ such that $R^*(A) = s$, and for all $1 \leq i \leq n$, $R^*(A_i) = s \cdot a_i$ and $A \to_1^Y A_i$.*

Proof. Suppose a bounded system Y simulates X_n. Since Y is bounded, there must exist $M \in \mathbb{Z}^+$ such that for all assembly sequences $\langle A_{\pi_1}, \ldots, A_{\pi_m} \rangle$ of Y where $R^*(A_{\pi_i}) = s$, it is the case that $m \leq M$. This is the case since each assembly A_{π_i} maps to a single tile under R^*, thereby limiting the size of each A_{π_i} to a finite integer based on the (finite) scale-factor of the simulation and the (finite) fuzz factor c of the simulation. The number of state changes and tile attachments for each assembly A_{π_i} therefore has a finite bound in a system with a finite burnout number.

Since the length of such sequences cannot be extended infinitely, there must exist a sequence $\langle A_{\pi_1}, \ldots A_{\pi_m} \rangle$ for which no additional $A_{\pi_{m+1}}$ exists for which $A_{\pi_m} \to_1^Y A_{\pi_{m+1}}$ and $R^*(A_{\pi_{m+1}}) = s$. For this sequence that must exist, Lemma 2.3.4 implies that for all $1 \leq i \leq n$ there exists an assembly A_i such that $A_{\pi_m} \to_1^Y A_i$ and $R^*(A_i) = s \cdot x_i$. Therefore, there must exist $A = A_{\pi_m}, A_1, \ldots, A_n \in \mathbf{PROD}_Y$ that satisfy the requirements of the lemma. □

Lemma 2.3.6. *A bounded seeded TA system with fewer than $n^{\frac{1}{5}}$ states cannot simulate X_n.*

Proof. If a bounded system Y simulates X_n, then by Lemma 2.3.5 it must be the case that there exists $A, A_1, \ldots, A_n \in \mathbf{PROD}_Y$ such that $R^*(A) = s$, and for all $1 \leq i \leq n$ it holds that $R^*(A_i) = s \cdot a_i$

and $A \rightarrow_1^Y A_i$. Since $A \rightarrow_1^Y A_i$ for each A_i, we know that each pair of assemblies A_i and A_j differ at either a single point or two adjacent points (corresponding to a tile attachment or a pairwise state change). We now consider two cases:

Case 1: Suppose there exists an i and j such that A_i and A_j differ at non-overlapping points. In this case, we know that the rule or attachment applied to A to attain A_i is also applicable to A_j, and vice versa. This implies there exists a common assembly $A_{i\oplus j}$ such that $A_i \rightarrow^Y A_{i\oplus j}$ and $A_j \rightarrow^Y A_{i\oplus j}$. But since $X_n \dashv_R Y$ (X_n follows Y), it must then be the case that $R^*(A_i) \rightarrow^{X_n} R^*(A_{i\oplus j})$ and $R^*(A_j) \rightarrow^{X_n} R^*(A_{i\oplus j})$. This implies that $R^*(A_{i\oplus j}) = s \cdot a_i$ and $R^*(A_{i\oplus j}) = s \cdot a_j$, which is a contradiction.

Case 2: Suppose the points of difference for all assemblies A_i overlap each other in at least one of their points. Since each assembly's pair of points are adjacent (in the case that there are two), this implies that the union of all such points of difference is at most 5 points. If Y has $|\Sigma_Y|$ states, then there are at most $|\Sigma_Y|^5$ distinct state assignments possible for this 5-tile region. Thus, if $|\Sigma_Y| < n^{\frac{1}{5}}$, then there are fewer than n distinct 5-tile regions, implying that $A_i = A_j$ for some $i \neq j$, which is a contradiction since $R(A_i) \neq R(A_j)$. □

Lemma 2.3.7. *A passive seeded TA system with fewer than n states cannot simulate X_n.*

Proof. Suppose a proposed system Y with fewer than n states simulates X_n with representation function R^*. By Lemma 2.3.5 there exists $A, A_1, \ldots, A_n \in \mathbf{PROD}_Y$ such that $R^*(A) = s$, and for all $1 \leq i \leq n$, $R^*(A_i) = s \cdot a_i$ and $A \rightarrow_1^Y A_i$. As each A_i is attained by attaching a single tile to A, let point p_i denote the point of this attached tile in assembly A_i. Now consider two cases:

Case 1: Suppose there exist $1 \leq i, j \leq n$ such that $p_i \neq p_j$. In this case the tile attached to form A_j from A can also be attached to A_i, and vice versa, implying that the assembly consisting of attaching both such tiles, call it $A_{i\oplus j}$, is such that $A_i \rightarrow^Y A_{i\oplus j}$ and $A_j \rightarrow^Y A_{i\oplus j}$. But since $X_n \dashv_R Y$ (X_n follows Y), it must be that $R^*(A_i) \rightarrow^{X_n} R^*(A_{i\oplus j})$ and $R^*(A_j) \rightarrow^{X_n} R^*(A_{i\oplus j})$, which implies that $R^*(A_{i\oplus j}) = s \cdot a_i$ and $R^*(A_{i\oplus j}) = s \cdot a_j$, which is a contradiction.

Case 2: Suppose instead that $p_i = p_j$ for all $1 \leq i, j \leq n$. Since Y has less than n states, there must exist some $1 \leq i, j \leq n$ such that A_i and A_j use the same state at point $p_i = p_j$. This implies that $A_i = A_j$, which is a contradiction since $R^*(A_i) \neq R^*(A_j)$. □

Lemmas 2.3.6 and 2.3.7 show that without the unbounded state change capability of the full seeded Tile Automata model, there exists a simple class of passive systems that cannot be simulated under non-committal simulation without arbitrarily larger state spaces. This gives us the following negative results for two established models in regards to non-committal simulation, directly following from Lemma 2.3.7 and 2.3.6:

Theorem 2.3.8. *The Abstract Tile Assembly Model (aTAM) is not intrinsically universal under non-committal simulation.*

Theorem 2.3.9. *The Freezing Seeded Tile Automata model is not intrinsically universal under non-committal simulation.*

2.4 Overview of Intrinsic Universality in TA

Now that we have shown that any bounded system cannot be intrinsically universal under non-committal simulation, we will show that Tile Automata (TA), with its unbounded state changes, is non-committal intrinsically universal. We do so by characterizing a TA system that can simulate any other TA system. For ease of presentation, we first give a high-level overview of the framework and some of the main techniques used to achieve the simulation of any TA system with the one presented. Both a detailed exposition of the techniques, as well as any proofs omitted in this section can be found in the later sections covering the details.

We show that seeded TA is IU by first showing that temperature-1 seeded TA is IU at scale $O(|\Sigma|^3)$ with constant states. We then show how we can simulate a seeded TA system at any temperature with a temperature-1 system at scale-1 with $O(\min(|\Sigma|^3, \tau|\Sigma|))$ states. These combine to create a general IU result for any seeded TA system by bounding the number of states to a constant for any temperature, or by scaling based on the temperature.

2.4.1 Temperature-1 Seeded TA is Intrinsically Universal

Section 2.5 gives the full details for the IU results with temperature-1. We show that there exists an intrinsically universal temperature-1 system with a constant number of states if we increase the scale factor to $O(|\Sigma|^3)$. Here, we give an overview of the framework used to prove the following.

Theorem 2.4.1. *There exists a tile set* $(\Sigma_U, \Lambda_U, \Pi_U, \Delta_U)$ *such that, for all systems* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, 1)$*, there exists a* $\Gamma' = (\Sigma_U, \Lambda_U, \Pi_U, \Delta_U, s_U, 1)$ *that simulates* Γ *at scale* $O(|\Sigma|^3)$*.*

Supertiles. Supertiles are $m \times m$ blocks that map to a specific tile in some state from the original system that is being simulated. Each supertile contains all information necessary for the simulation. For specifics, please refer to the detailed walk-through of the operations in Section 2.6.

Figure 2.2 shows a simplified diagram of a single supertile. Every supertile has binding sites on each of the four sides, and wires on each side that lead to a central lookup table corresponding to valid affinities and transitions for the system being simulated. Within the table each column

represents a state in the system being simulated and each row a state and direction of a neighboring supertile. There are datacells at each intersection.

The supertile makes use of several small gadgets for effective and correct communication and information transmission, as well as ensuring non-committal simulation. The most important gadgets are listed here. For a complete explanation of their workings and purpose, see Section 2.6.

- **Lookup Table.** Each supertile contains a lookup table that contains all the affinity and transition information of the system to be simulated. Thus, every tile has all information necessary to update itself from its neighbors.

- **Transition Storage Area.** All of the transitions for a pair of states are stored within a storage area in ordered data strings. For each transition rule, only the halves of pairs pertaining to the current supertile are stored.

- **Datacells.** For each directed pair of states in the system, a datacell stores the possible transitions and affinity status between them. The datacell is a compound gadget comprised of a transition storage area, and a single tile containing the affinity and current state status. For non-committal IU, the affinity must be chosen in one step, which is why the affinity is stored in a single tile.

- **Transition Selection.** For non-committal IU, any change to the mapping of a supertile must occur with a single tile placement or transition. This requires careful collaboration and ordering around the tiles that can change this mapping. Most of the information must be obtained from the lookup table and brought to the edge. The transition selection gadget contains this reversible process of getting the supertile (or supertiles) ready to change the mapping (or mappings) irreversibly in such a way that it could be reversed at any point up until the single mapping transition. Once the state mapping has changed, it must communicate this information to the rest of the supertile.

Attachments. The attachment process for a new supertile works approximately as follows, the details of which can be found in Section 2.7. Attachment is triggered by a supertile, when it discovers that no neighbor exists adjacent to it. The builder supertile finds that an affinity in that direction exists, it prepares itself for construction, by locking its outer edge, wiping its wires, and deactivating its gadgets.

Next, it starts building the new supertile. Should the supertile find a competitor trying to build in this spot, one of the two nondeterministically prevails. The builder supertile then copies over each part of the supertile one by one.

Once built, the new supertile requests the states of all its neighbors to select its own state. From the valid states, one is chosen nondeterministically, and the representative state column is activated. Finally, the new supertile's state is sent to its neighbors.

Transitions. The process of transitioning happens in seven general phases. The full details can be found in Section 2.8.

1. The existence of one or more transitions between two neighboring supertiles is confirmed, and each supertile's table is locked for the duration of the transition process.

2. The data strings within the transition storage area of the datacell at each supertile's respective intersections are copied and transmitted to the transition selection gadget.

3. Agents within the transition selection gadget nondeterministically select the new states associated with a transition rule or abort the transition altoghether.

4. Once a transition has been chosen, the new states are copied and transmitted to each supertile's respective tables for updating.

5. In the table, the old state is deselected, and the new state is activated.

6. The transition selection gadget is wiped.

7. The tables are unlocked, and new states are transmitted to neighboring supertiles.

2.4.2 Temperature Simulation at Scale-1

Using these techniques we will show that seeded TA at temperature-1 is intrinsically universal. We also show that at scale-1, we can simulate a seeded TA system at any temperature if we scale the number of states in the system. We provide two bounds on the scale factor: $O(\min(\tau|\Sigma|, |\Sigma|^3)$. Resulting in the following two Lemmata.

Lemma 2.4.2. *For all Tile Automata systems* $\Gamma_\tau = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ *there exists a system* $\Gamma_1 = (\Sigma_1, \Lambda_1, \Pi_1, \Delta_1, s_1, 1)$ *that simulates it with 1-fuzz at scale-1 such that* $|\Sigma_1| = O(\tau|\Sigma|)$.

Lemma 2.4.3. *For all Tile Automata systems* $\Gamma_\tau = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ *there exists a system* $\Gamma_1 = (\Sigma_1, \Lambda_1, \Pi_1, \Delta_1, s_1, 1)$ *that simulates it with 1-fuzz at scale-1 such that* $|\Sigma_1| = O(|\Sigma|^3)$.

2.4.3 Seeded TA is Intrinsically Universal

By taking Theorem 2.4.1 in conjunction with Lemmas 2.4.2 and 2.4.3, we achieve the desired result that seeded Tile Automata is non-committal intrinsically universal. This follows by directly plugging in the state-scaling into the temperature-1 construction.

Theorem 2.4.4. *There exists a tile set* $(\Sigma_U, \Pi_U, \Delta_U, 1)$ *such that, for all systems* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$, *there exists a* $\Gamma' = (\Sigma_U, \Lambda_U, \Pi_U, \Delta_U, s', 1)$ *that simulates* Γ *with 1-fuzz at scale factor* $O(\min((\tau|\Sigma|)^3, |\Sigma|^9)$.

2.5 Temperature Simulation

We now give a detailed construction of our universal Tile Automata system. We show how any Tile Automata system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ of any temperature τ can be simulated by temperature 1 with 1-fuzz by using ghost tiles and adding intermediary states.

In order to attach tiles that require *cooperative binding*, the necessity of needing multiple neighbors to attach a single tile in order to reach necessary affinity strength, we use intermediary states to add together the affinity strengths of surrounding tiles to the interim state tile that is attempting to be placed at that location, see Figure 2.3 for an example, and Figure 2.4 for the assemblies it produces.

Lemma 2.4.2. *For all Tile Automata systems $\Gamma_\tau = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ there exists a system $\Gamma_1 = (\Sigma_1, \Lambda_1, \Pi_1, \Delta_1, s_1, 1)$ that simulates it with 1-fuzz at scale-1 such that $|\Sigma_1| = O(\tau|\Sigma|)$.*

Proof. The set of states Σ_1 contains τ states for each $\sigma \in \Sigma$. We simulate the system in Figure 2.1 with the one in Figure 2.3, which contains the following states:

- An unlocked state σ for every $\sigma \in \Sigma$.
- Locked states σ_{Ld} for $d \in \{N, S, E, W\}$ for the directions. The lock L is represented by a lock icon in Figure 2.3.
- Counting states $\sigma_{i,Q}$ numbered from 1 to $\tau - 1$, where Q is all subsets of $\{N, S, E, W\}$.
- Success unlocking neighbor states $\sigma_{Q,\checkmark}$.
- Failure unlocking neighbor states $\sigma_{Q,\times}$.
- An empty state **g**, which we call a ghost tile.

Empty States. The state **g** has affinity with all non-ghost tiles σ, the states which map to something. This ghost state transitions with unlocked tiles adjacent to it to enter a counting state

representing a tile which may attach. This process is outlined in Figure 2.4. If the strength of the affinity is greater than or equal to the input system temperature, then the counting tile immediately transitions to a success state and starts unlocking its neighbors. If the sum is not yet τ the neighbor state is transitioned to locked and the counting tile increased based on the new binding strength.

Additionally, the attachment may nondeterministically choose to fail and begin the unlocking process of all locked surrounding states at any time. This has two functions. First, if the tile does not have 4 neighbors and it cannot reach the affinity strength, then it would be unable to detect the lack of neighbor on its own. The second reason is to ensure that the strict definition of simulation can be met.

Simulation. A ghost tile may not attach to another ghost tile or to a tile with a temporary state, nor can a temporary state affect its own affinity strength count. This ensures the system has 1-fuzz. A tile can only transition from a ghost tile to simulate attachment if there exists enough locked neighbors which reach τ so we know every assembly in $\mathbf{PROD}(\Gamma_1)$ maps to something in $\mathbf{PROD}(\Gamma)$. This shows equivalent production.

For following and strongly modeling we note that transitions are simulated in one step so the rules in $\Delta \subset \Delta_1$. For every attachment that could take place in Γ we simulate this via the adding states. The failure states serve two purposes, first if we select a tile that does not have enough affinity it will eventually be abandoned and another will be selected. Second it allows us to satisfy the Strongly Models definition as when an assembly $A' \in \mathbf{PROD}(\Gamma_1)$, which represents $A \in \mathbf{PROD}(\Gamma)$, can abandon any attachment step to circle back and reach an assembly which represents any B such that $A \to^{\Gamma} B$. □

2.5.1 Alternate Upper Bound

The dominating factor of the tile set is the adding tiles. We may replace these by instead of storing the temperature we may store the current neighbors. This is a better bound in the case that $\tau \geq \mathscr{O}(\Sigma^2)$.

Lemma 2.4.3. *For all Tile Automata systems* $\Gamma_\tau = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ *there exists a system* $\Gamma_1 = (\Sigma_1, \Lambda_1, \Pi_1, \Delta_1, s_1, 1)$ *that simulates it with 1-fuzz at scale-1 such that* $|\Sigma_1| = O(|\Sigma|^3)$.

Proof. Consider an alternate set of adding states, which store the current neighbors of the ghost tile instead of adding the strength. This encodes Π into Δ directly without adding up τ. We only need to store up to 3 neighbors as the fourth neighbor will be read in the final transition. □

2.6 Supertiles

A supertile is a block of m by m tiles that maps to a single tile in the system it is simulating via the m-block representation function. Each supertile contains the complete rules of the system it is simulating and, hence, can perform attachment and transition operations locally with its neighboring supertiles. To do this, the supertile contains a lookup table that stores the possible transition rules and affinities for each combination of states and neighbor directions.

Each state of the system to simulate is first mapped to a unary encoding. The table contains four smaller subtables, one for each neighboring direction. Each of these subtables is constructed as a matrix. The column indicates the state this supertile represents, and the row the states the neighbor can represent. We call an entry in this matrix a *datacell*. A datacell that is part of the East subtable stores at position (i, j) the affinities and transition rules that apply if this supertile represents state j and the East supertile would represent state i. Lastly, each supertile has an *active* column. This column indicates which state the supertile currently represents.

Besides the lookup table, a supertile contains *wires* that connect the table to the edges of the supertile and *gadgets* for reading, writing, and locking the table.

2.6.1 Agents & Gadgets

A supertile is comprised of several gadgets, groups of tiles that together perform a specific function, such as facilitating data traversal or table lookup queries. Agents are small packets of information encoded by tile states that traverse a supertile and can transport information from one part of the system to another. Figure 2.2 shows a single agent that has traversed from the supertile neighboring to the south to perform a lookup in the table. Other tasks specific agents can perform include locking the edges of a supertile or its table, clearing wires, or coordinating construction functions.

Gadgets are groups of tiles that together serve a specific purpose. They are reset after each use and are, therefore, reusable. Whereas agents move through the system, gadgets are largely

stationary. Agents can interact with gadgets, and each gadget serves a specific purpose.

Wire. The simplest gadget is the wire. The only purpose of a wire is to allow the one-way traversal of an agent from one part of the system to another. A wire is a one-wide string of tiles. The states of the wire tiles not only indicate it is a wire tile, but also indicate which direction the wire is going. An agent can traverse a wire by swapping states with a neighboring wire tile if the direction of that wire tile allows it.

Wires connect supertiles and allow them to communicate. Since a supertile has a wire connected to its neighbor for each state it can be in, the specific wire on which a supertile S communicates with its neighbor is an implicit communication of the state of S, see Figure 2.2.

Data Strings. Data strings are a series of tiles carrying data capable of traveling down wires. Transition related data string consist of a start data string tile, a string of unary 1 tiles, an end data string and on occasion a prepended instruction.

Door. Doors are tiles placed along wires to control the flow of data and construction. They consist of two parts. The first part is the actual door, placed on the wire in question. The other is its *handle*. When an agent or data string reaches a door, it can pass if the door allows it.

Each door has a specific direction dependent on the wire it is on. If the wire the door is on switches directions, then the door's direction will also flip. An agent trying to pass an open door swaps states with the door as if it were a normal wire. The door then enters a *reset* state, indicating it recently let an agent through and is currently not connected to its handle. No other agent is allowed to pass in this state. Once the agent has moved on, the door can swap states with the new wire tile on its original spot, resetting the door. This prevents doors from getting lost or mis-matched to the wrong handle, see Figure 2.5.

When a door swaps with a wire tile, the wire tile will be transitioned to a blank wire tile and take on the wire state of the next wire tile it encounters. Agents may swap with blank wire tiles but, due to the lack of direction, may also swap back if a normal wire tile state has not been selected.

The copy director may also copy into blank wire tiles. This blank wire tile method ensures that wires will not become inappropriately shuffled during traversal.

Additionally, agents or data strings may occasionally swap with the wire tile a door needs to return to its normal position, blocking it from doing so. In this case the door sends a repelling signal to the offending tile making it traverse with a wire tile backward once, if there is not a wire tile behind it and instead another agent or data string tile the repel state will be passed from tile to tile until one is able to swap. Then, all of the previous tiles still in the repelling state may return to normal by swapping with this wire tile until it reaches the door in need of resetting.

Crossover Gadget. When wires crossover within the table, at the edge of the supertile, and within construction wires, we require a gadget to control the flow of information across these wires, see Figure 2.6. The crossover gadget has 3-4 doors arranged with crossover gadget handles at each corner, all around a center wire tile. In most cases, a locking/unlocking agent or the agent's own transition rules ensure it passes through a crossover gadget in the appropriate direction; however, in a few rare instances, the agent will need to signal through the crossover gadget to lock other doors within the crossover before traversing, unlocking them upon exit.

Punchdown Gadget. The punchdown mechanism allows the distance between datacells to be calculated by decrementing a data string representing the necessary number of columns to traverse in the table. See Figure 2.7 for how decrementing works with the punchdown gadget. When a data string comes along a wire, the punchdown gadget will "delete" one of the unary digits by transitioning it into a normal wire tile. The rest of the data string is allowed through the door as normal. The end-of-string tile resets the punchdown gadget.

Transition Selection Gadget. On each of the four edges of the supertile, directly next to the wires, lies a Transition Selection Gadget, see Figure 2.8. Upon initiation of a transition with a neighboring supertile, this gadget and its mirror on the neighboring side together determine which transition to take. The gadget is filled with the transition rules when a transition is initiated. Once

this is done, each supertile initiates a nondeterministic selection agent to walk up and down the border between the two selection gadgets. When they meet at a rule, they may or may not transition with one another to select that transition rule to be executed on the two supertiles. The newly selected states are then returned to the tables of the two supertiles to update their respective active columns. The complete workings of supertile state transitions are explained in Section 2.8.

2.6.2 Table & Wiring

Table. The table is the primary gadget of the supertile. It stores the affinities and transitions of all possible pairs of states in the system and the functional state of the supertile.

The rows are ordered to prevent the crossing of wires at the border of supertiles. Hence, the rows are ordered in reverse for the North and West subtable, as shown in Figure 2.2. The columns are ordered normally.

A datacell is a single cell in a subtable and stores both the affinity and transition information for a specific neighbor direction, see Figure 2.9. It is a compound gadget comprised of an incoming wire, an outgoing wire, wire traversal doors, a punchdown gadget, an affinity door, and the transition storage area. It can interact with data strings, and agents can use it to get information about the state of the supertile.

Since we use temperature 1, affinities stored in a datacell are simply a Boolean; a combination of two states either has an affinity in this orientation or not. Each column in the table has a vertical wire next to each datacell. On this wire, for each datacell, this door indicates if this specific datacell has an affinity or not. If there is no affinity, the door is a normal door. Otherwise, the door is a special affinity door, indicating the affinity.

Transition rules are stored in a transition rule storage compartment at the bottom of each datacell. They are stored as follows. If the combination of the supertile state and the neighboring supertile state corresponding to this datacell has a transition rule, the storage contains only the new state this supertile would become if this transition is taken. If the system we are simulating allows

for multiple transitions for this pair of states, we define a fixed order for these transitions prior to the simulation. The resulting states are then stored in the transition storage area according to this order. The rules always match up because we predefined this order and because the supertile template is copied every time. The storage is templated to be the size that is necessary to store the maximum number of transition rules of any state pair in the system, such that all transition storage compartments have the exact same size. If a compartment contains fewer transitions than the maximum, it is filled with blank transitions.

Table Locking. The wires enter the table from the left. Each wire has a special table locking door and corresponding handle, that is situated at the edge of the table. If an agent tries to act on the table, it first must pass its corresponding door. If that door is open, two locking agents move up and down along the left edge of the table. These signals lock all other doors corresponding to incoming wires, allowing for only one operation on the table at a time. If two of these locking agents meet, only one is allowed to continue on while the other disappears. Once the locking agent reaches either the top or bottom of the left edge, it transforms into a successful locking agent and moves back toward its door. A door that sent out locking agents and that sees another locking agent coming by, knows that its own locking agent was the one that failed, and the door will lock itself. Even if multiple doors try to lock the table at the same time, only a single door will receive its corresponding successful locking agents coming back from both the top as well as the bottom. This door then allows the waiting agent to enter the table. An agent trying to enter a locked table will simply wait. This does not lead to deadlocks, since the origin table of this agent was not locked when this agent left it.

2.6.3 Outer Shell

The outer frame of both the table and the supertile are comprised of outer frame tiles, with the appropriate doors along wires to access the supertile or table, see Figure 2.2. Inside, they contain an inner wire for traversing the boundary, and then inner frame tiles, again with the appropriate

doors. Initially, supertiles are built with doors that indicate that no neighbor is present. Once a neighbor is found, they transition to their regular counterparts.

2.7 Attachment

The attachment process consists of several phases. First, if the supertile detects it has a spot next to it without a supertile build, it checks if there is an affinity in that direction. Next, the supertile is copied piece by piece into the neighboring spot. Lastly, when the construction is completed, this newly built ghost tile sends a signal to all neighboring tiles to select an actual state for itself.

2.7.1 Initiation

Whenever a supertile changes state, either via transition or via attachment, it sends out state transmission agents to all four directions. When such an agent reaches an edge of its supertile and finds no neighbor, the process of determining whether to attach a new supertile starts, see Figure 2.10. The state transmission agent is not able to reach the true edge of the supertile if there is no neighbor present. Instead, it reaches the inner row of doors on the edge of the supertile that controls access to the outline wire. Since there is no neighbor, the agent changes state to a lookup state, enters the outline wire, and goes to the affinity selection wire, see Figure 2.10. Initially, all outline wires on the edges are directed such that if no neighbor is found the state transmission agent will be directed to the affinity selection wire.

The affinity selection wire wraps around the outside of the entire subtable for its given direction (E/N/W/S) allowing for the agent to drop into the active state column wire and search every possible state in the system for an attachment. For example, if the missing tile is to the West of the supertile then the wire runs above West and its outgoing wire is below West 1.

2.7.2 Checking Attachment

Each datacell has an affinity door in the vertical active column wire next to it, indicating that there is an affinity between these two states in this direction. If there is no affinity, there is a door with a no affinity state. The initially selected possible state may not be the final state of the supertile so as not to preclude every possible state that the supertile may end up in. This initial lookup is just to ensure we do not begin construction of a neighboring supertile if no tile can attach there in the

system we are simulating.

Lookup in Table Section. Once the signal reaches the edge of the table it will initiate the standard table locking process described in Section 2.6.2, and visible in Figure 2.11. The signal will then traverse the table until reaching the active state column and begins its descent down the active state column wire with the affinity doors; see figure 2.12.

The lookup agent may nondeterministically transition with any affinity door to a *found* state so that the construction process can begin. It will traverse down to the state lookup exit wire and to the edge of the supertile. The table stays locked. Upon reaching the edge of the supertile, the agent transitions into the Copy Checkpoint. The Copy Checkpoint is a stationary tile on the edge of the supertile, orchestrating the copy process. See Figure 2.12 for the state lookup process at a high level. If there are no affinities in that direction or the lookup agent never transitions with an affinity door, the agent will not change to the found state and simply unlocks the table after exiting from the bottom of the state lookup wire. No attachment is started in that case; see Figure 2.13.

We only begin construction if there is a possible attachment in that particular direction. This does give a slight fuzz advantage over previous builds. A ghost tile (if simulating a system that requires or includes them) will not begin growing if the state has no affinities in that direction.

2.7.3 Preparing for Copying

In order to properly copy the tile, we need to clear the interior wires, lock any further outside communication from coming in, and activate a number of processes.

Locking the Supertiles Outer Frame for Construction. First, the Copy Checkpoint sends a locking agent around the outer supertile wire, which locks every outer door on the edges of the supertile, preventing any further agents from entering the supertile.

Clearing Table and Wires. Once the locking agent returns to the Copy Checkpoint, it will turn into a wiping agent. The main wiping agent will traverse the edge wire and spawn minor wiping agents to sweep every wire. They delete any waiting agents outside of the table. Moreover,

the interior and edge of the table are reset to be entirely inactive. This process is shown in figure 2.14. Should an agent trigger a locking process before it is wiped, the locking agents spawned will transition into wire tiles upon contact with an inactive door. The active state of the current supertile is stored within a containment area at the top of the active column to ensure it is spared from wiping.

2.7.4 Copying Supertile Outline

Now, the supertile is ready to start the copy process. We will copy the supertile in several steps, piece by piece. We start with the outline, then the table, and then the wires. It could be that multiple adjacent supertiles are trying to build in the same location. In this case, it is necessary to ensure that only a single supertile gains construction jurisdiction over this spot.

Claiming Mirror Side. The mirror edge is the edge of the tile under construction that is immediately adjacent to the supertile initiating construction. Once the wiping agent reaches the Copy Checkpoint, the Copy Checkpoint will spawn 2 claiming agents sent along the outer wire to the adjacent corners. If building to the West, these claiming agents go to the North and South. The purpose of these agents is to place and claim the mirror edge corners. These are important to prevent construction conflicts. Once the agents reach the corners, the northwest and southwest corners in our example, as seen in Figure 2.15, they open the supertile corner doors and try to build the corner of the neighboring tile.

They build the corner of the neighboring tile as follows. The agent first goes through the door in the direction it wants to build. Then, it attaches the first empty construction tile and then a second, transitioning the first into a door as it swaps into the second. Whenever we say that a certain tile is attached or built, we mean that an empty construction tile is attached, which is then transitioned into the correct state. After the wire tile is built, the agent swaps with the wire tile, and tries to build the crossover gadget. It marks the crossover gadget as claimed by the East side where it came from.

It could also be the case that there already exists a corner crossover gadget. This gadget can

then either be claimed, or unclaimed. If it is unclaimed, it is claimed, otherwise, the agent goes back to the Copy Checkpoint to report a failure. If the crossover corner is successfully claimed, the agent also goes back, but this time to report a success. If the Copy Checkpoint receives at least one failure, it relinquishes its claim to any successful corners (via agent) and aborts construction.

To abort a construction at this point, the only thing necessary is to unlock the doors on the outline wire, set the active state column in the table, and finally unlock the table. These processes are the same as their counterparts at the start of construction.

When the Copy Checkpoint receives two successes, it knows that from the three other supertiles that could potentially try to build in this spot, only one might still be trying. In this case, we initiate construction of the full mirror edge. This is done as follows. One mirror edge agent is sent to one of the corners that has just been claimed. This agent will then build the outline wire of the mirror edge. At the same time, it will attach (via a blank construction tile) *mirror* tiles. These tiles then transition with the tiles in the original supertile to mirror their state. Should the corner it is attempting to mirror be a crossover gadget door, the mirror tile will spawn a crossover copy agent, which when the subordinate copy agent recognizes this will instead back up, transitioning the tiles it traverses back from into blank construction tiles. When the crossover is activated, it will continue. In this way, the doors get placed in the correct positions. Moreover, the mirror edge agent ensures intersections are properly constructed and construction doors are activated, see Figure 2.15. Once it has reached the other corner, it goes back to the Copy Checkpoint, which can then initiate the next phase.

Copying Placement General Notes. Before explaining the next phase, we will first detail the standard copy procedure. This procedure is used to copy the rest of the supertile. It uses a Copy Director, which acts in the original supertile and sends copies of tiles to the Placement Director, which is located in the newly constructed tile, and places the copies on the correct locations. The copies are send over the outline wires and/or construction wires of the tiles. For this to work, the route from the Copy Director to the Placement Director needs to be clear and doors along this path

need to be set correctly to ensure copies of tiles end up in the correct spot.

The setup is done by a copy agent send out from the Copy Checkpoint. It first places the Placement Director in the appropriate spot. Then, it goes to place the Copy Director. It takes the same path as the copied tiles will take. While going over this wire, it ensures all wire tiles are pointing in the correct direction and doors that lead in the wrong direction are closed.

Once at the correct spot, it transitions into the Copy Director and starts copying tiles and send them to the Placement director via the path it just created. As soon as all tiles of this part are copied, it goes to the Placement Director, deletes it, and finally returns back to the Copy Checkpoint, which can then start the next phase of copying.

The Copy Process. For each tile that needs copying, the Copy Director follows the following scheme, visualized in Figure 2.16. First, the Copy Director sends a direction to the Placement Director (North/South/East/West). Then, it swaps with the tile that needs copying. This tile then spawns a copy of itself on the wire that also goes to the Placement Director. Lastly, the Copy Director swaps back with the tile that now has copied itself.

The Placement Director ensures it is always at the end of the part that is built. It first receives a direction. It then swaps with that direction tile which attaches an empty construction tile. Then, when the copy arrives, the Placement Director swaps with the copy, and the copy can transfer its state to the newly attached empty construction tile. Then, the Placement Director swaps back with the now copied tile and deletes it in the process. This process is shown in Figure 2.17.

Not every tile is copied over individually, to reduce the number of states, we copy the crossover gadget in one go. Instead of sending a copy of every tile in the crossover gadget, we send a single tile containing a template of full information of the crossover gadget. To stop the directionality from being an issue our copy director will send a second special direction tile before a crossover gadget. This way the agent may be in the middle of the gadget attaching blank tiles and transitioning the surrounding doors into them without knowing the direction from which the crossover came. Each door remain in waiting state until it has attached its handle. The placement

director will lock any necessary doors when construction is complete.

Constructing Adjacent Supertile Outline Wires. This copy process is used to build the other edges of the supertile, see Figure 2.18. At this point, only the mirror edge has been constructed. We use the copy process to build one edge at a time. For horizontal attachments, we first build the top edge, then the bottom edge. For vertical attachments, we first build the left edge, then the right edge. This is to ensure that if there is still another supertile that is trying to build in the same spot, we recognize this situation and deal with it accordingly.

To build these edges of the supertile, the Copy Checkpoint spawns a new copy agent. This agent will put a Placement Director at the corner that was already built, then moves over the outline wire of the original supertile to the other side, where it will transfer into a Copy Director.

If there is still another supertile building in this spot, the two Placement Directors will eventually meet. Nondeterministically, one of the two continues, while the other is removed. The losing Placement Director sends a signal to its respective Copy Checkpoint, which then starts the abortion process similar as before. The Placement Director that is left over will ignore and remove any copied tiles intended for the now removed Placement Director.

Along with the outline wire the transmission selection gadgets are added.

For a tile to start internal construction it must verify it has claimed all 4 corners. Only doors on its side are active all others are blocked.

Building the Far Side of the Outline. Once we have confirmed claim of all four corners we then can build the opposing edge starting at the designated corner. This again uses the normal copy process. The Copy Checkpoint sends out a copy agent, which places a Placement Director. Then, this copy agent traverses the outline wire to the appropriate place, locking doors on the way if necessary. Lastly, it will transition into the Copy Director and start copying the last edge of the supertile, see Figure 2.19.

2.7.5 Construction Wires

Once construction of the outline of the new supertile is complete and confirmed by an agent at the Copy Checkpoint, we start copying the table. Each table has eight construction wires. These wires extend from the corners of the table, and mark the width and height of the table, plus its position within the supertile.

To copy the table, we first copy the construction wires. For horizontal copying, we start by copying the horizontal construction wires, then the vertical ones. For vertical copying, we do the opposite. These wires are not only used to indicate the placement of the table within the supertile, but they will also be used to copy the contents of the table.

To copy them, we first open the doors connecting the respective wires to the outline wire. We then copy the wires using the normal copy process, starting at the far end. The horizontal and vertical process can be seen in figure 2.20 and figure 2.21.

2.7.6 Copying Table

The construction wires already contain the outer edges of the table, including the table control edge on the eastern side of the table. Once these are in, we build the rest of the table. Importantly, the only variable aspects of a datacell are the width and the height of the transition storage. These depend on the system we are simulating.

Copying Horizontal Table Wires. To transfer this information over to the new supertile, we first copy all the horizontal wires in the table. These already contain the transition, affinity, and state lookup chute crossover doors. We copy these over using the normal copy procedure, see Figure 2.22. Both the Copy and Placement Directors start this procedure on the East side of the table, and the route that the copies take is via the construction wires.

Every time the Copy Director has finished a horizontal wire, it moves down to the next horizontal wire. For every step that it takes, it sends a token to the Placement Director.

For every one of these tokens that the Placement Director receives, it goes one step forward.

The height of a datacell is implicitly transmitted by the distance between the transition storage door on one wire and the end of transitions tile along the wire just below.

Constructing Datacells. Next, the placement director will send subordinates down each table input row to construct datacells, reporting back as each is completed until it reaches the end of the table. Within the row, when the placement subordinate meets the datacell punchdown door it will traverse through the transition storage door adding doors to the west, wires to the south, and border tiles to the east, until it reaches the end of transitions tile below it. It will add the transition exit door to its east before returning to the top of the datacell, traversing through the affinity crossover to begin constructing the next datacell. This can be seen at a high level in Figure 2.23.

Copying Transition Rules. The process of copying datacell transition rules works as follows: Our agent transitions the transition row door into a copy activation state, which will swap with every unary tile, the end of data string state, and border tile state behind to transition each into a copy yourself state. Each will copy themselves onto the wire and then flip backwards until reaching the door and transitioning back into a normal state. The door will stay at the back of the row until the border tile returns, signifying it has copied itself. At this point, the door will traverse to the front of the row and activate the door below it to begin the same process. The door will also copy buffer tiles and includes rows that have no transitions in them and are comprised of filler blocks.

When this is complete the fill marker is moved to the next datacell. When the row is completed the copy agent will confirm its completion to the copy director indicating that the next row can be started.

On the opposite side the data strings are activated for placement, and due to special placement states this allows them to attach their own blank construction tiles to the east.

Copying Vertical Table Wires. As all of the necessary information has already been copied into the table, what is left is to copy the vertical wires, see Figure 2.25. The placement director simply sends filling agents down the top of each column that will traverse south and place

south wires and protective border tiles to their side as they go. When these agents reach the bottom of the table they reverse walk north until it reaches the top of the table where the placement director will absorb and check it off. As the placement director was at the final vertical wire when it finished sending the agents down it will wait there until the agent returns with confirmation that it has completed filling the southern wires. The placement director will then wait to proceed at each intersection until it has reached the control edge of the table. Once this occurs, the placement director will send a signal to the Copy Checkpoint that the copying of state transmission wires may begin. When a placement subordinate is traversing northward on the state lookup chute wires it will lock the appropriate doors to shut down the construction wires connecting chutes of different directions.

Copying State Transmission Wires. After the table is complete the state transmission wires are copied, see Figure 2.26. After the appropriate wires are set and locked, we begin with the wires entering from the east of the tile. The Copy Directors for any direction will skip copying any vertical/horizontal construction wire and tile/table edge crossovers.

2.7.7 Activating Tile and Determining State

Once the supertiles construction has finished the process of activating the supertile for use begins. First, the construction wires must be deactivated.

Construction Wire Deactivation. After construction phase 10, phase 11 is started. To begin, phase 11 locking unlocking agents are dispersed throughout both tiles with the copy and placement directors, checking off their respective tiles intersections after each phase 11 locking unlocking agent reports back from the construction wire it was sent down until it returns to where it started. The copy and placement directors will not unlock the outer edges of their respective tiles yet.

Reporting Construction Completion to Neighboring Construction Tile. After the tile is confirmed to be complete, construction wires are deactivated, and table is activated, a signal is

sent to the neighboring tile that was in charge of construction that it may reactivate most of its non-construction functions and its active state column, see Figure 2.27.

Activating Table. When it is confirmed that the construction wires have been locked, state transmission wires appropriately unlocked, and neighboring wires reactivated, the placement director will move into phase 12 where it activates the table edge and punchdown gadgets.

Requesting Tile State From Neighbors. After the supertiles construction is complete it will send out a Requesting State Agent to its neighbors along its state lookup output wire for each direction. Upon reaching the edge of the supertile it will meet doors that indicate the supertile has no neighbor but it may traverse through them anyway. After the supertiles state has been selected and activated the has “no neighbor doors" will be transitioned to standard tile edge crossover doors. If a transition request reaches these has no neighbor doors it will be dissolved on contact. Only the newly constructed supertile can transition its has no-neighbor doors and that of the tile next to it.

Transmitting Neighboring Supertile States. Once a New State Requesting Agent has entered the lookup chute in the active column in a neighboring supertile it will traverse to each row until it reaches a self-intersection with the active state for that direction. It will report the state of the neighboring tile back to the new tile table.

Selecting (and Deselecting) Possible States. After one of the New State Request - Neighbor Reporting Agents have won the table locking race it will enter the table and traverse each column. At the end of each datacell, in the southern slot of the vertical wire/datacell crossover gadget, there is an affinity (or no affinity) door that the agent may transition nondeterministically with to select the state of the new tile. This agent may traverse the row backward or forward at any time and may even completely exit the table and restart the locking race so that another directions New State Request - Neighbor Reporting Agent may enter the table and potentially select the state, see Figure 2.28.

Activating Column and Regular Table. Once a state is selected state column activation agents are sent to the north and south of the initial selection door, turning each incoming datacell punchdown door/vertical intersection crossover into an active superstate mode and storing the state in the state storage box at the top of the column, see Figure 2.29.

Only after a state is selected and column confirmed to be activated in both directions does the New State Request - Neighbor Reporting Agent Eraser Door Agent Activator spawn and traverse to the top of the table and over to the left edge where it changes each of the table control edges inner doors to its normal active state. This causes New State Request - Neighbor Reporting Agents to dissolve upon contact (leaving an omni-directional wire tile behind) and triggers the table outer doors to unlock.

Unlocking Neighboring Supertile and Testing for Neighbors. When the New State Request - Neighbor Reporting Agent Eraser Door Agent Activator has reached the table's south marker tile it will send an Unlock Neighbor Outline and Test for Neighbors Agent. This agent will also unclaim corners of the new supertile. The agent will traverse around the edges of both supertiles unlocking them or testing for neighbors if the doors say they have none.

Sending Out New Supertile State. When the Unlock Neighbor Outline and Test for Neighbors Agent returns through the southern construction wire door (which it may traverse due to its special state) it will change to a State Transmission Trigger Agent that will traverse the active state column and send out State Transmission Agents at each self-intersection. The first State Transmission Agent to reach the inside edge of the table will unlock it, see Figure 2.31.

2.8 Transitioning Tiles

The transition process starts once a state notification agent from a neighboring supertile reaches another supertiles table. The table is locked by standard procedure, and if the lock is successful, the agent is admitted into the table.

2.8.1 Finding Intersection

The wire this agent is entering on implicitly encodes the state of the neighboring supertile. To determine whether there is a transition, it needs to find the active state column. The active state column has special door states, so when the agent reaches the active state door, it is no longer able to traverse further into the table. Reaching this door ensures that the agent will have the chance to transition with the transition storage door below, which is only unlocked (if a transition exists) in the active state column. If there are no transitions then the transition storage area door will be in a no transitions available state, see Figure 2.32.

If there are transitions and the state notification agent does not choose to back out of the transition process and table altogether, then the process to prepare both supertiles for transitioning begins.

2.8.2 Transmitting Intention to Transition

First, it must be confirmed that the neighboring supertile that sent the state notification agent is still in the state it was when it sent the agent. The agent confirming the state locks the tile into its current state.

This process starts with the State Notification Agent transitioning into a Transition Preparation Agent—Confirm Neighbor when swapping with the unlocked transition storage area door in the active column. During the swap, the transition storage area door has “awaiting confirmation” appended to its state.

The Transition Preparation Agent—Confirm Neighbor traverses down the storage area wire and out the transition storage exit door. As it cannot swap with the south door of the datacell’s

bottom crossover door, thus we can ensure it exits out of the output wire in the same state and in the same direction it came from.

The Transition Preparation Agent—Confirm Neighbor will leave the table locked as it exits. If we find the neighboring table locked, the transition is rejected, and an agent is sent to reset the transition storage area door and unlock the table afterward.

Rejection of Neighbor Tile State Once the Transition Preparation Agent has locked the neighboring supertiles table and traversed the columns to the active state, it will enter through the transition storage door as within the other datacell. As the Transition Preparation Agent - Confirm Neighbor became Transition Preparation Agent - Confirm Self-Intersection when swap-transitioning with the current datacells transition storage door, it will then traverse to the bottom of the transition storage area where the self-intersection marker tile sits. If the self-intersection marker tile is instead a not self-intersection tile, the Transition Preparation Agent will reject the transition, traversing back to the transition storage entrance door and deselecting it, then walking out of the datacell and unlocking the table upon exit. Once it reaches the initial supertile, it will enter the table (still unlocked at that particular door) and remove the "awaiting confirmation" designation from the active states transition storage area entrance door, finally locking the table on its way out.

Confirmation Neighbor Tile State However, if the lookup is confirmed to be at the "self-intersection," we know the neighbor's state has not changed. Thus we instead send a Transition Preparation Agent - Neighbor Confirmed back to the originating supertile. In addition, we send a wire setting agent to open the way from our datacell down to the state lookup chute exit wire for the respective direction and an agent to follow that to trigger the transition selection gadget, see Figure 2.33.

Copying Transition Rules Each supertile begins copying its transitions largely using the same method as during attachment. First, the door is activated, which activates the unary (or filler) tile behind it, which then transition-swaps with the door into a spawn copy state, which will

spawn into the wire outside the door. Once the number has successfully copied onto the wire it will flip through its row until it hits the end of data row tile where it will return to its normal inactive state. Once the door has cycled through all of the data string tiles, it will shift to a finish buffer cycling state and stop opening, simply transitioning with buffer tiles so that they flip behind the data string as before until the beginning of the data string reaches the door again. Once this occurs, the current door will trigger the door below it to begin its copying process. This is continued until all data strings have been copied into the wire and are heading to the transition selection gadget, as in Figure 2.34.

Filling the Transition Selection Gadget. For supertiles transitioning with a neighbor to their east or north, a wire setting/locking agent must be sent by the Transition Director upon reaching the edge of the tile, as the transition selection gadget is on the opposite side of their lookup chute exit.

Once the Transition Director reaches the transition selection gadget, the entrance door and the first row will be activated to begin overwriting the blank filler tiles within the gadget rows. The fill door will allow one data string to go through before it needs to be unlocked by the Transition Director again. After each row is filled, the wire tile next to the door below it is unfrozen. As we are filling top to bottom, this ensures data strings do not fill improperly to doors below.

Selecting the Transition. Once all of the data strings are properly in their rows the transition director will activate the selection agent for its half of the transition selection. The agent will traverse on the transition border wire parallel to (and touching) the other supertiles transition border wire. The agents randomly walk up and down the transition selection wire.

Once both have been activated, it is possible for them to align next to one another at one of the transition rules and have the chance to select, see Figure 2.35. When a transition is selected, the two agents must double transition, instantly choosing the new state of their respective supertiles simultaneously. When determining the supertile of a state, the highest priority is if a row/column

has been selected in the transition selection gadget, and then if there is none, the active state column of the supertile.

The agents may also choose an abort transition row/column double transition option at any time.

Once a transition is selected, the data string is copied out of the appropriate row/column it was being held in and led by a transition director to the top state lookup chute wire. The transition director has special override authority at the table border to open this top wire and allow the data string to traverse through the table.

2.8.3 Transitioning States

Each supertile is in charge of completing its respective half of the transition rule.

Punch Down Mechanism and Data String Traversal. The data string will traverse the input wire however many columns are specified by the data string. The punchdown tile will transition with a 1 tile turning the 1 tile into an east wire tile, thus decrementing the data string, and the punchdown tile will tell the door handle to unlock the door. Once the door swaps and transitions with the end-of-data string tile, it will reset fully with its door handle, return to its locked state, and tell the punchdown gadget to reactivate through its handle.

Deselecting the Previous State. The data string ends in an end-of-data string tile that, when it is punched down, updates that column as the new state. Before this can occur, the old column must be deactivated. Thus 2 agents are sent, one to the east and one to the west to search for and deselect the old active column.

When one reaches the old active column, it will spawn agents to traverse north and south, deactivating active state doors. When they reach the bottom of the table, they report back to the Deselection Agent, who reports back to the state update pending door, waiting for confirmation of deselection (and no column from the other direction).

Activating New State Column. The activation of the column looks much the same as the deselection of the state column; see Figure 2.36. The state update pending agent (waiting next to the affinity door) sends agents north and south to activate each column door as the active state and fill the state storage tile at the top with the new active state.

If the column selected is the current active state column, then only the wipe transition selection gadget agent is sent.

Wiping Transition Selection Gadget. Once this has been reported to be complete, the door will spawn a wipe transition gadget agent, which will traverse through the new active state column to the bottom of the state lookup chute wire for its direction and travel to the transition selection gadget. Upon arrival, the agent will enter the transition selection gadget and trigger the rows/columns to cycle wipe each of the tiles in its slot. This works much like copying for the filler tile cycling; except instead, it is erasing values. After the wiping agent has checked off all slots to be complete, a recapture of the agent will be triggered. When recapturing the agent has been confirmed it can then exit the transition selection gadget, resetting the gadget door and any wires set/doors locked by the transition process at the edge of the tile.

Sending Out New State. Once the wiping agent has returned to the active state column, the column is triggered to send out the new active state at each self-intersection. The table is unlocked by the first state transmission agent that reaches it.

2.9 Metrics

As the vast majority of states are dedicated to the operation and copying of crossover doors we cover this aspect first. Next is agents and finally other gadgets states.

The number of states was calculated as follows:

Nearly all crossover gadgets have four doors, each of these four doors may have been set to one of two directions (standard and reversed) each one of these doors has the following states: active, waiting, open, reset, pushback, and locked. Thus, there are 7 states for standard operation over 8 possible doors making 56 per crossover gadget type as standard. All doors use the same handle set adding only 12 states to the total calculation. In total 42 crossover gadgets are necessary coming to 2352 states for standard operation.

Additionally, during the copying process the majority of crossover gadgets have a copy yourself state for each door (though the doors enter a state reset when complete unlike other copy processes) and a copy agent for 9 additional states per type. For placement we transition with the first blank tile it swaps with to a wire tile, then we overwrite the leftover second placement tile, and lastly each of the four doors are put into their respective blank tiles, adding 5 additional states (other agents will set them to standard or reverse as necessary). As such the copying process of each crossover door requires 15 states and thus 630 for the entire construction.

There are 16 non-crossover doors in the system that each have the above standard operations (and half have independent handles) adding to 120 states. Crossover doors have special states to indicate that there is no neighboring door, adding 96 states to the incoming and outgoing state transmission crossovers for each direction. Corner crossover doors can be claimed by a neighboring supertile, adding 64 states.

2.9.1 Agents

Agents locking and unlocking the table requires 30 agent states and 5 additional door states. Initiating transitioning, copying the data strings, and ensuring they reach the appropriate locations

requires an additional 31 agent states, and 24 door states. Selecting the new state within the gadget, copying out the data string for the new state, and wiping or aborting the transition requires 34 states.

2.9.2 Copying States

General Copy States and Agents. Each direction of placement tiles requires an inactive, active, complete, and special crossover double state, adding to 16 states.

Copying Crossovers. Each of the crossover doors (regular and reversed) must have a state indicating they should copy themselves, a state indicating to spawn the same agent, an agent which must traverse 2 steps to the center of the crossover gadget then check off that the doors (not reversed) have been attached at each side. As this requires 15 states and there are 42 crossovers, this adds 630 states.

Locking Agents. Each step that requires a locking agent needs a spawn/waiting state for the copy director and the locking agent itself needs an active, lock door 1, lock door 2, exit crossover, and locking complete state. Nearly every locking agent also needs the copy director and an unlocking agent with the same states for a total of 14 states. There are a total of 10 phases that require locking agents, but the state transmission wire construction needs these for each side. In addition, there are 30 other miscellaneous states that are used across various phases. This brings the total of these to 212 states associated with copy locking.

Placement Director. The placement director has an awaiting direction tile, an overwrite completed direction tile, a waiting state tile, an overwrite state tile, a waiting crossover agent completion, overwrite crossover agent, lock door 1, lock door 2, exit crossover, and complete states, making 10 states over 4 construction directions for 40 states. This standard version applies to 4 phases, but cycling is done 11 times due to subphases for a total of 462 states.

Additionally, the copy director and/or placement director will spawn placement directors or subordinate placement directors and wait for their completion 22 times. Doing this for 4 directions for 44 states per directions makes for 176 states added.

Aborting Process. The abort construction process (not including reactivation) takes 10 states to overwrite, wipe, and inform the copy checkpoint/director for each direction, adding a total of 40 states.

Traversing Opposite. In 13 phases and subphases the copy director must traverse to the opposite side of the tile. Adding 52 states.

Datacell Outlines. The subordinate prime placement director must be spawned, place a wire to the south, door to the west and a border tile to the east before moving on, when it runs into a no state tile to its south it will instead place an exit door and mark itself complete. As this doesn't depend on the direction, it only adds 6 states.

Filling Datacell. Copying each transition rule requires the copy director to activate each tile for copying, flipping through them without sending direction tiles at this phase; they will mark themselves complete in addition to the copy director, the placement director and tiles do in this in reverse on the opposite side. With the necessary checkpoints included this adds 22 states.

Vertical Table Wires. In addition to the check off states (counted above) the south traversal agents need to skip crossovers and the final one needs to lock on the way up adding 6 states.

State Transmission Wires. As each copy and placement director needs to check off first and last for each direction and crossovers need to be skipped there are 32 states added.

Reactivating Neighboring Supertile. The agent must delete the checkpoint, traverse to the top of the table to spawn a generic sub-agent that doesn't depend on construction direction, unlock the table, check for where the active state column is, spawn an activation agent, and let the newly finished tile know this process is complete adding 12 new states.

Activating New Table. In the new supertile the table doors must be moved into special door states added to the east and west of each table edge state transmission wire crossover. This adds 14 new states.

Requesting, Receiving, and Selecting States. Requesting and receiving states requires and agent to send them from each direction in the active state column, the state requesting agents themselves, and special state transmission agents. Selecting the state requires abort and select, if the state is not a full state then an additional special agent is required, adding 9 states.

Activation, Unlocking, and Transmitting. The activation of the new column, doing a special unlock of the table the self and neighboring tiles outlines and transmission of the new state adds 11 new states.

2.9.3 Final Count

There are an additional 40 miscellaneous states used in the construction bringing the total number of states to 4600, including 2600 non-copy states for our final ACA state count.

2.10 Correctness of Construction

Here we give proofs of correctness. We first (re)state our main lemma.

Theorem 2.4.1. *There exists a tile set* $(\Sigma_U, \Lambda_U, \Pi_U, \Delta_U)$ *such that, for all systems* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, 1)$*, there exists a* $\Gamma' = (\Sigma_U, \Lambda_U, \Pi_U, \Delta_U, s_U, 1)$ *that simulates* Γ *at scale* $O(|\Sigma|^3)$*.*

We prove this via the following lemmas which each satisfy a condition of simulation. We start with a helper Lemma.

Lemma 2.10.1. *For any assemblies* $A \in \textbf{\textit{PROD}}(\Gamma)$ *and* $A_U \in \textbf{\textit{PROD}}(\Gamma_U)$ *such that* $A = R^*(A_U)$*, any assembly* B_U *such that* $A_U \to_1 B_U$ *satisfies either* $R^*(A_U) = R^*(B_U)$ *or* $R^*(A_U) \to_1^\Gamma R^*(B_U)$*.*

Proof. An attachment can never change a mapping because if a supertile is incomplete it maps to the empty state. Once the datacell has been built it sends a signal to it's neighbors. Its neighbors will respond by sending an agent which walks into the table. If it reaches an intersection in the table where there is an affinity rule it immediately changes the mapping to the new state simulating an attachment. The next available transitions mark the remaining tiles in the active state column.

Until a superstate transition is selected none of the changes that can be made in the supertile change the mapping since they do not change the active state column. □

Equivalent Production.

Lemma 2.10.2. *For any assembly* $A_U \in \textbf{\textit{PROD}}(\Gamma_U)$*, the assembly* $R^*(A_U) \in \textbf{\textit{PROD}}(\Gamma)$*.*

Proof. Any producible supertile either (1) maps to a empty state, (2) has only an active column which signifies the state in Σ it represents, or (3) has an active column and a selected transition in which case it maps to the state after the transition.

We will use induction along with Lemma 2.10.1 to prove that all assemblies are producible. For our base case we consider the seed in both systems. We replace each tile in the seed s by supertiles representing that tile to get seed assembly s_U. Then by Lemma 2.10.1 every move we

make on assemblies A_U in $\mathbf{PROD}(\Gamma_U)$ creates an assembly B_U which represents an assembly B that is reachable by A in Γ. □

Lemma 2.10.3. *For all $A_U \in \boldsymbol{PROD}(\Gamma_U)$, A_U maps cleanly to $R^*(A_U)$ with 1-fuzz.*

Proof. The seed s_U we create maps cleanly to the original seed s as we only place supertiles in locations where tiles take place.

Each ghost tile is built from a neighbor boundary first. Once the boundaries are built, the ghost tile copies the contents of the supertile. It is not until the supertile is complete and has selected a state that it begins to attempt to build neighboring ghost tiles. Therefore each ghost tile is adjacent to at least one properly mapped tile. □

Equivalent Dynamics.

Lemma 2.10.4. *For all $A,B \in \boldsymbol{PROD}(\Gamma)$ such that $A \to^{\Gamma} B$, it holds that for all A_U such that $R^*(A_U) = A$, we have $A_U \to^{\Gamma_U} B_U$ for some $B_U \in \boldsymbol{PROD}(\Gamma_U)$ with $R^*(B_U) = B$.*

Proof. Consider any pair of assemblies $A,B \in \mathbf{PROD}(\Gamma)$ such that $A \to_1^{\Gamma} B$. Pick an arbitrary A_U such that $R^*(A_U) = A$. If this transition was achieved via an attachment the agent selects the active tile column by traversing the datacells at an intersection. It may also chose to not stop at the intersection and continue on or go backwards to select another tile. This allows A_U to achieve any attachment performed by A.

For transitions, all available rules will be loaded up into the transition selection gadget. If the two agents meet they may select the transition and instantly change the mapping of both tiles, transitioning from A_U to B_U based on our mapping. However, the non-deterministic process may not select a transition at all and will allow the agents to keep walking to select any transition, or abort. □

Lemma 2.10.5. *If $A_U \to^{\Gamma_U} B_U$ for some $A_U,B_U \in \boldsymbol{PROD}(\Gamma_U)$, then $R^*(A_U) \to^{\Gamma} R^*(B_U)$ or $R^*(A_U) = R^*(B_U)$.*

Proof. If a attachment or transition does not change its mapping then we satisfy $R^*(A_U) = R^*(B_U)$. For a ghost tile to transition to a valid mapped tile, it must have an active state column. This active state column is only build and actually activated if there was a neighboring supertile that had the appropriate affinity.

For a transition the agents must both match and find the same transition in order to change the mapping of the tile. Only proper legal transition may be placed in the table so all of these must be valid transitions from $R*(A')$ to $R^*(B)$. □

Transitivity of Simulation. Here we show the definition of simulation is transitive, and hence we may chain many simulations together. It is possible that chaining 1-fuzz simulations results in an increase in fuzz by a constant factor. However, in our case we preserve 1-fuzz which we prove in Theorem 2.10.7.

Lemma 2.10.6. *The definition of simulation is transitive. If each simulation is 1-fuzz and has scale factor larger than* 1 *then the resulting simulation has at most 3-fuzz.*

Proof. First consider a chain of k simulating systems where Γ_i simulates Γ_{i+1} for $0 \leq i < k$.

Condition 1 from *equivalent productions*, and both the *follows* and *models* conditions of equivalent dynamics are all preserved by the fact we may compose the representation functions.

The second condition of *equivalent productions*, namely the c-fuzz bound, requires more care as the fuzz of a simulation is not immediately preserved. However, we can ensure that the fuzz will be bounded by at most 3. At each simulation step, the size of a supertile is getting smaller by a fraction $\alpha \leq \frac{1}{2}$. Since each simulation has at most one ghost tile next to its valid parts of the assembly, every simulation can add at most one ghost tile neighboring the previous one, which is a fraction α smaller than the previous. Since $\alpha \leq \frac{1}{2}$, this geometric series in the plane can reach a distance of at most 3 from the original supertile. □

Even though chaining 1-fuzz simulations can lead to a simulation using 3-fuzz, chaining our specific construction would never lead to more than 1-fuzz.

Theorem 2.10.7. *Chaining our simulations results in a 1-fuzz simulation.*

Proof. The individual tiles of a supertile S would never go outside the boundingbox of S. Take an individual tile t on the edge of S. If we would chain simulations, t would be simulated using a supertile S'. Supertile S' would only build a new ghosttile outside of S if t would want to build outside of S. Since this never happens, chaining our simulation only results in 1-fuzz. □

Universality Results.

Theorem 2.4.4. *There exists a tile set* $(\Sigma_U, \Pi_U, \Delta_U, 1)$ *such that, for all systems* $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$*, there exists a* $\Gamma' = (\Sigma_U, \Lambda_U, \Pi_U, \Delta_U, s', 1)$ *that simulates* Γ *with 1-fuzz at scale factor* $O(\min((\tau|\Sigma|)^3, |\Sigma|^9)$.

Proof. Lemma 2.4.1 states that temp-1 is IU for itself.

Chaining these two simulations will still result in a 1-fuzz simulation as ghost tiles are only built where a new tile may attach. Our construction in Theorem 2.4.2 has 1-fuzz and the ghost tiles that attach do not have any other affinities with neighboring tiles. Thus the supertile simulating them in Lemma 2.4.1 will not place any additional ghost tiles. For the same reason any assembly which has no attachments will not build any ghost tiles and thus have no fuzz. □

2.11 IU TA Simulates 2D Asynchronous CA $N = 2$

Previously, a partial proof of 1D asynchronous cellular automata (ACA) being intrinsically universal was shown in [30]. Here, we apply techniques used throughout this paper to show two subsets of asynchronous cellular automata are intrinsically universal. We start by defining pairwise and block-pairwise ACA.

Asynchronous Cellular Automata An Asynchronous Cellular Automata (ACA) system is a 4-tuple $\Gamma = \{\Sigma, N, \Delta, C\}$, where Σ is a set of states, $N \in \mathbb{N}$ is the neighborhood of Γ, Δ is a mapping $\Delta : \Sigma^N \to \Sigma$ and C is a configuration that is a mapping $C : \mathbb{N}^2 \to \Sigma$. We refer to each mapping in Δ as a transition rule.

Pairwise ACA. A pairwise ACA is an ACA with one extra consideration. More formally, it is defined as a 4-tuple $\Gamma = \{\Sigma, S, \Delta, C\}$, where S consists of all possible subsets of size 2 between a cell c and adjacent cells in each cardinal direction, and Δ is a mapping $\Delta : \Sigma^s \to \Sigma$, where $s \in S$. If Δ is a mapping $\Delta : \Sigma^s \to \Sigma^s$, we consider Γ a block-pairwise ACA.

Note that these automaton are a subset of radius-1 ACAs since we can transform each transition rule in Δ into larger mappings that ignore the neighbors not included in the rule. However, this increases the number of rules by a factor of $|\Sigma|^3$ since we need to account for all possible configurations of the neighborhood.

Lemma 2.11.1. *Block-pairwise ACA is strongly intrinsically universal.*

Proof. Let R be a representation function for a given block-pairwise ACA system $\Gamma = (\Sigma, S, \Delta, C)$. Map each cell $c \in C$ to c' such that $R(c') = C(c)$, including a mapping for empty cells, in the same manner as a seeded TA system. As with the techniques described in earlier sections, each new cell c' stores state information about each of its 4 neighboring cells and the transitional information from Γ, ensuring cells are only able to change from some $R(c') = \sigma$ to $R(c') = \sigma'$ if there exists a valid transition rule from Δ to allow it. □

Lemma 2.11.2. *Dual transitions in Tile Automata can be simulated by single-sided transitions at constant scale with $O(\Delta_2)$ additional states where Δ_2 is the number of dual-transition rules in the system.*

Proof. This was previously addressed in [4] in relation to the signal tile model. Figure 2.37 gives a general overview of how to do this simulation at scale-3 with an additional 53 states. Basically, all tiles around the two 3×3 macroblocks change before changing the states. This locks them into the transitions, and is reversible until state b_8 changes to x_{17}. The x's then change to y's after the $A \to C$ and $B \to D$ change. The y's then turn to c's and d's. □

Theorem 2.11.3. *The Asynchronous Cellular Automata model with a cardinal-direction neighborhood of size-2 and radius-1 (pairwise ACA) is strongly intrinsically universal.*

Proof. Pairwise ACA is a special case of block-pairwise ACA. However, any cell transitions based on its neighbors. Thus, all transitions are single-sided in terms of Tile Automata. Thus, we modify the block-pairwise IU result from Theorem 2.11.1 to only use single-sided transitions through scaling as shown in Lemma 2.37. This means that there is a constant-size set of states that is intrinsically universal. □

2.12 Conclusion

We showed that no passive or freezing tile assembly model can be non-committal intrinsically universal. However, we showed that the seeded Tile Automata model, with its unbounded state changes, is non-committal intrinsically universal. This is done by showing TA is intrinsically universal even under temperature 1 using 1-fuzz. Moreover, a Tile Automata system using temperature $\tau > 1$ can be simulated using a system that uses temperature at most 1. Chaining these two simulations shows that there exists a tile set that can simulate any Tile Automata system. This intrinsic universality result has direct implications for certain Cellular Automata. Moreover, the result directly implies that the original aTAM model can be simulated using Tile Automata.

There is significant room to optimize and minimize the tile set. For example, the number of tile states necessary to copy a supertile is large, whereas big sections of the supertile will always be the same, independent of what system we are simulating. Furthermore, the temperature simulation, and consequently the universal simulation, uses a lot of states. It might be possible to combine both simulations into one, by storing the affinity strength in the datacell. A ghost tile would then need to check all neighboring supertiles for their affinity strengths and add them up, before deciding which state it will become.

Another obvious open problem is that of dimensions other than two. It is still unknown whether the Tile Automata model is intrinsically universal if you extend the model to one, or to three or higher dimensions. Even though our simulation could technically simulate a one dimensional tile set, the supertiles would still use two dimensions themselves.

Finally, as in the aTAM model, our construction heavily relies on the fact that (locally) only a single tile can attach at a time. Because of this, our current construction only shows the seeded Tile Automata model to be intrinsically universal. Hence, the question arises whether or not the non-seeded Tile Automata model is intrinsically universal.

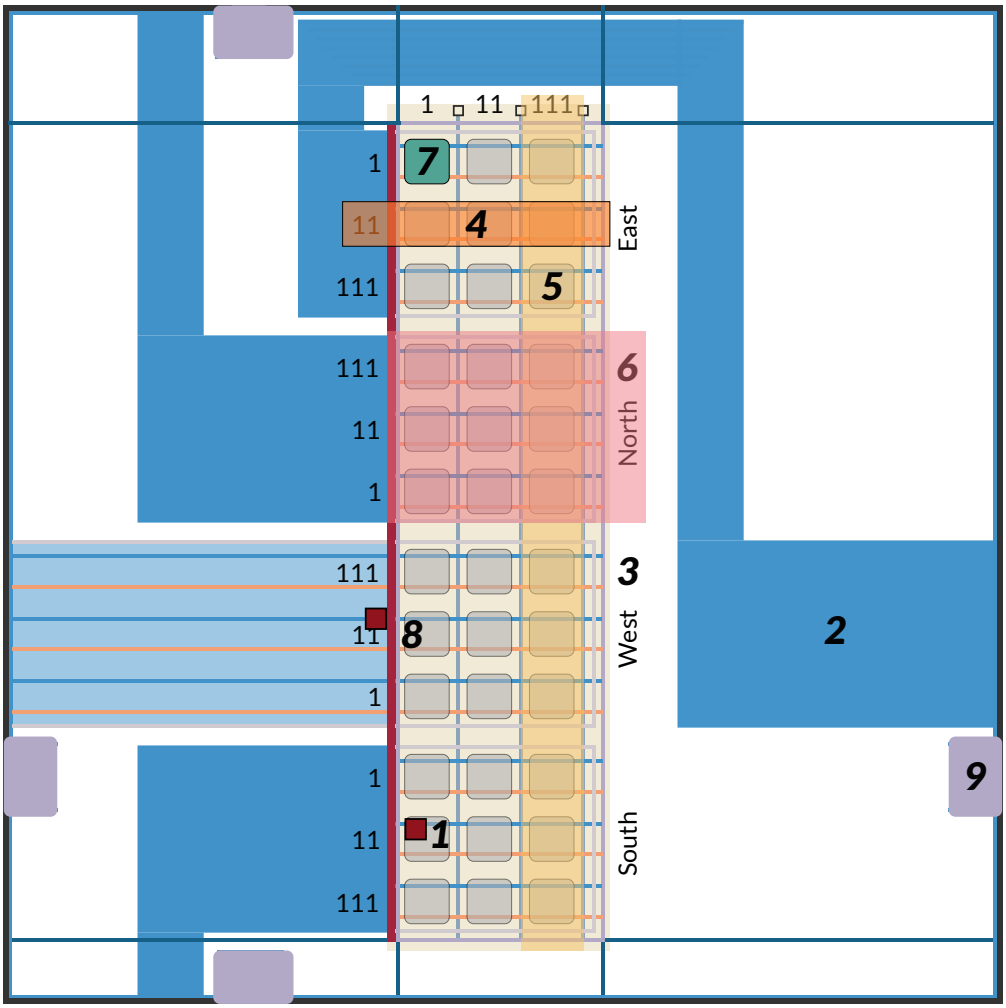


Figure 2.2: An overview of a supertile. (1) An agent inside of the supertile. (2) Wires connecting supertiles from each edge to the lookup table. West wires are drawn individually. (3) The lookup table storing the information about the system being simulated. (4) A row containing the information about the state of the east neighbor of the supertile. (5) The active column, representing the current state of the supertile. (6) A group of datacells storing all information for the north side. (7) A single datacell, in this case, storing the affinities and transitions for when both this supertile and the East supertile are in state 1. (8) The table control edge, with an agent waiting to enter. (9) Transition selection gadget at each edge, dictating the transition of this supertile with its east neighbor.

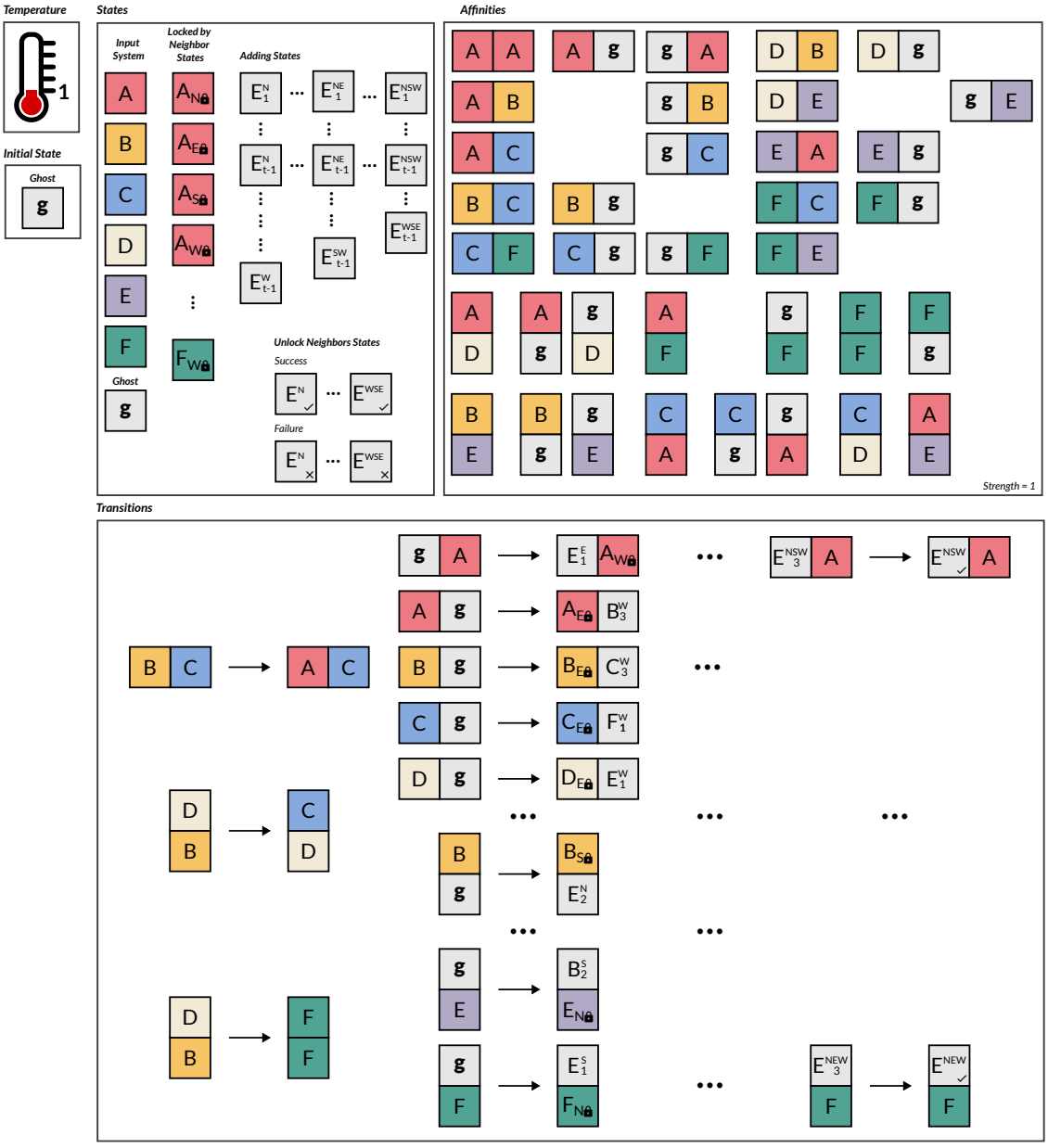


Figure 2.3: The temperature-1 system that simulates the system in Figure 2.1.

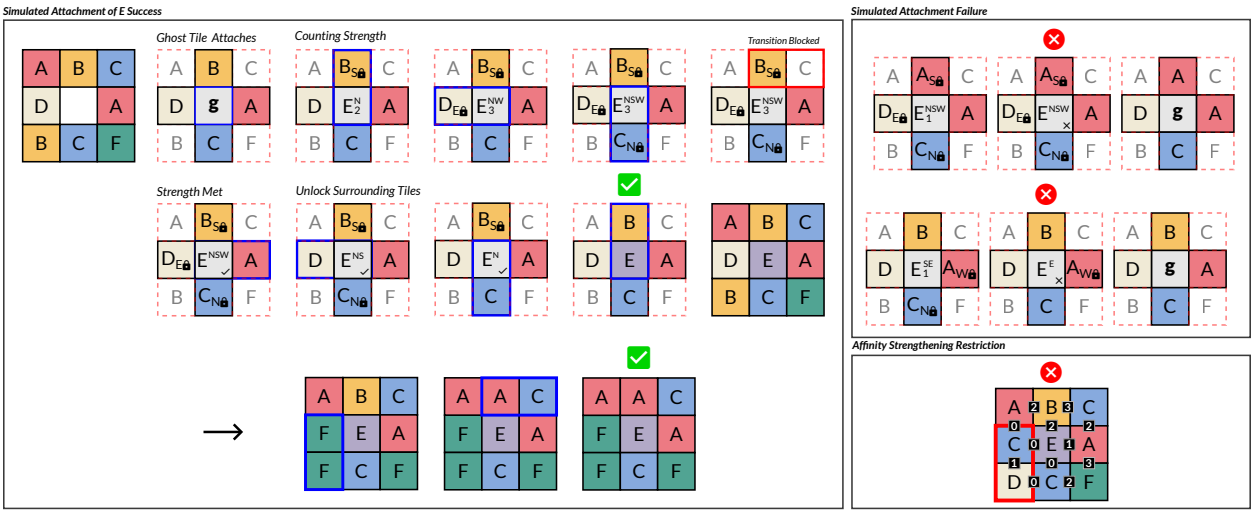


Figure 2.4: The construction process that the Tile Automata in Figure 2.3 builds, representing the same attachments and transitions as in Figure 2.1

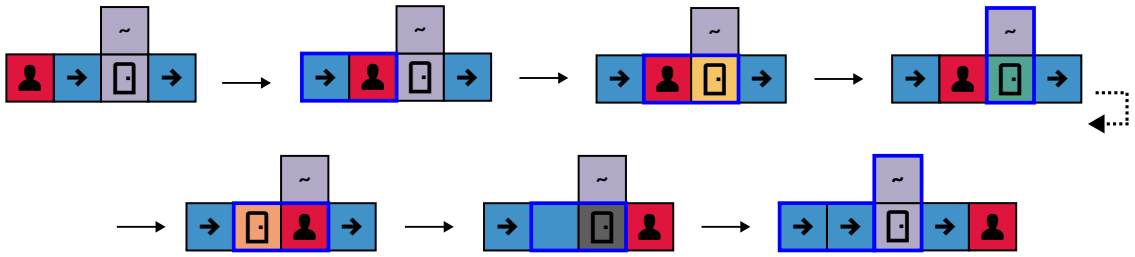


Figure 2.5: The door in action. Once an agent asks to pass the door, the door first confirms with its handle, after which it goes into an open state. The agent can then pass the door. The door goes into an orange warning state, after which it is only allowed to swap with a wire tile to go back to its original position.

Figure 2.6: Left: Standard crossover gadget. Right: Agent traverses a crossover gadget horizontally. The red doors are locked, and the green door is open.

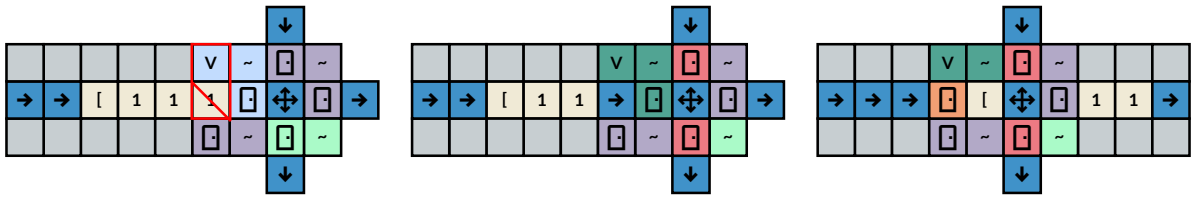


Figure 2.7: The punchdown gadget first decrements a data string by turning the 1 into a wire tile (Left). Then, the associated door is unlocked, after the north and south crossover doors are locked (Middle). Lastly, the punchdown door resets after the end-of-data string tile swaps with the door (Right).

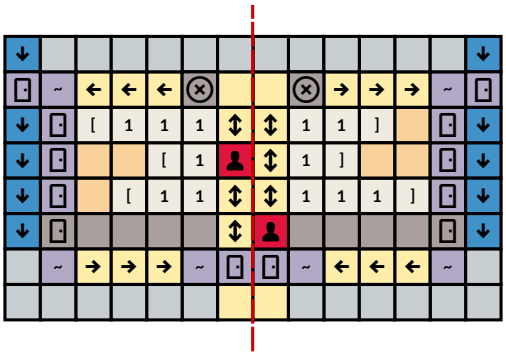


Figure 2.8: The transition selection gadgets of two neighboring supertiles. The border between supertiles is depicted in red. The agents non-deterministically walk up and down and can eventually select a transition or abort by transitioning with each other.

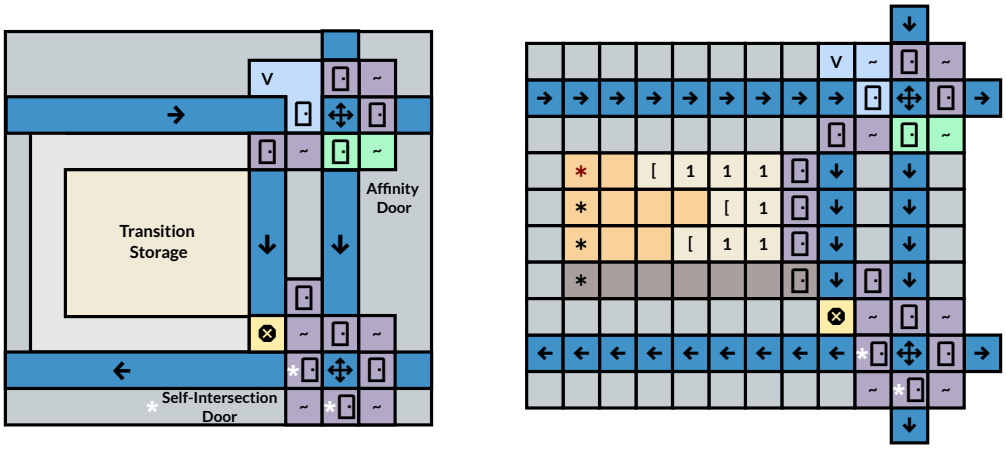


Figure 2.9: An overview of a datacell.

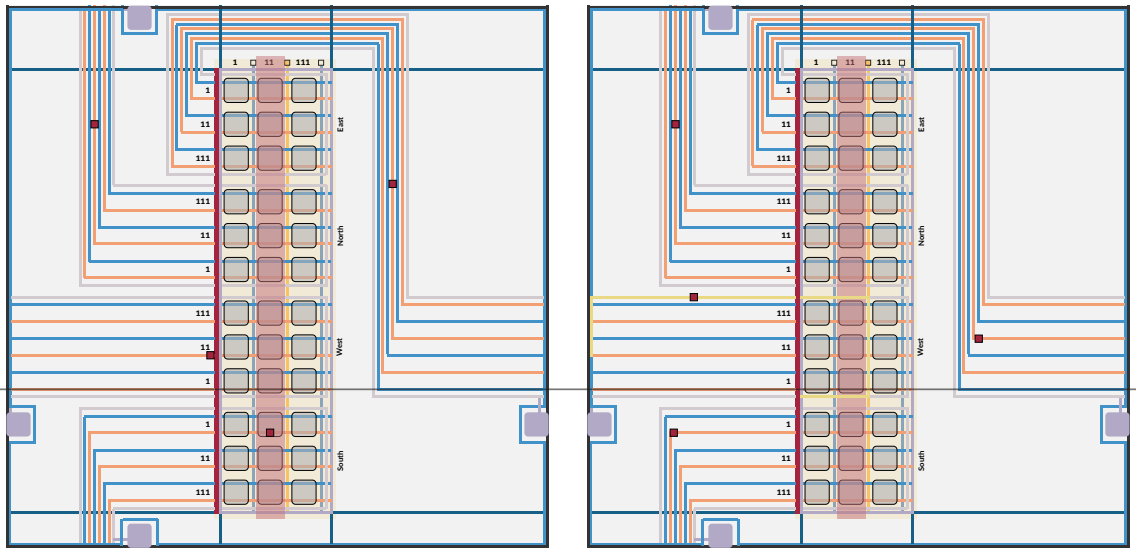


Figure 2.10: Left: The tile transmits its state to neighboring tiles and discovers it has no neighbors. Right: The west agent has moved over to the affinity selection wire (in yellow).

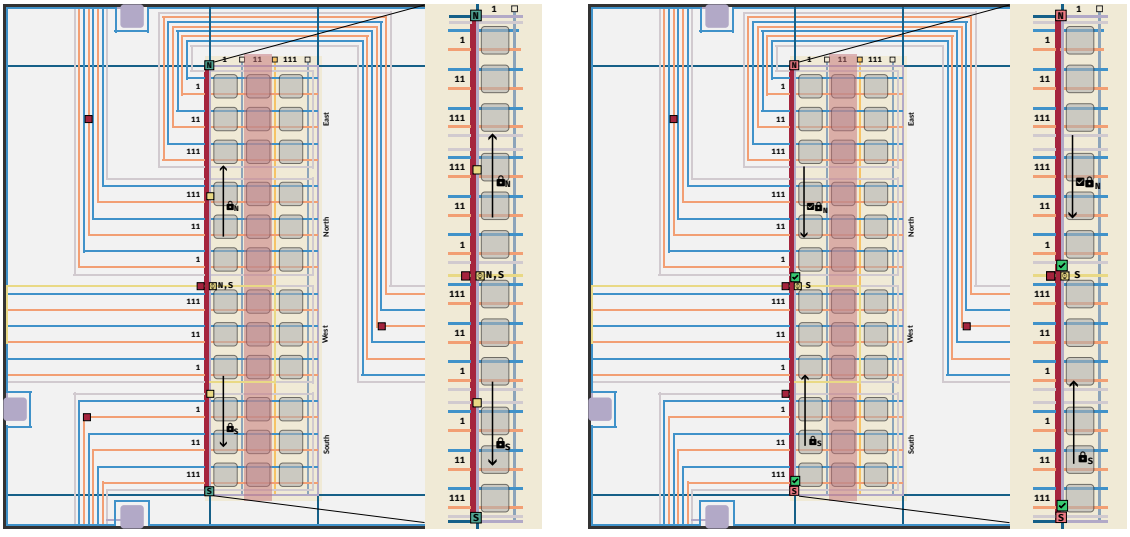


Figure 2.11: The lookup agent reaches table and locks the table

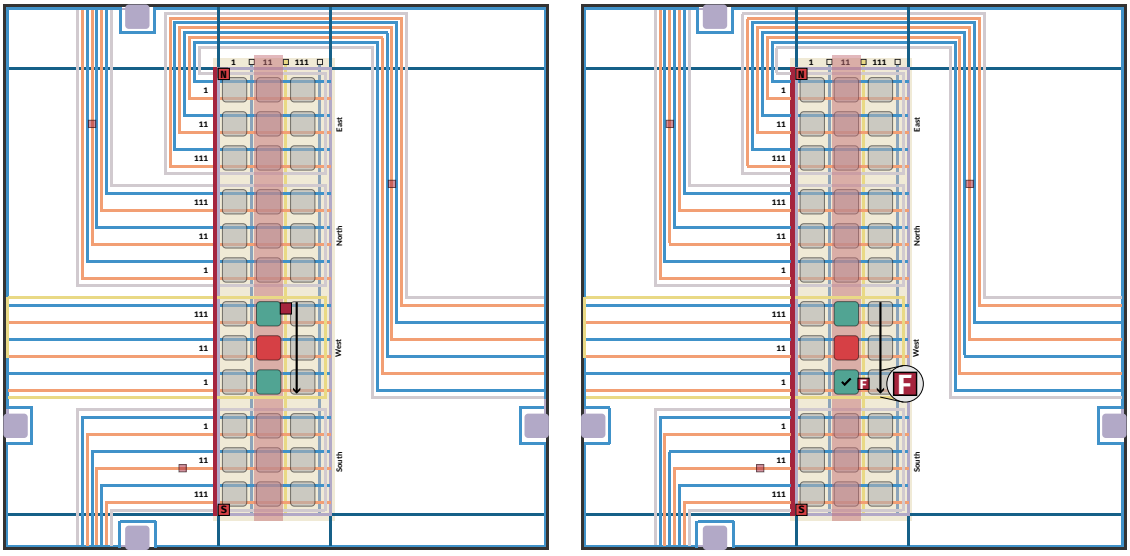


Figure 2.12: The lookup agent reaches the intersection with the active state and confirms there is an attachment possible for that direction.

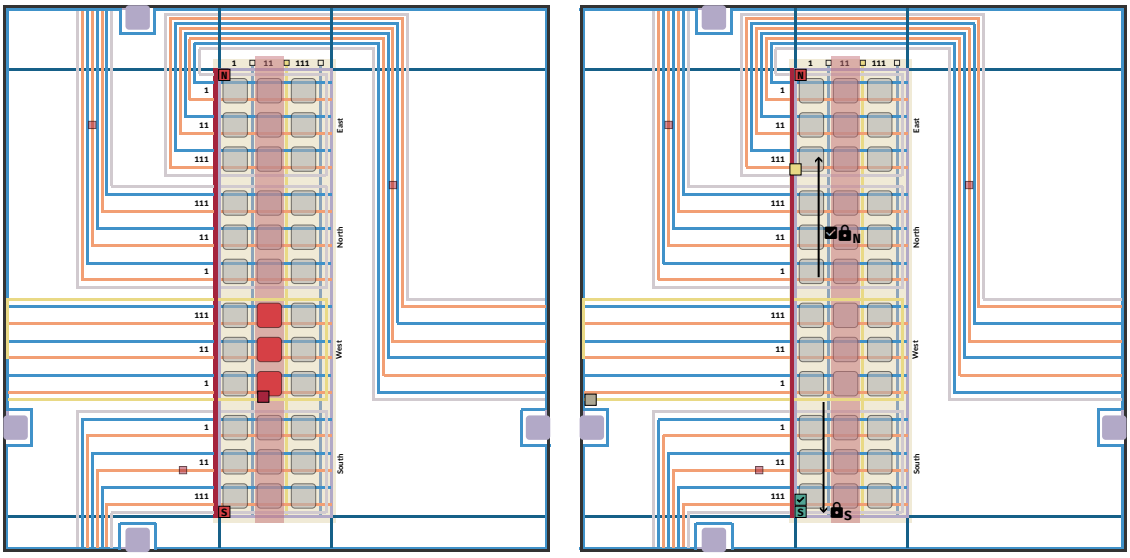


Figure 2.13: The lookup agent finds no attachment and unlocks the table, deleting itself when it reaches the edge of the supertile.

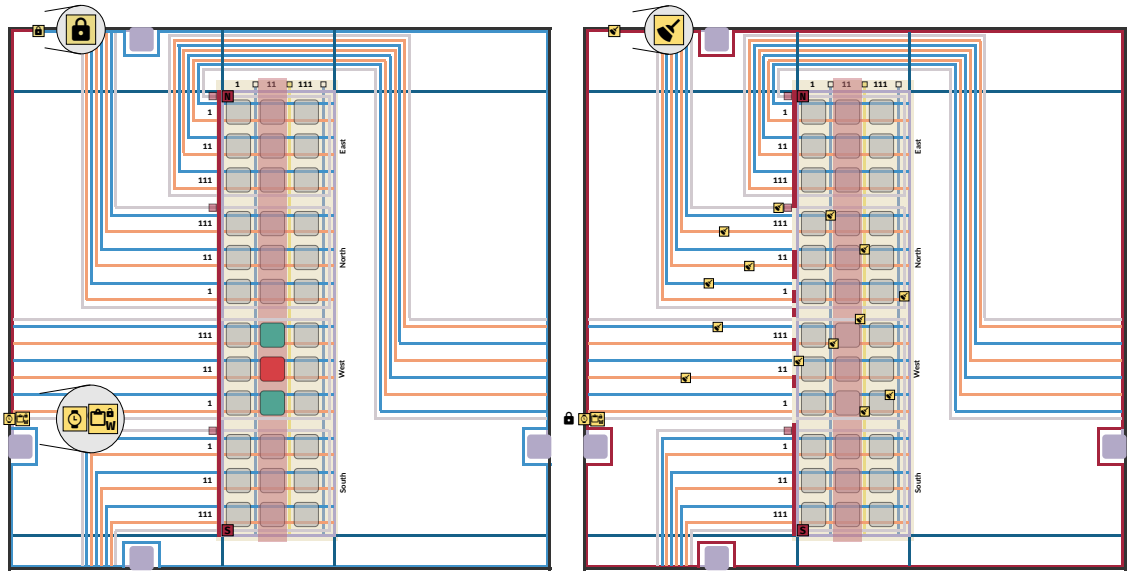


Figure 2.14: Copy Checkpoint (West) begins construction by locking then resetting/wiping the supertiles interior.

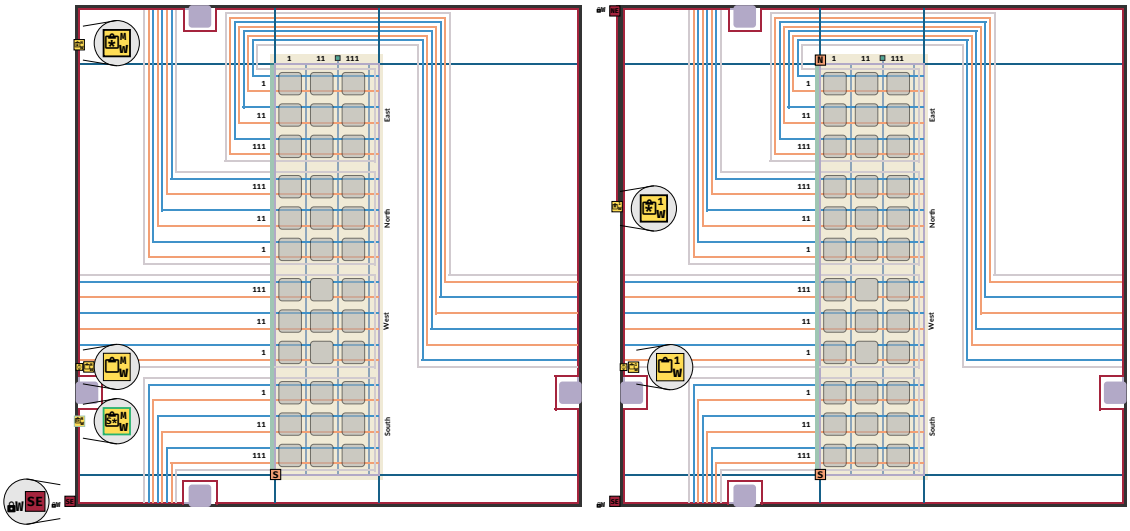


Figure 2.15: The Copy Checkpoint sends 2 claiming agents to claim the mirror edge. The Copy Checkpoint then sends a mirror edge agent to place the mirror edge.

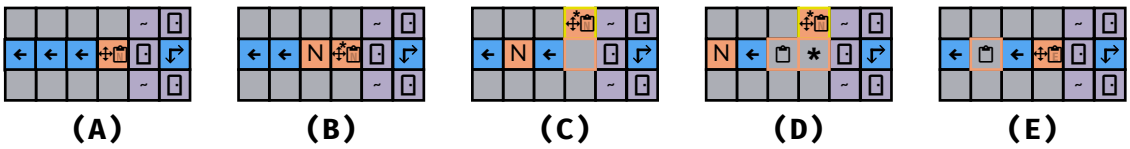


Figure 2.16: The general copy process.

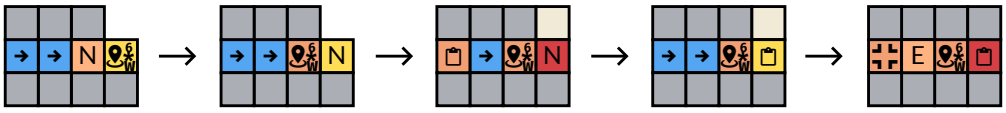


Figure 2.17: The placement process of a border tile up to the arrival of a crossover construction agent just before the previous border tile is deleted.

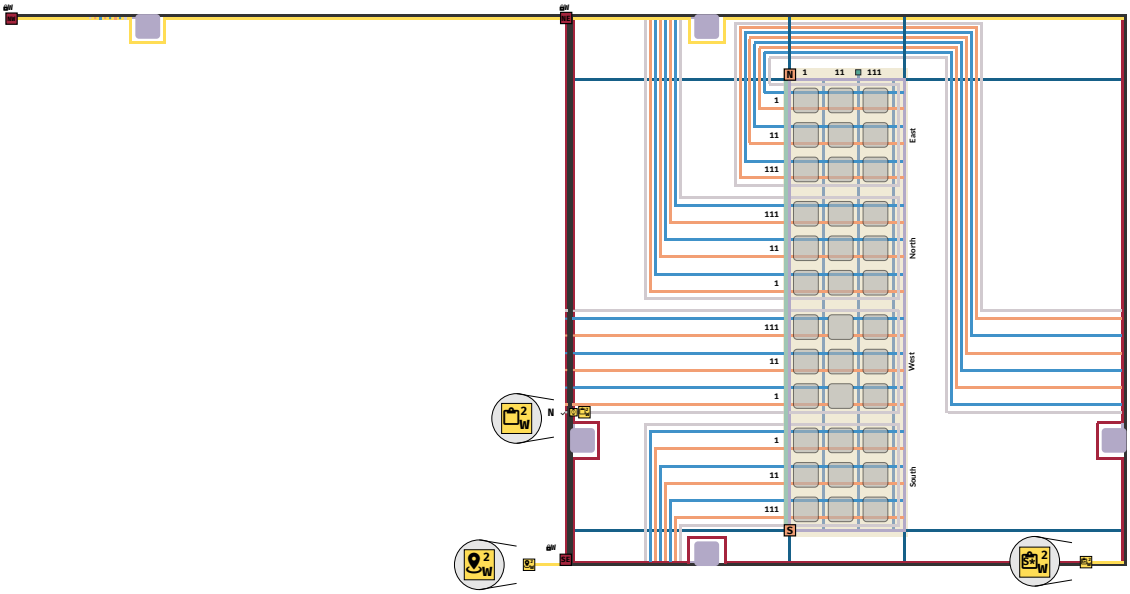


Figure 2.18: The copy director copies each adjacent edge. It first does the north, then the south edge.

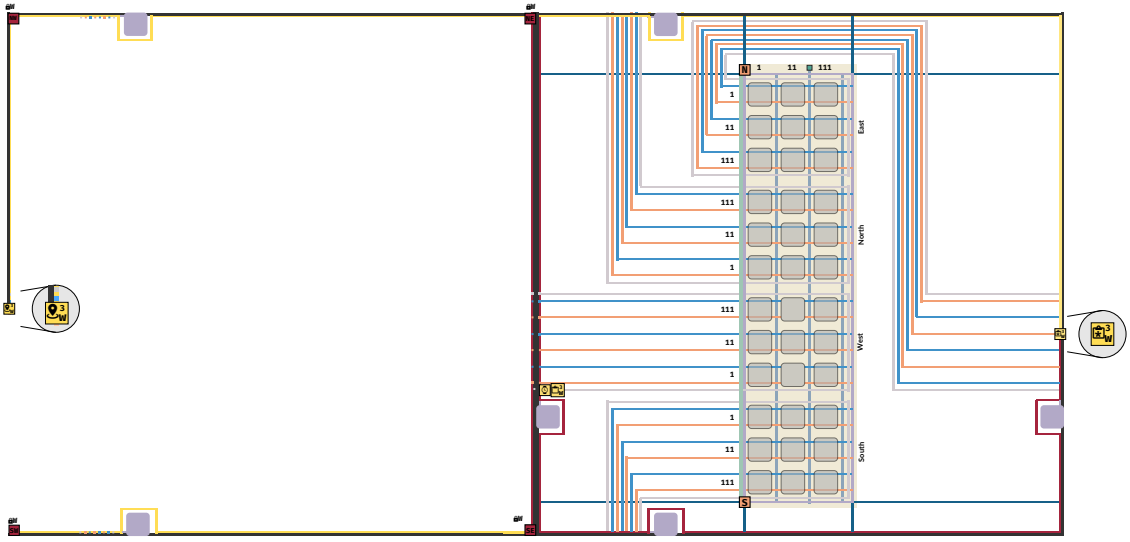


Figure 2.19: The copy director and placement directors copy the far side edge.

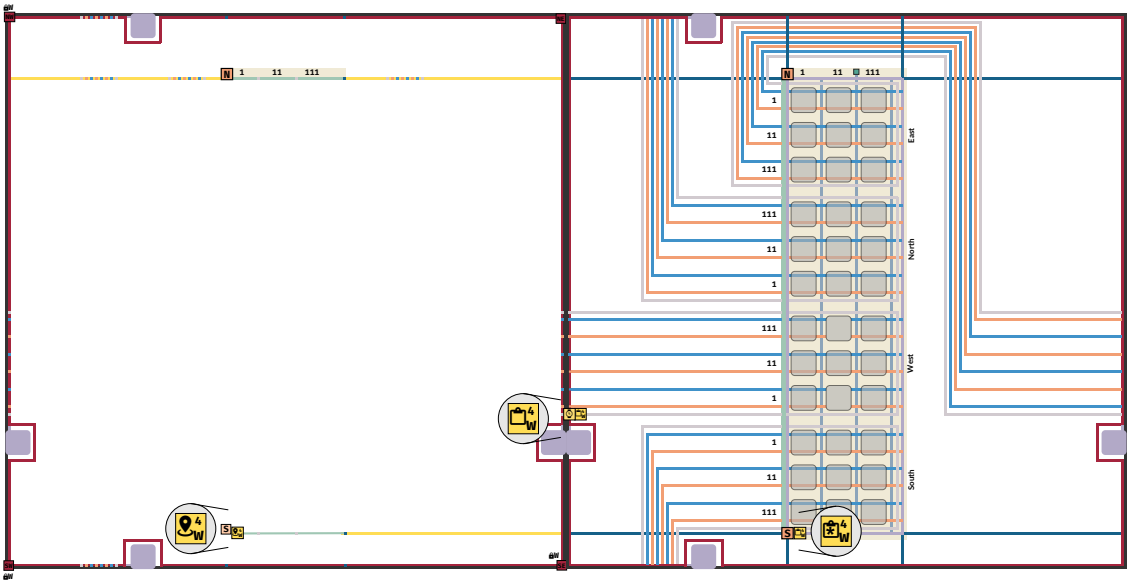


Figure 2.20: Copying the horizontal table outline.

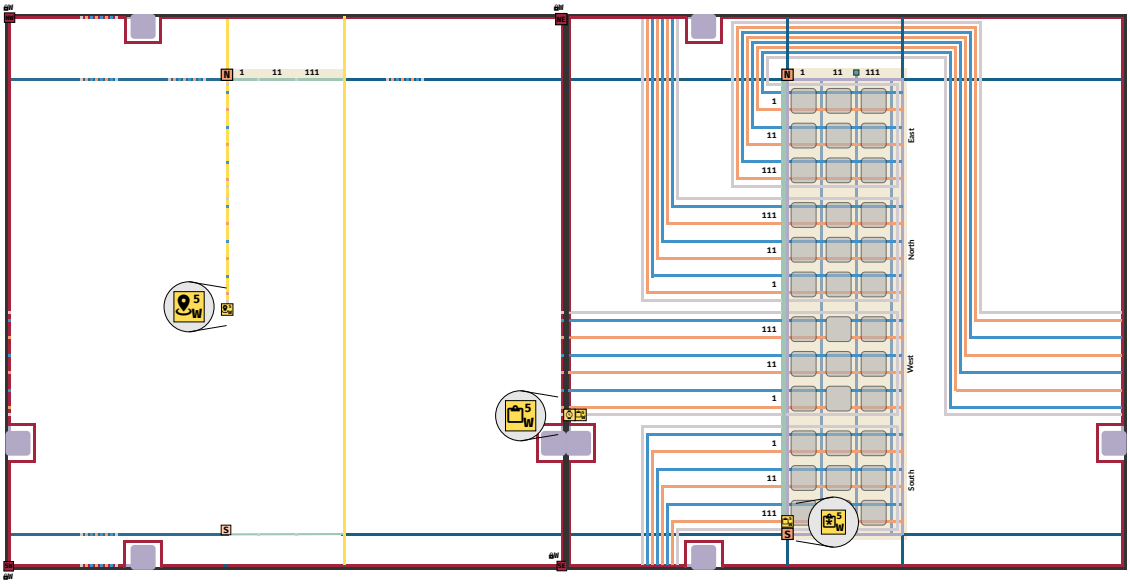


Figure 2.21: Copying the vertical table outline.

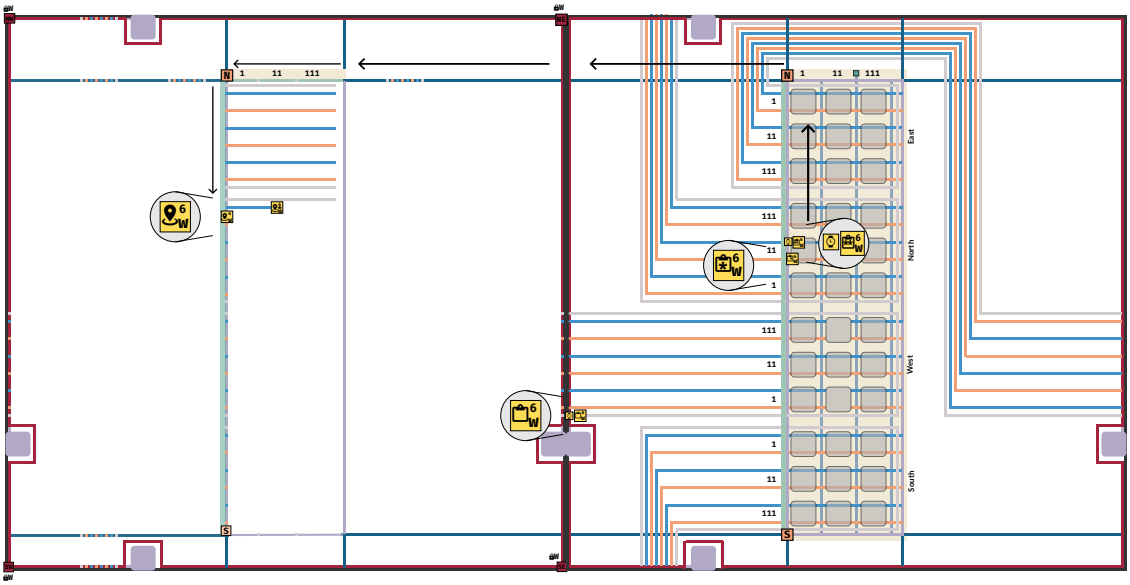


Figure 2.22: Copying the table row wires.

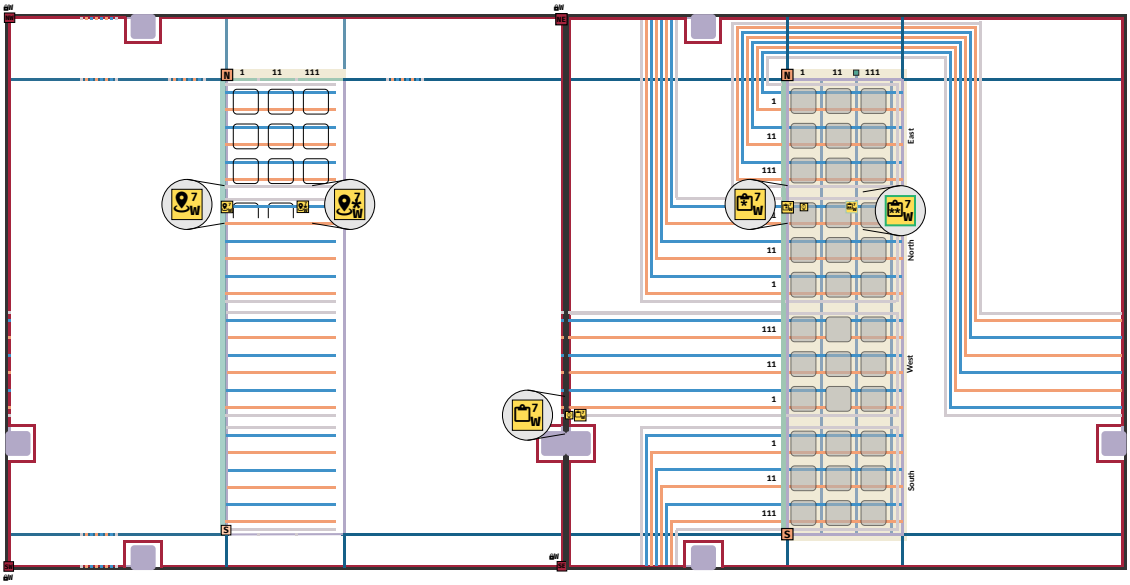


Figure 2.23: Constructing Datacell outlines.

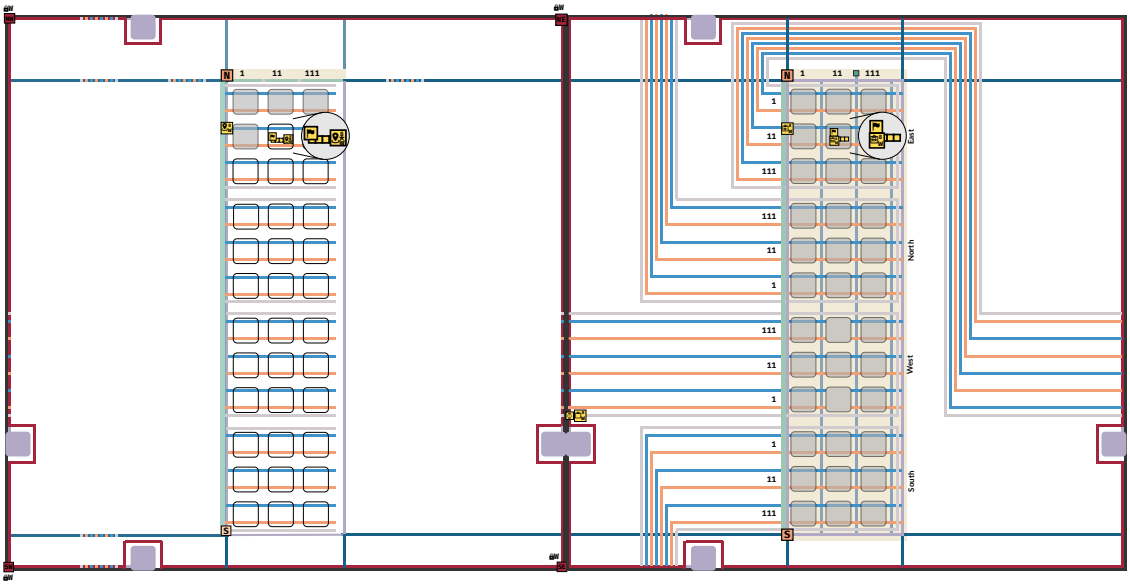


Figure 2.24: Filling datacells with transition rules.

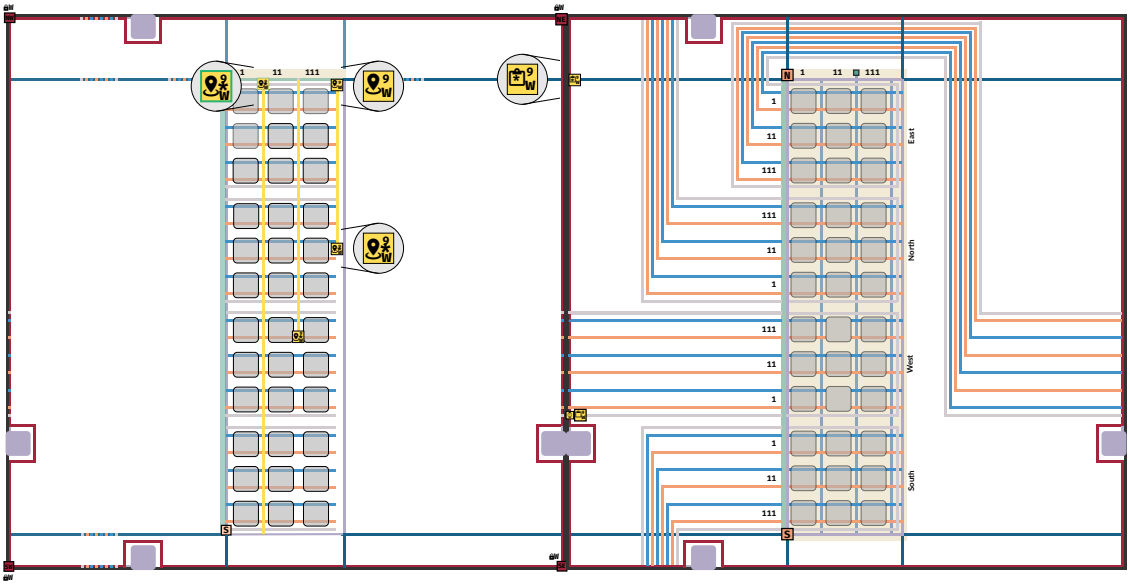


Figure 2.25: Constructing vertical table wires.

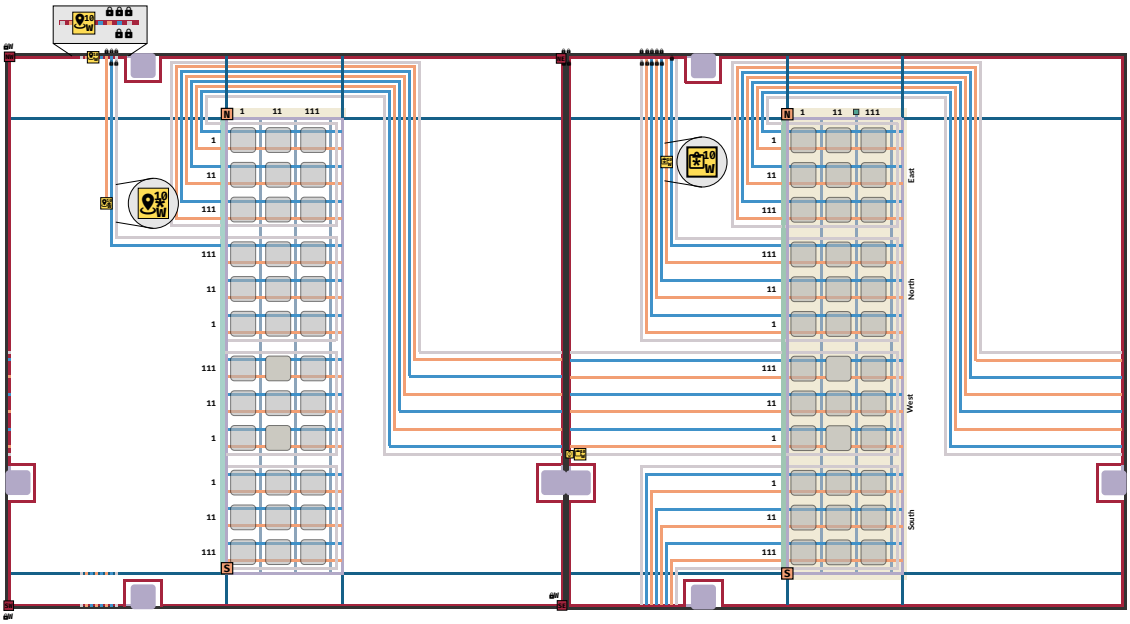


Figure 2.26: Constructing state transmission wires.

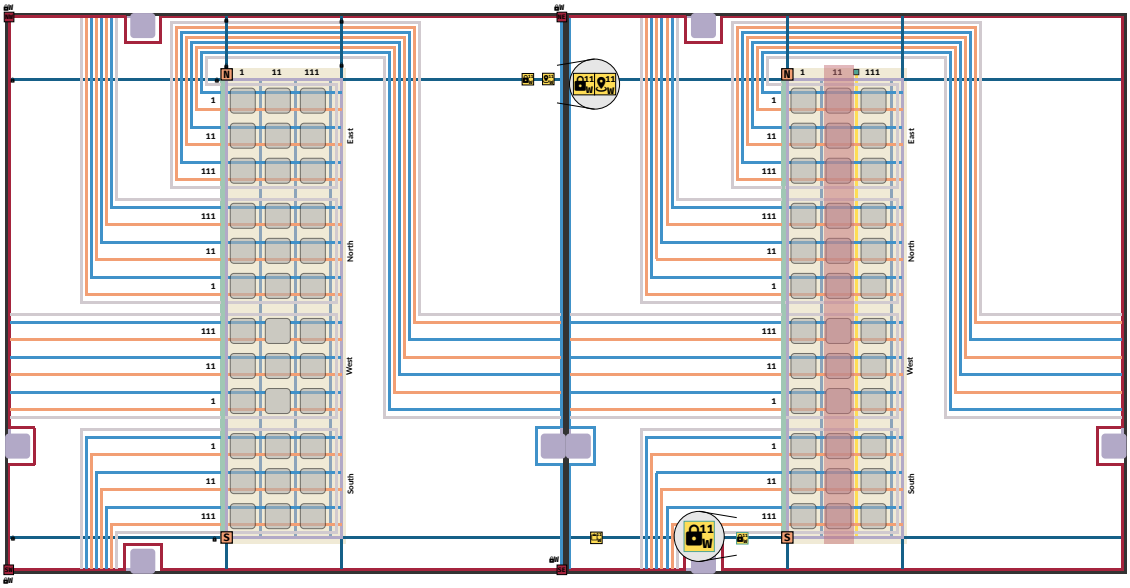


Figure 2.27: Locking construction wires and reactivating neighboring supertile.

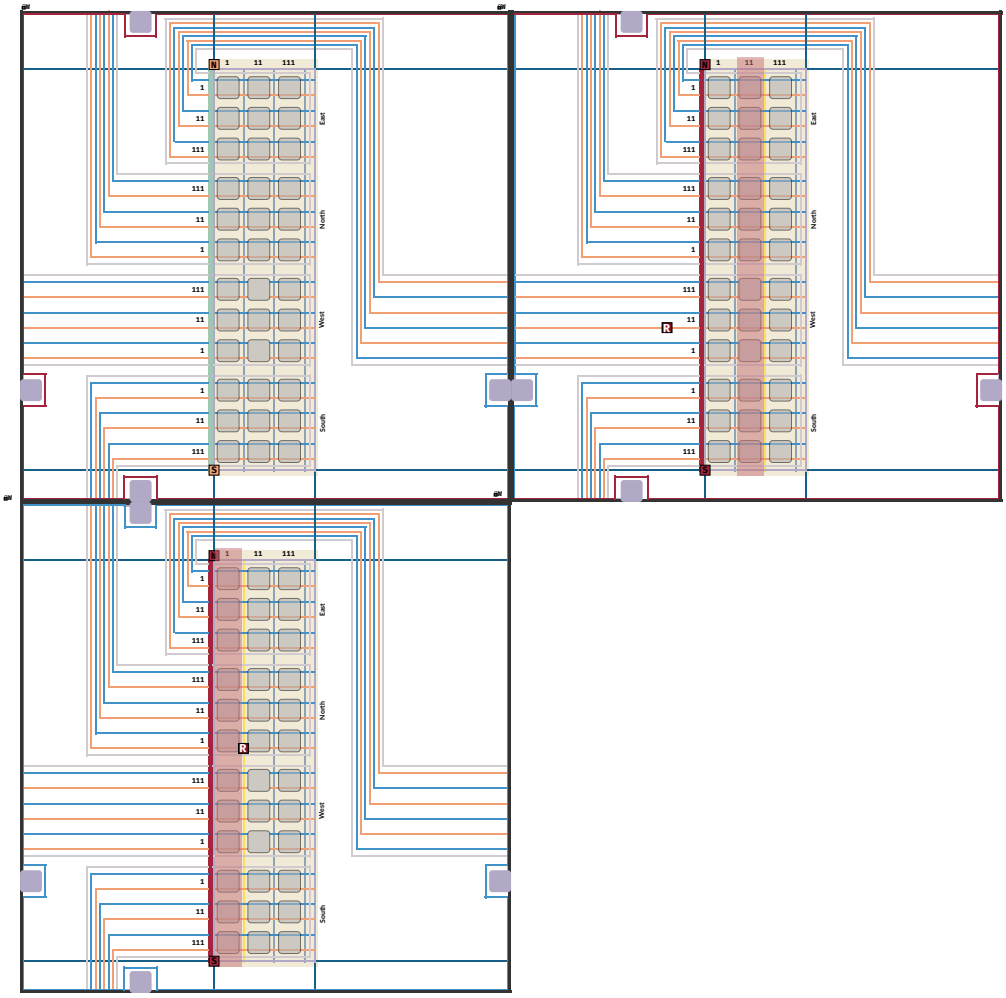


Figure 2.28: Receiving states from neighboring supertiles.

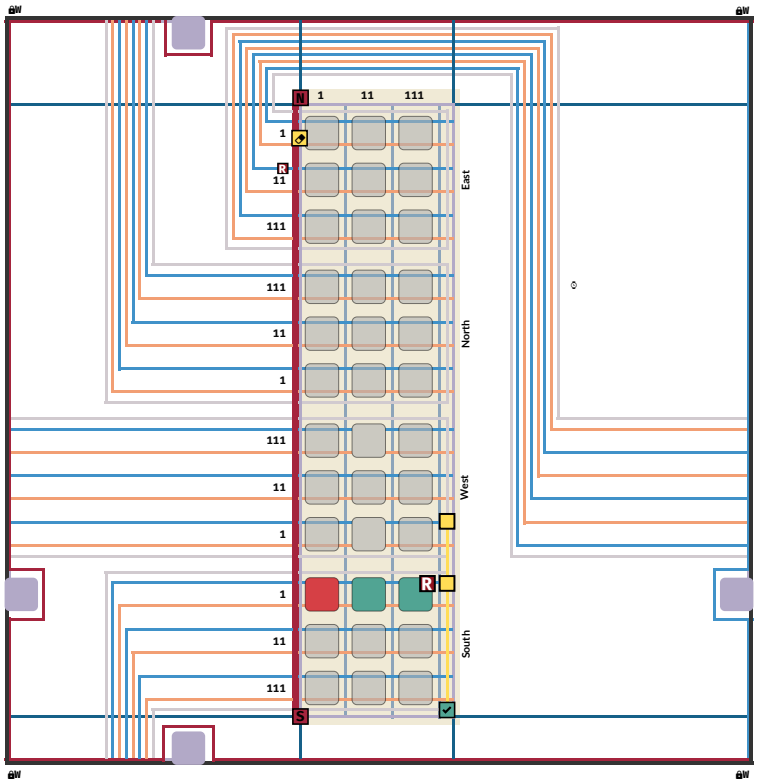


Figure 2.29: Selecting the state of the supertile.

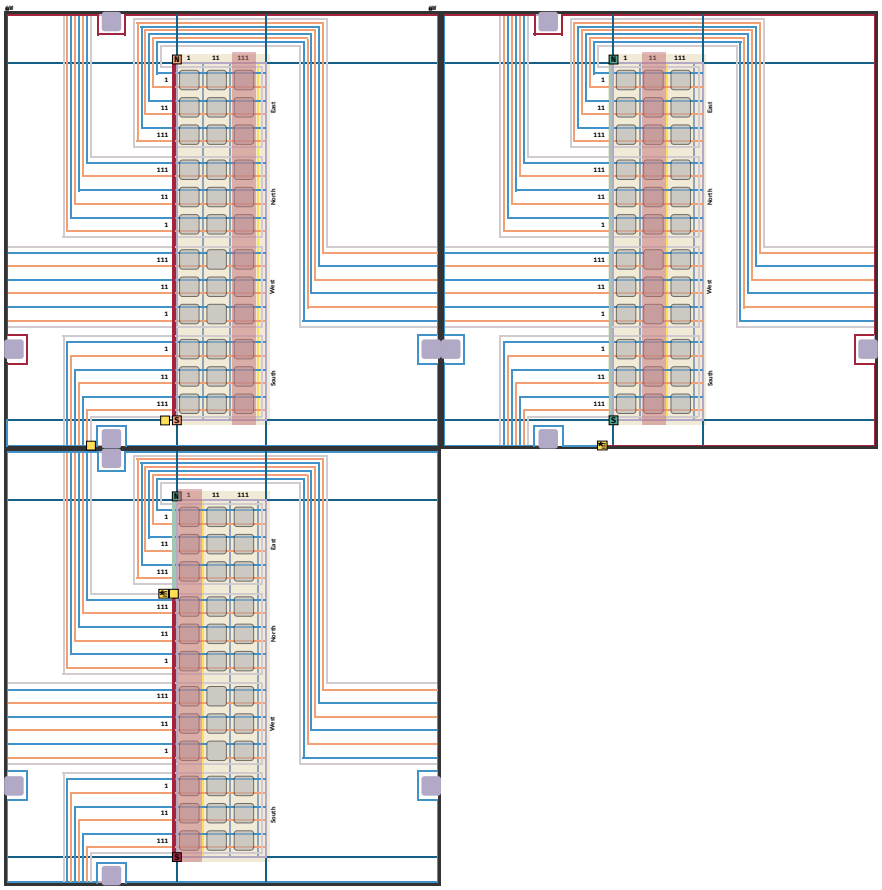


Figure 2.30: Testing for neighbors and unlocking supertiles.

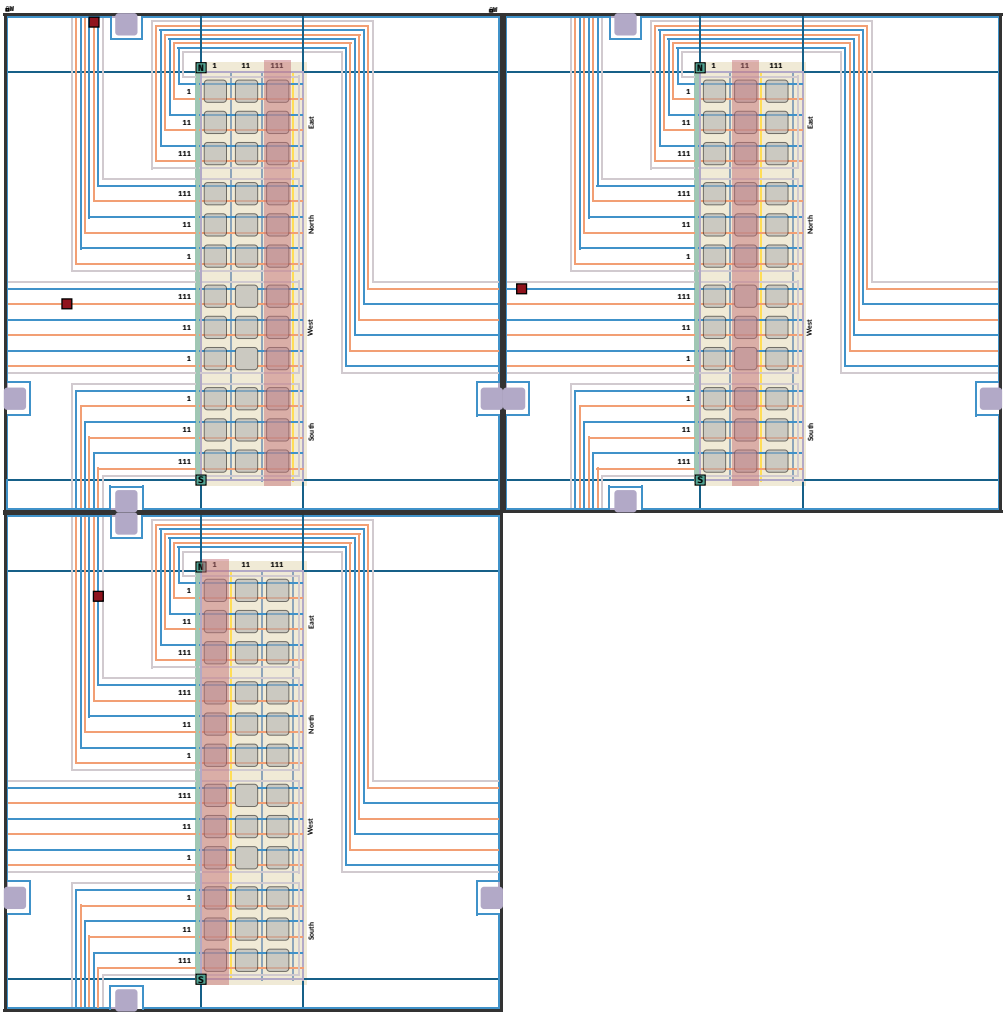


Figure 2.31: Sending out new state to neighbors.

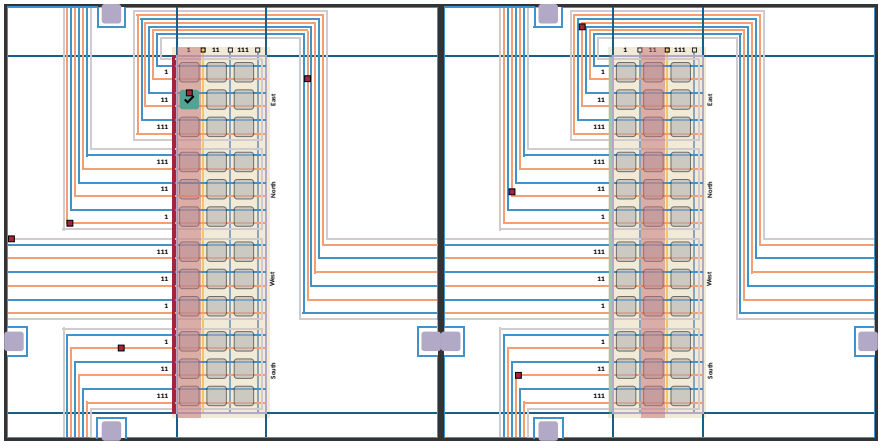


Figure 2.32: An agent discovers the existence of a transition with its neighbor.

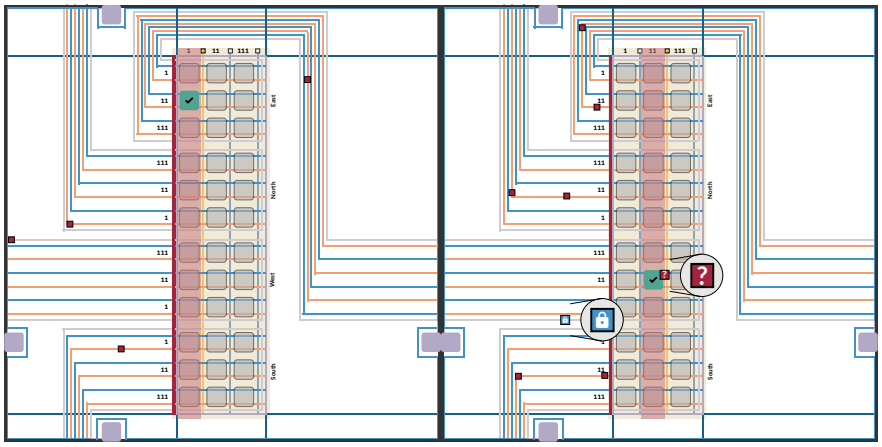


Figure 2.33: The agent checks whether the neighboring supertile is still in the same state and locks the neighbor's table.

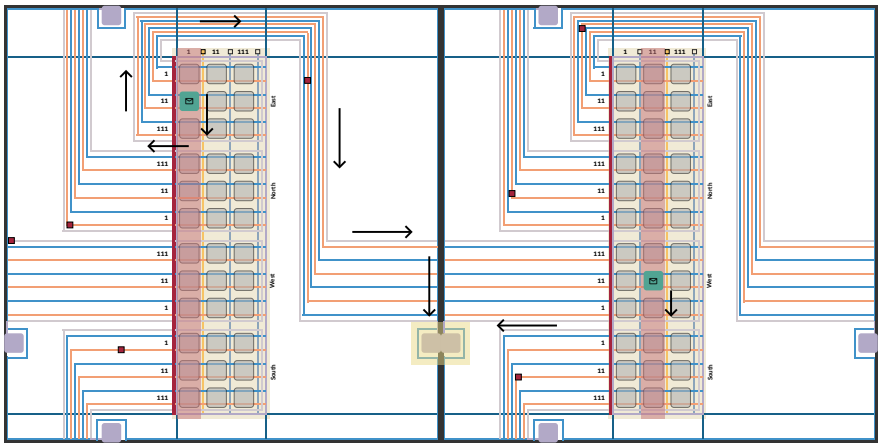


Figure 2.34: The transitions are sent to the transition selection gadget.

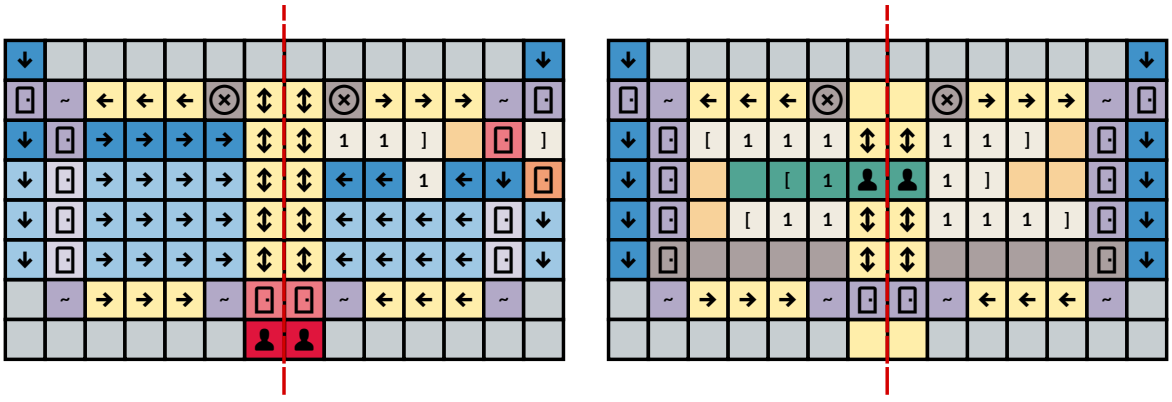


Figure 2.35: Left: The transition selection gadget is filled row by row. Right: Transition Selection gadget selects a transition to take.

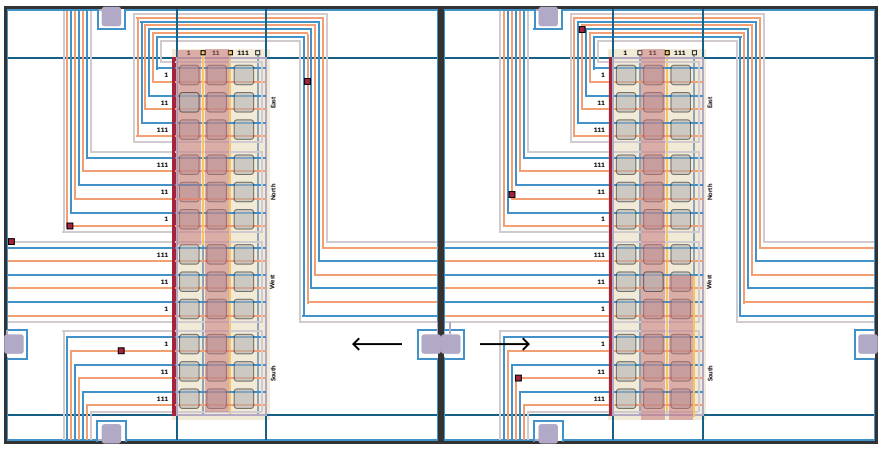


Figure 2.36: The supertiles independently transition by first deselecting the old column and then selecting the new one.

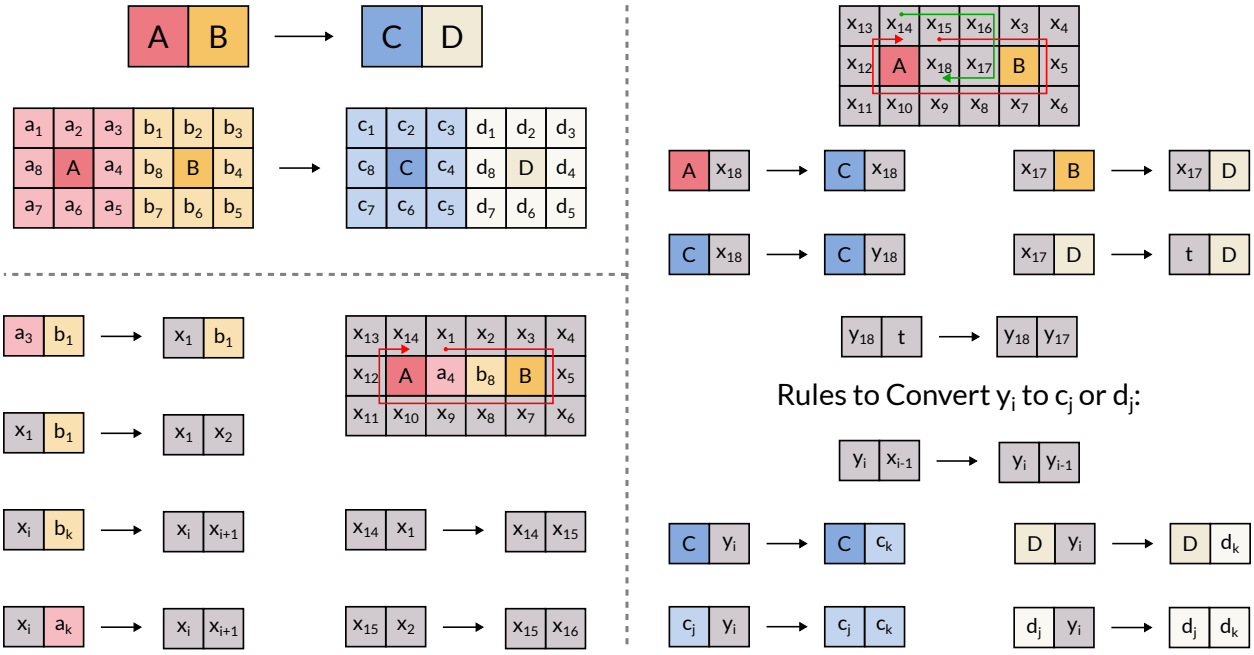


Figure 2.37: Simulating a dual-transition rule with only single-sided transitions. We scale the simulation by 3 and any transition occurs by "locking" the two tiles, transitioning the two tiles, and then unlocking them. The rules shown are a general idea, but it requires an additional 53 states.

CHAPTER III

OTHER TILE AUTOMATA RESULTS

3.1 Building squares with optimal state complexity in restricted active self-assembly

My Contributions In this paper I originally produced a lines result showing that of length $O(2^n)$ lines could be built using $O(n)$ states which was later pulled from the paper. Additionally, I helped program AutoTile, our Python simulator, especially the Universal Classes file. I also helped create figure 1.

Abstract Tile Automata is a recently defined model of self-assembly that borrows many concepts from cellular automata to create active self-assembling systems where changes may be occurring within an assembly without requiring attachment. This model has been shown to be powerful, but many fundamental questions have yet to be explored. Here, we study the state complexity of assembling $n \times n$ squares in seeded Tile Automata systems where growth starts from a seed and tiles may attach one at a time, similar to the abstract Tile Assembly Model. We provide optimal bounds for three classes of seeded Tile Automata systems (all without detachment), which vary in the amount of complexity allowed in the transition rules. We show that, in general, seeded Tile Automata systems require $\Theta(\log_{\frac{1}{4}} n)$ states. For Single-Transition systems, where only one state may change in a transition rule, we show a bound of $\Theta(\log_{\frac{1}{3}} n)$, and for deterministic systems, where each pair of states may only have one associated transition rule, a bound of $\Theta((\frac{\log n}{\log\log n})_{\frac{1}{2}})$.

See Appendix A for full paper.

3.2 Simulation of Multiple Stages in Single Bin Active Tile Self-Assembly

My Contributions I primarily worked on editing, wrote the abstract and made the first figure of this paper.

Abstract Two significant and often competing goals within the field of self-assembly are minimizing tile types and minimizing human-mediated experimental operations. The introduction of the Staged Assembly and Single Staged Assembly models, while successful in the former aim, necessitate an increase in mixing operations later. In this paper, we investigate building optimal lines as a standard benchmark shape and building primitive. We show that a restricted version of the 1D Staged Assembly Model can be simulated by the 1D Freezing Tile Automata model with the added benefits of the complete automation of stages and completion in a single bin while maintaining bin parallelism and a competitive number of states for lines, patterned lines, and context-free grammars.

See Appendix B for full paper.

CHAPTER IV

COVERT COMPUTATION IN THE ABSTRACT TILE-ASSEMBLY MODEL

My Contributions In this paper, my primary contribution to the results was in the "Exponential Assembly Covert Computer in 2D" section where I worked out the details of how to properly propagate the result and stop the construction from growing infinitely. I was given an initial construction for this section that was incomplete and then made figures 9, 10, and 11 from it. I also wrote the Theorem 2 proof and large portions of this section.

Additionally, I wrote much of the introduction and motivation section. I recolored and edited all of the 2D figures for figures 1-6 and I created figures 7 and 8.

I presented the paper at SAND 2023 in Pisa, Italy.

Abstract. There have been many advances in molecular computation that offer benefits such as targeted drug delivery, nanoscale mapping, and improved classification of nanoscale organisms. This power led to recent work exploring privacy in the computation, specifically, covert computation in self-assembling circuits. Here, we prove several important results related to the concept of a hidden computation in the most well-known model of self-assembly, the Abstract Tile-Assembly Model (aTAM). We show that in 2D, surprisingly, the model is capable of covert computation, but only with an exponential-sized assembly. We also show that the model is capable of covert computation with polynomial-sized assemblies with only one step in the third dimension (just-barely 3D). Finally, we investigate types of functions that can be covertly computed as members of P/Poly.

See Appendix C for the full paper.

CHAPTER V

CHEMICAL REACTION NETWORKS

5.1 Reachability in Restricted Chemical Reaction Networks

My Contributions In this paper, I wrote the overview of section 5 Void and Autogenesis Rules as well as the (3, 0) void rules / (0, 3) autogenesis rules are NP-Complete. I also created the first figure of the paper and designed the example.

Additionally, I assisted in editing the paper.

Abstract In this work, we fully characterize monotone reachability problems based on various restrictions such as the allowed rule size, the number of rules that may create a species (k-source), the number of rules that may consume a species (k-consuming), the volume, and whether the rules have an acyclic production order (*feed-forward*). We show PSPACE-completeness of reachability with only bimolecular reactions in two-source and two-consuming rules. This proves hardness of reachability in a restricted form of Population Protocols. This is accomplished using new techniques within the motion planning framework.

We give several important results for feed-forward CRNs, where rules are single-source or single-consuming. We show that reachability is solvable in polynomial time as long as the system does not contain special *void* or *autogenesis* rules. We then fully characterize all systems of this type and show that with void/autogenesis rules, or more than one source and one consuming, the problems become NP-complete. Finally, we show several interesting special cases of CRNs based on these restrictions or slight relaxations and note future significant open questions related to this taxonomy.

See Appendix D for full paper.

5.2 Computing Threshold Circuits with Void Reactions in Step Chemical Reaction Networks

My Contributions. In this paper I did the previous work research. Reading and evaluating two dozen publications as well as writing the introduction and general editing. In addition, I worked on the development of several of the results and fine-tuning, including fanout and AND gates in (3,0).

While unsuccessful, I also attempted fanout and assisted in the later fanout constructions in (2,0) and (3,0).

Abstract We introduce a new model of *step* Chemical Reaction Networks (step CRNs), motivated by the step-wise addition of materials in standard lab procedures. Step CRNs have ordered reactants that transform into products via reaction rules over a series of steps. We study an important subset of weak reaction rules, *void* rules, in which chemical species may only be deleted but never changed. We demonstrate the capabilities of these simple limited systems to simulate threshold circuits and compute functions using various configurations of rule sizes and step constructions, and prove that without steps, void rules are incapable of these computations, which further motivates the step model. Additionally, we prove the coNP-completeness of verifying if a given step CRN computes a function, holding even for $O(1)$ step systems.

See Appendix E for full paper.

CHAPTER VI

SURFACE CHEMICAL REACTION NETWORKS

For both of these papers I attended the associated meetings and participated in discussion.

6.1 Complexity of Reconfiguration in Surface Chemical Reaction Networks

In this paper I made the example figures for the example as well as created the figures for the burnout section, working out many of the details. I made figures 1, 2, 7, 8, 9, and 10. In addition, I edited the paper.

Abstract We analyze the computational complexity of basic reconfiguration problems for the recently introduced surface Chemical Reaction Networks (sCRNs), where ordered pairs of adjacent species nondeterministically transform into a different ordered pair of species according to a predefined set of allowed transition rules (chemical reactions). In particular, two questions that are fundamental to the simulation of sCRNs are whether a given configuration of molecules can ever transform into another given configuration, and whether a given cell can ever contain a given species, given a set of transition rules. We show that these problems can be solved in polynomial time, are NP-complete, or are PSPACE-complete in a variety of different settings, including when adjacent species just swap instead of arbitrary transformation (swap sCRNs), and when cells can change species a limited number of times (k-burnout). Most problems turn out to be at least NP-hard except with very few distinct species (2 or 3).

6.2 Reconfiguration of Linear Surface Chemical Reaction Networks with Bounded State Change

In this paper I edited, had discussions with Ryan over naming and refining his algorithm and I created all 3 figures in the paper.

Abstract We present results on the complexity of reconfiguration of surface Chemical Reaction Networks (sCRNs) in a model where surface vertices can change state a bounded number of times based on a given burnout parameter k. We primarily focus on linear $1 \times n$ surfaces. Without a burnout bound, or even with an exponentially high bound on burnout, reconfiguration on linear surfaces is known to be PSPACE-complete. In contrast, we show that the problem becomes NP-complete when the burnout k is polynomially bounded in n. For smaller $k = O(1)$, we show the problem is polynomial-time solvable, and in the special case of $k = 1$ burnout, reconfiguration can
be solved in linear $O(n + |R|)$ time, where $|R|$ denotes the number of system rules. We additionally explore some extensions of this problem to more general graphs, including a fixed-parameter tractable algorithm in the height m of an $m \times n$ rectangle in 1-burnout, a polynomial-time solution for 1-burnout in general graphs if reactions are non-catalytic, and an NP-complete result for 1-burnout in general graphs.

APPENDIX A

APPENDIX A

BUILDING SQUARES WITH OPTIMAL STATE COMPLEXITY IN RESTRICTED ACTIVE SELF-ASSEMBLY

Building Squares with Optimal State Complexity in Restricted Active Self-Assembly

Robert M. Alaniz
Department of Computer Science, University of Texas Rio Grande Valley

David Caballero
Department of Computer Science, University of Texas Rio Grande Valley

Sonya C. Cirlos
Department of Computer Science, University of Texas Rio Grande Valley

Timothy Gomez
Department of Computer Science, University of Texas Rio Grande Valley

Elise Grizzell
Department of Computer Science, University of Texas Rio Grande Valley

Andrew Rodriguez
Department of Computer Science, University of Texas Rio Grande Valley

Robert Schweller
Department of Computer Science, University of Texas Rio Grande Valley

Armando Tenorio
Department of Computer Science, University of Texas Rio Grande Valley

Tim Wylie
Department of Computer Science, University of Texas Rio Grande Valley

Abstract

Tile Automata is a recently defined model of self-assembly that borrows many concepts from cellular automata to create active self-assembling systems where changes may be occurring within an assembly without requiring attachment. This model has been shown to be powerful, but many fundamental questions have yet to be explored. Here, we study the state complexity of assembling $n \times n$ squares in seeded Tile Automata systems where growth starts from a seed and tiles may attach one at a time, similar to the abstract Tile Assembly Model. We provide optimal bounds for three classes of seeded Tile Automata systems (all without detachment), which vary in the amount of complexity allowed in the transition rules. We show that, in general, seeded Tile Automata systems require $\Theta(\log^{\frac{1}{4}} n)$ states. For Single-Transition systems, where only one state may change in a transition rule, we show a bound of $\Theta(\log^{\frac{1}{3}} n)$, and for deterministic systems, where each pair of states may only have one associated transition rule, a bound of $\Theta((\frac{\log n}{\log \log n})^{\frac{1}{2}})$.

2012 ACM Subject Classification Theory of computation → Self-organization; Theory of computation → Computational geometry; Applied computing → Computational biology

Keywords and phrases Active Self-Assembly, State Complexity, Tile Automata

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding This research was supported in part by National Science Foundation Grant CCF-1817602.

Acknowledgements We would like to thank the reviewers for their comments, specifically for pointing us toward relevant Cellular Automata Literature.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:18

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

1 Introduction

Self-assembly is the process by which simple elements in a system organize themselves into more complex structures based on a set of rules that govern their interactions. These types of systems occur naturally and can be easily constructed artificially to offer many advantages when building micro or nanoscale objects. One abstraction of these systems that has yielded interesting results is Tile Self-Assembly.

In the abstract Tile Assembly Model (aTAM) [35], the elements of a system are represented using labeled unit squares called tiles. A system is initialized with a seed (a tile or assembly) that grows as other single tiles attach until there are no more valid attachments. The behavior of a system can then be programmed, using the interactions of tiles, and is known to be capable of Turing Computation [35], is Intrinsically Universal [14], and can assemble general scaled shapes [33]. However, many of these results utilize a concept called *cooperative binding*, where a tile must attach to an assembly using the interaction from two other tiles. Unlike with cooperative binding, the non-cooperative aTAM is not Intrinsically Universal [25, 27] and more recent work has shown that it is not capable of Turing Computation [26]. Many extensions of this model increase the power of non-cooperative systems [4, 16, 18, 22, 23, 30].

One recent model of self-assembly is Tile Automata [8]. This model marries the concept of state changes from Cellular Automata [19, 28, 37] and the assembly process from the 2-Handed Assembly model (2HAM) [6]. Previous work [3, 7, 8] has explored Tile Automata as a unifying model for comparing the relative powers of the many different Tile Assembly models. The complexity of verifying the behavior of systems along with their computational power was studied in [5]. Many of these works impose additional experimentally motivated limitations on the Tile Automata model that help connect the model and its capabilities to potential molecular implementations, such as using DNA assemblies with sensors to assemble larger structures [21], building spacial localized circuits on DNA origami [10], or DNA walkers that sort cargo [34].

In this paper, we explore the aTAM generalized with state changes; we define our producible assemblies as what can be grown by attaching tiles one at a time to a seed tile or performing transition rules, which we refer to as seeded Tile Automata. This is a bounded version of Asynchronous Cellular Automata [15]. Reachability problems, which are similar to verification problems in self-assembly, have been studied with many completeness results [13]. Further, the freezing property used in this and previous work also exists in Cellular Automata [20, 29].[1] Freezing is defined differently in Cellular Automata by requiring that there exists an ordering to the states.

While Tile Automata has many possible metrics, we focus on the number of states needed to uniquely assemble $n \times n$ squares at the smallest constant temperature, $\tau = 1$. We achieve optimal bounds in three versions of the model with varying restrictions on the transition rules. Our results, along with previous results in the aTAM, are outlined in Table 1.

1.1 Previous Work

In the aTAM, the number of tile types needed, for nearly all n, to construct an $n \times n$ square is $\Theta(\frac{\log n}{\log n \log n})$ [1, 31] with temperature $\tau = 2$ (row 2 of Table 1). The same lower bounds hold for $\tau = 1$ (row 1 of Table 1). The run time of this system was also shown to be optimal $\Theta(n)$ [1]. Other bounds for building rectangles were shown in [2]. While no tighter bounds[2]

[1] We would like to thank a reviewer for bringing these works to our attention.
[2] Other than trivial $\mathcal{O}(n)$ bounds.

Model	τ	$n \times n$ Squares		
		Lower	Upper	Theorem
aTAM	1	$\Omega(\frac{\log n}{\log \log n})$	$\mathcal{O}(n)$	[31], [1]
aTAM	2	$\Theta(\frac{\log n}{\log \log n})$		[31], [1]
Flexible Glue aTAM	2	$\Theta(\log^{\frac{1}{2}} n)$		[2]
Seeded TA Det.	1	$\Theta((\frac{\log n}{\log \log n})^{\frac{1}{2}})$		Thm. 2, 12
Seeded TA ST	1	$\Theta(\log^{\frac{1}{3}} n)$		Thm. 4, 12
Seeded TA	1	$\Theta(\log^{\frac{1}{4}} n)$		Thm. 3, 12

Table 1 Bounds on the number of states for $n \times n$ squares in the Abstract Tile Assembly model, with and without cooperative binding, and the seeded Tile Automata model with our transition rules. ST stands for Single-Transition.

have been shown for $n \times n$ squares at $\tau = 1$ in the aTAM, generalizations to the model that allow (just-barely) 3D growth have shown an upper bound of $\mathcal{O}(\log n)$ for tile types needed [11]. Recent work in [17] shows improved upper and lower bounds on building thin rectangles in the case of $\tau = 1$ and in (just-barely) 3D.

Other models of self-assembly have also been shown to have a smaller tile complexity, such as the staged assembly model [9, 12] and temperature programming [24]. Investigation into different active self-assembly models have also explored the run time of systems [32, 36].

1.2 Our Contributions

In this work, we explore building an important benchmark shape, squares, in non-cooperative seeded Tile Automata. We also consider only affinity-strengthening transition rules that remove the ability for an assembly to break apart. Our results are shown in Table 1.

We start in Section 3 by proving lower bounds for building $n \times n$ squares based on three different transition rule restrictions. The first is nondeterministic or general seeded Tile Automata, where there are no restrictions and a pair of states may have multiple transition rules. The second is Single-Transition rules where only one tile may change states in a transition rule, but we still allow multiple rules for each pair of states. The last restriction, Deterministic, is the most restrictive where each pair of states may only have one transition rule (for each direction).

In Section 4, we use Transition Rules to optimally encode strings in the various versions of the model. We use these encodings as gadgets to seed the future constructions. We show how to build optimal state complexity rectangles in Section 5, and finally optimal state complexity squares in Section 6. Future work is discussed in Section 7.

AutoTile. To test our constructions, we developed AutoTile, a seeded Tile Automata simulator. Each system discussed in the paper is currently available for simulation. AutoTile is available at `https://github.com/asarg/AutoTile`.

2 Definitions

The Tile Automata model differs quite a bit from normal self-assembly models since a *tile* may change *state*, which draws inspiration from Cellular Automata. Thus, there are two aspects of a TA system being: the self-assembling that may occur with tiles in a state and the changes to the states once they have attached to each other. To address these aspects, we define the building blocks and interactions, and then the definitions around the model

and what it may assemble or output. Finally, since we are looking at a limited TA system, we also define specific limitations and variations of the model. For reference, an example system is shown in Figure 1.

2.1 Building Blocks

The basic definitions of all self-assembly models include the concepts of tiles, some method of attachment, and the concept of aggregation into larger assemblies. The Cellular Automata aspect also brings in the concept of transitions.

Tiles. Let Σ be a set of *states* or symbols. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^2$ and has a state of $\sigma \in \Sigma$.

Affinity Function. An *affinity function* Π over a set of states Σ takes an ordered pair of states $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$, and outputs an element of $\mathbb{Z}^0$. The orientation d is the relative position to each other with $\vdash$ meaning horizontal and $\perp$ meaning vertical, with the σ_1 being the west or north state respectively. We refer to the output as the *Affinity Strength* between these two states.

Transition Rules. A *Transition Rule* consists of two ordered pairs of states $(\sigma_1, \sigma_2), (\sigma_3, \sigma_4)$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$. This denotes that if the states (σ_1, σ_2) are next to each other in orientation d (σ_1 as the west/north state) they may be replaced by the states (σ_3, σ_4).

Assembly. An assembly A is a set of tiles with states in Σ such that for every pair of tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2)$, $p_1 \neq p_2$. Informally, each position contains at most one tile. Further, we say assemblies are equal in regards to translation. Two assemblies A_1 and A_2 are equal if there exists a vector $\vec{v}$ such that $A_1 = A_2 + \vec{v}$.

Let $B_G(A)$ be the bond graph formed by taking a node for each tile in A and adding an edge between neighboring tiles $t_1 = (\sigma_1, p_1)$ and $t_2 = (\sigma_2, p_2)$ with a weight equal to $\Pi(\sigma_1, \sigma_2)$. We say an assembly A is τ-stable for some $\tau \in \mathbb{Z}^0$ if the minimum cut through $B_G(A)$ is greater than or equal to τ.

2.2 The Tile Automata Model

Here, we define and investigate the *Seeded Tile Automata* model, which differs by only allowing single tile attachments to a growing seed similar to the aTAM.

Seeded Tile Automata. A Seeded Tile Automata system is a 6-tuple $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ where Σ is a set of states, $\Lambda \subseteq \Sigma$ a set of *initial states*, Π is an *affinity function*, Δ is a set of *transition rules*, s is a stable assembly called the *seed* assembly, and τ is the *temperature* (or threshold). Our results use the most restrictive version of this model where s is a single tile.

Attachment Step. A tile $t = (\sigma, p)$ may attach to an assembly A at temperature τ to build an assembly $A' = A \bigcup t$ if A' is τ-stable and $\sigma \in \Lambda$. We denote this as $A \rightarrow_{\Lambda,\tau} A'$.

Transition Step. An assembly A is transitionable to an assembly A' if there exists two neighboring tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2) \in A$ (where t_1 is the west or north tile) such that there exists a transition rule in Δ with the first pair being (σ_1, σ_2) and $A' = (A \setminus \{t_1, t_2\}) \bigcup \{t_3 = (\sigma_3, p_1), t_4 = (\sigma_4, p_2)\}$. We denote this as $A \rightarrow_{\Delta} A'$.

Producibles. We refer to both attachment steps and transition steps as production steps, we define $A \rightarrow_* A'$ as the transitive closure of $A \rightarrow_{\Lambda,\tau} A'$ and $A \rightarrow_{\Delta} A'$. The set of *producible assemblies* for a Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ is written as $PROD(\Gamma)$. We define $PROD(\Gamma)$ recursively as follows,

- $s \in PROD(\Gamma)$
- $A' \in PROD(\Gamma)$ if $\exists A \in PROD(\Gamma)$ such that $A \rightarrow_{\Lambda,\tau} A'$.

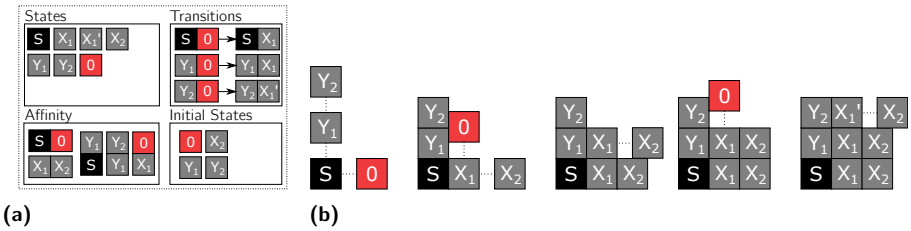


Figure 1 (a) Example of a Tile Automata system, it should be noted that $\tau = 1$ and state S is our seed. (b) A walkthrough of our example Tile Automata system building the 3×3 square it uniquely produces. We use dotted lines throughout our paper to represent tiles attaching to one another.

$A' \in PROD(\Gamma)$ if $\exists A \in PROD(\Gamma)$ such that $A \rightarrow_{\Delta} A'$.

Terminal Assemblies. The set of terminal assemblies for a Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, \tau\}$ is written as $TERM(\Gamma)$. This is the set of assemblies that cannot grow or transition any further. Formally, an assembly $A \in TERM(\Gamma)$ if $A \in PROD(\Gamma)$ and there does not exists any assembly $A' \in PROD(\Gamma)$ such that $A \rightarrow_{\Lambda,\tau} A'$ or $A \rightarrow_{\Delta} A'$. A Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ *uniquely* assembles an assembly A if $A \in TERM(\Gamma)$, and for all $A' \in PROD(\Gamma), A' \rightarrow_{*} A$.

2.3 Limited Model Reference

We explore an extremely limited version of seeded TA that is affinity-strengthening, freezing, and may be a single-transition system. We investigate both deterministic and nondeterministic versions of this model.

Affinity Strengthening. We only consider transitions rules that are affinity strengthening, meaning for each transition rule $((\sigma_1, \sigma_2), (\sigma_3, \sigma_4), d)$, the bond between (σ_3, σ_4) must be at least the strength of (σ_1, σ_2). Formally, $\Pi(\sigma_3, \sigma_4, d) \geq \Pi(\sigma_1, \sigma_2, d)$. This ensures that transitions may not induce cuts in the bond graph.

In the case of non-cooperative systems ($\tau = 1$), the affinity strength between states is always 1 so we may refer to the affinity function as an affinity set Λ_s, where each affinity is a 3-pule (σ_1, σ_2, d).

Freezing. Freezing systems were introduced with Tile Automata. A freezing system simply means that a tile may transition to any state only once. Thus, if a tile is in state A and transitions to another state, it is not allowed to ever transition back to A.

Deterministic vs. Nondeterministic. For clarification, a deterministic system in TA has only one possible production step at a time, whether that be an attachment or a state transition. A nondeterministic system may have many possible production steps and any choice may be taken.

Single-Transition System. We restrict our TA system to only use single-transition rules. This means that for each transition rule one of the states may change, but not both. It should be noted that we still allow Nondeterminism in this system.

3 State Space Lower Bounds

Let $p(n)$ be a function from the positive integers to the set $\{0, 1\}$, informally termed a *proposition*, where 0 denotes the proposition being false and 1 denotes the proposition being true. We say a proposition $p(n)$ holds for *almost all* n if $\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} p(i) = 1$.

▶ **Lemma 1.** *Let U be a set of TA systems, b be a one-to-one function mapping each element of U to a string of bits, and ϵ a real number from $0 < \epsilon < 1$. Then for almost all integers n, any TA system $\Gamma \in U$ that uniquely assembles either an $n \times n$ square or a $1 \times n$ line has a bit-string of length $|b(\Gamma)| \geq (1-\epsilon)\log n$.*

Proof. For a given $i \geq 1$, let $M_i \in U$ denote the TA system in U with the minimum value $|b(M_i)|$ over all systems in U that uniquely assembly an $i \times i$ square or $1 \times i$ line, and let M_i be undefined if no such system in U builds such a shape. Let $p(i)$ be the proposition that $|b(M_i)| \geq (1-\epsilon)\log i$. We show that $\lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} p(i) = 1$. Let $R_n = \{M_i | 1 \leq i \leq n, |b(M_i)| < (1-\epsilon)\log n\}$. Note that $n - |R_n| \leq \sum_{i=1}^{n} p(i)$. By the pigeon-hole principle, $|R_n| \leq 2^{(1-\epsilon)\log n} = n^{(1-\epsilon)}$. Therefore,

$$\lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} p(i) \geq \lim_{n\to\infty} \frac{1}{n}(n - |R_n|) \geq \lim_{n\to\infty} \frac{1}{n}(n - n^{1-\epsilon}) = 1.$$

◀

▶ **Theorem 2** (Deterministic TA). *For almost all n, any Deterministic Tile Automata system that uniquely assembles either a $1 \times n$ line or an $n \times n$ square contains $\Omega(\frac{\log n}{\log\log n})^{\frac{1}{2}}$ states.*

Proof. We can create a one-to-one mapping $b(\Gamma)$ from any deterministic TA system to bit-strings in the following way. Let S denote the set of states in a given system. We encode the state set in $\mathcal{O}(\log|S|)$ bits, we encode the affinity function in a $|S| \times |S|$ table of strengths in $\mathcal{O}(|S|^2)$ bits (assuming a constant bound on bonding thresholds), and we encode the rules of the system in an $|S| \times |S|$ table mapping pairs of rules to their unique new pair of rules using $\mathcal{O}(|S|^2 \log|S|)$ bits, for a total of $\mathcal{O}(|S|^2 \log|S|)$ bits to encode any $|S|$ state system.

Let Γ_n denote the smallest state system that uniquely assembles an $n \times n$ square (or similarly a $1 \times n$ line), and let S_n denote the state set. By Lemma 1, $|b(\Gamma_n)| \geq (1-\epsilon)\log n$ for almost all n, and so $|S_n|^2 \log|S_n| = \Omega(\log n)$ for almost all n. We know that $|S_n| = \mathcal{O}(\log n)$, so for some constant c, $|S_n| \geq c(\frac{\log n}{\log\log n})^{\frac{1}{2}}$ for almost all n. ◀

▶ **Theorem 3** (Nondeterministic TA). *For almost all n, any Tile Automata system (in particular any Nondeterministic system) that uniquely assembles either a $1 \times n$ line or an $n \times n$ square contains $\Omega(\log^{\frac{1}{4}} n)$ states.*

▶ **Theorem 4** (Single-Transition TA). *For almost all n, any Single-Transition Tile Automata system that uniquely assembles either a $1 \times n$ line or an $n \times n$ square contains $\Omega(\log^{\frac{1}{3}} n)$ states.*

4 String Unpacking

A key tool in our constructions is the ability to build strings efficiently. We do so by encoding the string in the transition rules.

▶ **Definition 5** (String Representation). *An assembly A over states Σ represent a string S over a set of symbols U if there exists a mapping from the elements of U to the elements of Σ and a $1 \times |S|$ (or $|S| \times 1$) subassembly $A' \sqsubset A$, such that the state of the i^{th} tile of A' maps to the i^{th} symbol of S for all $0 \leq i \leq |S|$.*

4.1 Deterministic Transitions

We start by showing how to encode a binary string of length n in a set of (freezing) transition rules that take place on a $2 \times (n+2)$ rectangle that will print the string on its right side. We extend this construction to work for an arbitrary base string.

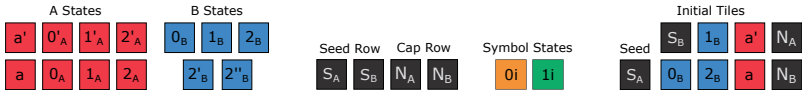


Figure 2 States to build a length-9 string in deterministic Tile Automata.

4.1.1 Overview

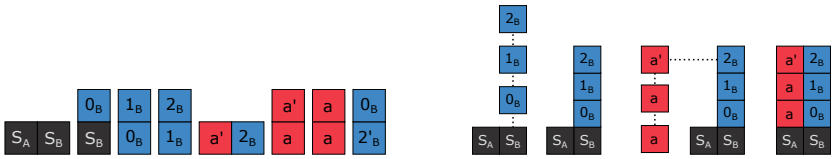
Consider a system that builds a length n string. First, we create a rectangle of index states that is two wide as seen on the left side of Figure 5c. Each row has a unique pair of index states so each bit of the string is uniquely indexed. We divide the index states into two groups based on which column they are in, and which "digit" they represent. Let $r = \lceil n^{\frac{1}{2}} \rceil$. Starting with index states A_0 and B_0, we build a counter pattern with base r. We use $\mathcal{O}(n^{\frac{1}{2}})$ states shown in Figure 2 to build this pattern. We encode each bit of the string in a transition rule between the two states that index that bit. A table with these transition rules can be seen in Figure 5b.

The pattern is built in r sections of size $2 \times r$ with the first section growing off of the seed. The tile in state S_A is the seed. There is also a state S_B that has affinity for the right side of S_A. The building process is defined in the following steps for each section.

1. The states $S_B, 0_B, 1_B, \ldots, (r-1_B)$ grow off of S_B, forming the right column of the section. The last B state allows for a' to attach on its west side. a tiles attach below a' and below itself. This places a states in a row south toward the state S_A, depicted in Figure 3b.
2. Once a section is built, the states begin to follow their transition rules shown in Figure 4a. The a state transitions with seed state S_A to begin indexing the A column by changing state a to state 0_A. For $1 \leq y \leq n-2$, state a vertically transitions with the other y'_A states, incrementing the index by changing from state a to state $(y+1)_A$.
3. This new index state z_A propagates up by transitioning the a tiles to the state z_A as well. Once the z_A state reaches a' at the top of the column, it transitions a' to the state z'_A. Figure 4b presents this process of indexing the A column.
4. If $z < n-1$, there is a horizontal transition rule from states $(z'_A, n-1_B)$ to states $(z'_A, n-1'_B)$. The state 0_B attaches to the north of $n-1_B$ and starts the next section. If $z = n$, there does not exist a transition.
5. This creates an assembly with a unique state pair in each row as seen in the first column of Figure 5c.

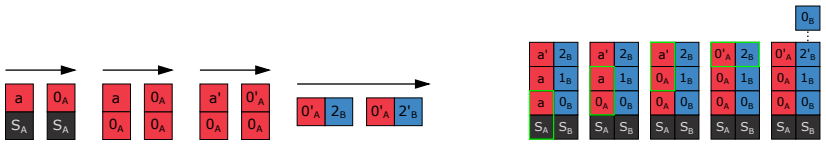
4.1.2 States

An example system with the states required to print a length-9 string are shown in Figure 2. The first states build the seed row of the assembly. The seed tile has the state S_A with initial tiles in state S_B. The index states are divided into two groups. The first set of index states, which we call the A index states, are used to build the left column. For each i, $0 \leq i < r$, we have the states i_A and i'_A. There are two states a and a', which exist as initial tiles and act as "blank" states that can transition to the other A states. The second set of index states are the B states. Again, we have r B states numbered from 0 to $r-1$, however, we do not have a prime for each state. Instead, there are two states $r-1'_B$ and $r-1''_B$, that are used to control the growth of the next column and the printing of the strings. The last states are the symbol states 0_S and 1_S, the states that represent the string.



(a) Affinity Rules for Initial Tiles **(b)** Process of Building a section

Figure 3 (a) Affinity rules to build each section. We only show affinity rules that are actually used in our system for initial tiles to attach, while our system would have more rules in order to meet the affinity strengthening restriction. (b) The B column attaches above the state S_B as shown by the dotted lines. The a' attaches to the left of 2_B and the other a states may attach below it until they reach S_A.



(a) Transition Rules to Index the first section **(b)** Process of Indexing A column

Figure 4 (a) The first transition rule used is takes place between the seed S_A and the a state changing to 0_A. The state 0_A changes the states north of it to 0_A or $0'_A$. Finally, the state $0'_A$ transitions with 2_B (b) Once the a states reach the seed row they transition with the state S_A to go to 0_A. This state propagates upward to the top of the section.

4.1.3 Affinity Rules/Placing Section

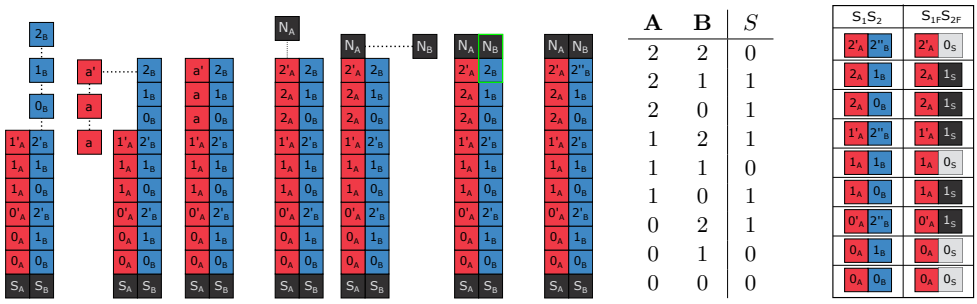
Here, we describe the affinity rules for building the first section. We later describe how this is generalized to the other $r-1$ sections. We walk through this process in Figure 3b. To begin, the B states attach in sequence above the tile S_B in the seed row. Assuming $r^2 = n$, n is a perfect square, the first state to attach is 0_B. 1_B attaches above this tile and so on. The last B state $r-1_B$ does not have affinity with 0_B, so the column stops growing. However, the state a' has affinity on the left of $r-1_B$ and can attach. a has affinity for the south side of a', so it attaches below. The a state also has a vertical affinity with itself. This grows the A column southward toward the seed row.

If n is not a perfect square, we start the index state pattern at a different value. We do so by finding the value $q = r^2 - n$. In general, the state i_B attaches above S_B for $i = q\%r$.

4.1.4 Transition Rules/Indexing A column

Once the A column is complete and the last A state is placed above the seed, it transitions with S_A to 0_A (assuming $r^2 = n$). A has a vertical transition rule with i_A $(0 \leq i < r)$ changing the state A to state i_A. This can be seen in Figure 4a, where the 0_A state is propagated upward to the A' state. The A' state also transitions when 0_A is below it, going from state A' to state $0'_A$. If n is not a perfect square, then A transitions to i_A for $i = \lfloor q/r \rfloor$.

Once the transition rules have finished indexing the A column if $i < r-1$, the last state i'_A transitions with $r-1_B$ changing the state $r-1_B$ to $r-1'_B$. This transition can be seen in Figure 4b. The new state $r-1'_B$ has an affinity rule allowing 0_B to attach above it allowing the next section to be built. When the state A is above a state j'_A, $0 \leq j < r-1$, it transitions with that state changing from state A to $j+1_A$, which increments the A index.



A	B	S
2	2	0
2	1	1
2	0	1
1	2	1
1	1	0
1	0	1
0	2	1
0	1	0
0	0	0

S_1S_2	$S_{1F}S_{2F}$
$2'_A$ $2''_B$	$2'_A$ 0_S
2_A 1_B	2_A 1_S
2_A 0_B	2_A 1_S
$1'_A$ $2''_B$	$1'_A$ 1_S
1_A 1_B	1_A 0_S
1_A 0_B	1_A 1_S
$0'_A$ $2''_B$	$0'_A$ 1_S
0_A 1_B	0_A 0_S
0_A 0_B	0_A 0_S

(a) Attaching Cap Row **(b)** Encoding of S **(c)** Transition Rules

Figure 5 (a) Once the last section finishes building the state N_A attaches above $2'_A$. N_B then attaches to the assembly and transitions with 2_B changing it directly to $2''_B$ so the string may begin printing. (b) A table indexing the string $S = 011101100$ using two columns and base $|S|^{\frac{1}{2}}$. (c) Transition Rules to print S. We build an assembly where each row has a unique pair of index states in ascending order.

4.1.5 Look up

After creating a $2 \times (n+2)$ rectangle, we can encode a length n string S into the transitions rules. Note that each row of our assembly consists of a unique pair of index states, which we call a *bit gadget*. Each bit gadget will *look up* a specific bit of our string and transition the B tile to a state representing the value of that bit.

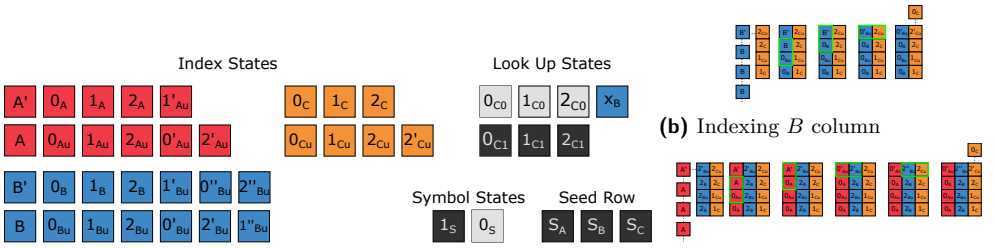
Figure 5b shows how to encode a string S in a table with two columns using r digits to index each bit. From this encoding, we create our transition rules. Consider the k^{th} bit of S (where the 0^{th} bit is the least significant bit) for $k = ir + j$. Add transition rules between the states i_A and j_B, changing the state j_B to either 0_S or 1_S based on the k^{th} bit of S. This transition rule is slightly different for the northmost row of each section as the state in the A column is i'_A. Also, we do not want the state in the B column, $r-1_B$, to prematurely transition to a symbol state. Thus, we have the two states $r-1'_B$ and $r-1''_B$. As mentioned, once the A column finishes indexing, it changes the state $r-1_B$ to state $r-1'_B$, allowing for 0_B to attach above it, which starts the next column. Once the state 0_B (or a symbol state) is above $r-1'_B$, there are no longer any possible undesired attachments, so the state transitions to $r-1''_B$, which has the transition to the symbol state.

The last section has a slightly different process as $r-1_B$ state will never have a 0_B attach above it, so we have a different transition rule. This alternate process is shown in Figure 5a. The state $r-1'_A$ has a vertical affinity with the cap state N_A. This state allows N_B to attach on its right side. This state transitions with $r-1_B$ below it, changing it directly to $r-1''_B$, allowing the symbol state to print.

▶ **Theorem 6.** *For any binary string s with length $n > 0$, there exists a freezing tile automata system Γ_s with deterministic transition rules, that uniquely assembles an $2 \times (n+2)$ assembly A_S that represents S with $\mathcal{O}(n^{\frac{1}{2}})$ states.*

4.1.6 Arbitrary Base

In order to optimally build rectangles, we first print arbitrary base strings. Here, we show how to generalize Theorem 6 to print base-b strings.



(a) States space for when $|S| = 27$

(b) Indexing B column

(c) Indexing the A column

Figure 6 (a) States needed to construct a length 27 string where $r = 3$. (b) The index 0 propagates upward by transitioning the tiles in the column to 0_B and 0_{Bu} and transitions a' to $0'_{Bu}$. The state $0'_{Bu}$ transitions with the state 2_{Cu}, changing the state 2_{Cu} to $2'_{Cu}$, which has affinity with 0_C to build the next section. These rules also exist for the index 1. (c) When the index state 2_B reaches the top of the section, it transitions b' to $2'_{Bu}$. This state does not transition with the C column and instead has affinity with the state a', which builds the A column downward. The index propagates up the A column in the same way as the B column. When the index state 0_A reaches the top of the section, it transitions the state $2'_B$ to $2''_B$. This state transitions with 2_{Cu} changing it to $2'_{Cu}$ allowing the column to grow.

▶ **Corollary 7.** *For any base-b string S with length $n > 0$, there exists a freezing tile automata system Γ with deterministic transition rules, that uniquely assembles an $(n+2) \times 2$ assembly which represents S with $\mathcal{O}(n^{\frac{1}{2}} + b)$ states.*

4.2 Nondeterministic Single-Transition Systems

For the case of Single-Transition systems, we use the same method from above but instead building bit gadgets that are of size 3×2. Expanding to 3 columns allows for a third index digit to be used giving us an upper bound of $\mathcal{O}(n^{\frac{1}{3}})$. The second row will be used for error checking which we will describe later in the section. This system utilizes Nondeterministic transitions, (two states may have multiple rules with the same orientation) and is non-freezing (a tile may repeat states). This system also contains cycles in its production graph, this implies the system may run indefinitely. We conjecture this system has a polynomial run time. Here, let $r = \lceil n^{\frac{1}{3}} \rceil$.

4.2.1 Index States and Look Up States

We generalize the method from above to start from a C column. The B column now behaves as the second index of the pattern and is built using B' and B as the A column was in the previous system. Once the B reaches the seed row, it is indexed with its starting value. This construction also requires bit gadgets of height 2, so we will use index states i_A, i_B, i_C and north index states i_{Au}, i_{Bu}, i_{Cu} for $0 \leq i < r$. This allows us to separate the two functions of the bit gadget into each row. The north row has transition rules to control the building of each section. The bottom row has transition rules that encode the represented bit.

In addition to the index states, we use $2r$ look up states, 0_{Ci} and 1_{Ci} for $0 \leq i < r$. These states are used as intermediate states during the look up. The first number (0 or 1) represents the value of the retrieved bit, while the second number represents the C index of the bit. The A and B indices of the bit will be represented by the other states in the transition rule.

In the same way as the previous construction, we build the rightmost column first. We

include the C index states as initial states and allow 0_C to attach above S_C. We include affinity rules to build the column northwards as follows starting with the southmost state $0_C, 0_{Cu}, 1_C, 1_{Cu}, \ldots, r-2_{Cu}, r-1_C, r-1_{Cu}$.

To build the other columns, the state b' can attach on the left of $r-1_{Cu}$. The state b is an initial state and attaches below b' and itself to grow downward toward the seed row. The state b transitions with the seed row as in the previous construction to start the column. However, we alternate between C states and Cu states. The state b above i_C transitions b to i_{Cu}. If b is above i_{Cu} it transitions to i_C. The state b' above state i_B transitions to i'_{Bu}. If $i < r-1$, the state i'_B and $r-1_{Cu}$ transition horizontally changing $r-1'_{Cu}$, which allows 0_C to attach above it to repeat the process. This is shown in Figure 6b.

The state a' attaches on the left of $r-1_{Cu}$. The A column is indexed just like the B column. For $0 \leq i < r-1$, the state i'_{Au} and $r-1'_{Bu}$ change the state $r-1'_{Bu}$ to $r-1''_{Bu}$. This state transitions with $r-1_{Cu}$, changing it to $r-1'_{Cu}$. See Figure 6c.

4.2.2 Bit Gadget Look Up

The bottom row of each bit gadget has a unique sequence of states, again we use these index states to represent the bit indexed by the digits of the states. However, since we can only transition between two tiles at a time, we must read all three states in multiple steps. These steps are outlined in Figure 7a. The first transition takes place between the states i_A and j_B. We refer to these transition rules as look up rules. We have r look up rules between these states for $0 \leq k < r$ of these states that changes the state j_B to that state k_{C0} if the bit indexed by i, j, and k is 0 or the state k_{C1} if the bit is 1.

Our bit gadget has Nondeterministically looked up each bit indexed by it's A and B states, Now, we must compare the bit we just retrieved to the C index via the state in the C column. The states k_{C0} and k_C transition changing the state k_C to the $0i$ state only when they represent the same k. The same is true for the state k_{C1} except C_k transitions to $1i$.

If they both represent different k, then the state k_C goes to the state B_x. This is the error checking of our system. The B_x states transitions with the north state j_{Bu} above it transitioning B_x to j_B once again. This takes the bit gadget back to it's starting configuration and another look up can occur.

▶ **Theorem 8.** *For any binary string S with length $n > 0$, there exists a Single-Transition tile automata system Γ, that uniquely assembles an $(2n+2) \times 3$ assembly which represents S with $\mathcal{O}(n^{\frac{1}{3}})$ states.*

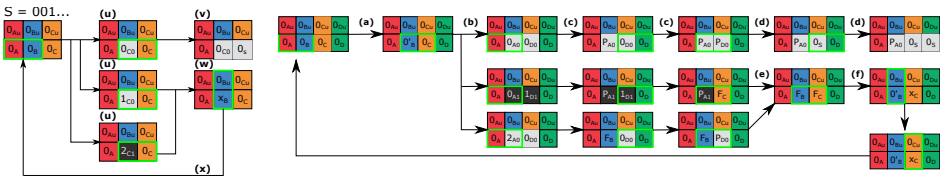
4.3 General Nondeterministic Transitions

Using a similar method to the previous sections, we build length n strings using $\mathcal{O}(n^{\frac{1}{4}})$ states. We start by building a pattern of index states with bit gadgets of height 2 and width 4.

4.3.1 Overview

Here, let $r = \lceil n^{\frac{1}{4}} \rceil$. We build index states in the same way as the Single-Transition system but instead starting from the D column. We have 4 sets of index states, A, B, C, D. The same methods are used to control when the next section builds by transitioning the state $r-1_D$ to $r-1'_D$ when the current section is finished building.

We use a similar look up method as the previous construction where we Nondeterministically retrieve a bit. However, since we are not restricting our rules to be a Single-Transition system, we may retrieve 2 indices in a single step. We include 2 sets of $\mathcal{O}(r)$ look up



(a) ST Bit Gadget look up **(b)** Nondeterminstic Bit Gadget look up

Figure 7 (a.u) For a string S, where the first 3 bits are 001, the states 0_A and 0_B have $|S|^{\frac{1}{3}}$ transition rules changing the state 0_B to a state representing one of the first $|S|^{\frac{1}{3}}$ bits. The state is i_{C0} if the i^{th} bit is 0 or i_{C1} if the i^{th} bit is 1 (a.v) The state 0_{C0} and the state 0_C both represent the same C index so the 0_C state transition to the 0_s. (a.w) For all states not matching the index of 0_C, they transition to x_B, which can be seen as a blank B state. (a.x) The state 0_{Bu} transitions with the state x_B changing to 0_B resetting the bit gadget. (b.a) Once the state A_0 appears in the bit gadget it transitions with 0_B changing 0_B to $0'_B$. (b.b) The states $0'_B$ and 0_C Nondeterminstically look up bits with matching B and C indices. The state $0'_B$ transitions to look up state representing the bit retrieved and the bit's A index. The state 0_C transitions to a look up state representing the D index of the retrieved bit. (b.c) The look-up states transition with the states 0_A and 0_D, respectively. As with the Single-Transition construction these may pass or fail. (b.d) When both tests pass, they transition the D look up state to a symbol state that propagates out. (b.e) If a test fails, the states both go to blank states. (b.f) The blank states then reset using the states to their north.

states, the A look up states and the D look up states. We also include Pass and Fail states $F_B, F_C, P_{A0}, P_{D0}, P_{A1}, P_{D1}$ along with the blank states B_x and C_x. We utilize the same method to build the north and south row.

Let $S(\alpha, \beta, \gamma, \delta)$ be the i^{th} bit of S where $i = \alpha r^3 + \beta r^2 + \gamma r + \delta$. The states β'_B and γ_C have r^2 transitions rules. The process of these transitions is outlined in Figure 7b. They transition from (β'_B, γ_C) to either $(\alpha_{A0}, \delta_{D0})$ if $S(\alpha, \beta, \gamma, \delta) = 0$, or $(\alpha_{A1}, \delta_{D1})$ if $S(\alpha, \beta, \gamma, \delta) = 1$. After both transitions have happened, we test if the indices match to the actual A and D indices. We include the transition rules (α_A, α_{A0}) to (α_A, P_{A0}) and (α_A, α_{A1}) to (α_A, P_{A1}). We refer to this as the bit gadget passing a test. The two states (P_{A0}, P_{D0}) horizontally transition to $(P_{A0}, 0s)$. The $0s$ state then transitions the state δ_D to $0s$ as well as propagating the state to the right side of the assembly. If the compared indices are not equal, then the test fails and the look up states will transition to the fail states F_B or F_C. These fail states will transition with the states above them, resetting the bit gadget as in the previous system.

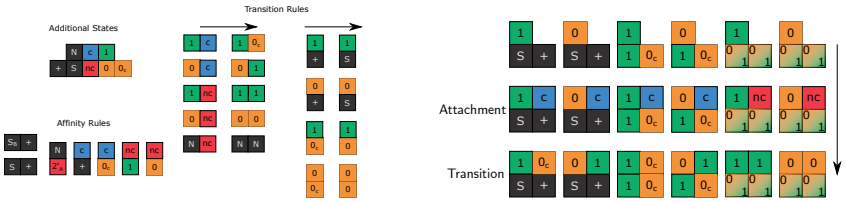
▶ **Theorem 9.** *For any binary string S with length $n > 0$, there exists a tile automata system Γ, that uniquely assembles an $(2n + 2) \times 4$ assembly which represents S with $\mathcal{O}(n^{\frac{1}{4}})$ states.*

5 Rectangles

In this section, we will show how to use the previous constructions to build $\mathcal{O}(\log n) \times n$ rectangles. All of these constructions rely on using the previous results to encode and print a string then adding additional states and rules to build a counter.

5.1 States

We choose a string and construct a system that will create that string, using the techniques shown in the previous section. We then add states to implement a binary counter that will count up from the initial string. The states of the system, seen in Figure 8a, have two purposes. The north and south states (N and S) are the bounds of the assembly. The plus,



(a) New states and rules for a binary counter **(b)** Every case for the half adder.

Figure 8 (b) The 0/1 tile is not present in the system. It is used in the diagram to show that either a 0 tile or a 1 tile can take that place.

(a) Binary Counter **(b)** Base-10 Counter

Figure 9 (a) The process of the binary counter. (b) A base-10 counter.

carry, and no carry states (+, c, and nc) forward the counting. The 1, 0, and 0 with a carry state make up the number. The counting states and the number states work together as half adders to compute bits of the number.

5.2 Transition Rules / Single Tile Half Adder

As the column grows, in order to complete computing the number, each new tile attached in the current column along with its west neighbor are used in a half adder configuration to compute the next bit. Figure 8b shows the various cases for this half adder.

When a bit is going to be computed, the first step is an attachment of a carry tile or a no-carry tile (c or nc). A carry tile is attached if the previous bit has a carry, indicated by a tile with a state of plus or 0 with a carry (+ or 0c). A no-carry tile is placed if the previous bit has no-carry, indicated by a tile with a state of 0 or 1. Next, a transition needs to occur between the newly attached tile and its neighbor to the west. This transition step is the addition between the newly placed tile and the west neighbor. The neighbor does not change states, but the newly placed tile changes into a number state, 0 or 1, that either contains a carry or does not. This transition step completes the half adder cycle, and the next bit is ready to be computed.

5.3 Walls and Stopping

The computation of a column is complete when a no-carry tile is placed next to any tile with a north state. The transition rule changes the no-carry tile into a north state, preventing the column from growing any higher. The tiles in the column with a carry transition to remove the carry information, as it is no longer needed for computation. A tile with a carry changes states into a state without the carry. The next column can begin computation when the plus tile transitions into a south tile, thus allowing a new plus tile to be attached. The assembly stops growing to the right when the last column gets stuck in an unfinished state. This column, the stopping column, has carry information in every tile that is unable to transition. When a carry tile is placed next to a north tile, there is no transition rule to change the state of the carry tile, thus preventing any more growth to the right of the column.

▶ **Theorem 10.** *For all $n > 0$, there exists a Tile Automata system that uniquely assembles a $\mathcal{O}(\log n) \times n$ rectangle using,*

- *Deterministic Transition Rules and $\mathcal{O}(\log^{\frac{1}{2}} n)$ states.*
- *Single-Transition Transition Rules and $\Theta(\log^{\frac{1}{3}} n)$ states.*
- *Nondeterministic Transition Rules and $\Theta(\log^{\frac{1}{4}} n)$ states.*

5.4 Arbitrary Bases

Here, we generalize the binary counter process for arbitrary bases. The basic functionality remains the same. The digits of the number are computed one at a time going up the column. If a digit has a carry, then a carry tile attaches to the north, just like the binary counter. If a digit has no carry, then a no-carry tile is attached to the north. The half adder addition step still adds the newly placed carry or no-carry tile with the west neighbor to compute the next digit. This requires adding $\mathcal{O}(b)$ counter states to the system, where b is the base.

▶ **Theorem 11.** *For all $n > 0$, there exists a Deterministic Tile Automata system that uniquely assembles a $\mathcal{O}(\frac{\log n}{\log\log n}) \times n$ rectangle using $\Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states.*

6 Squares

In this section we utilize the rectangle constructions to build $n \times n$ squares using the optimal number of states.

Let $n' = n - 4\lceil\frac{\log n}{\log\log n}\rceil - 2$, and Γ_0 be a determinstic Tile Automata system that builds a $n' \times (4\lceil\frac{\log n}{\log\log n}\rceil + 2)$ rectangle using the process described in Theorem 11. Let Γ_1 be a copy of Γ_0 with the affinity and transition rules rotated 90 degrees clockwise, and the state labels appended with the symbol "*1". This system will have distinct states from Γ_0, and will build an equivalent rectangle rotated 90 degrees clockwise. We create two more copies of Γ_0 (Γ_2 and Γ_3), and rotate them 180 and 270 degrees, respectively. We append the state labels of Γ_2 and Γ_3 in a similar way.

We utilize the four systems described above to build a hollow border consisting of the four rectangles, and then adding additional initial states which fill in this border, creating the $n \times n$ square.

We create Γ_n, starting with system Γ_0, and adding all the states, initial states, affinity rules, and transition rules from the other systems ($\Gamma_1, \Gamma_2, \Gamma_3$). The seed states of the other systems are added as initial states to Γ_n. We add a constant number of additional states and transition rules so that the completion of one rectangle allows for the "seeding" of the next.

Reseeding the Next Rectangle. To Γ_n we add transition rules such that once the first rectangle (originally built by Γ_0) has built to its final width, a tile on the rightmost column of the rectangle will transition to a new state pA. pA has affinity with the state $S_A * 1$, which originally was the seed state of Γ_1. This allows state $S_A * 1$ to attach to the right side of the rectangle, "seeding" Γ_1 and allowing the next rectangle to assemble (Figure 10). The same technique is used to seed Γ_2 and Γ_3.

Filler Tiles. When the construction of the final rectangle (of Γ_3) completes, transition rules propagate a state pD towards the center of the square (Figure 11). Additionally, we add an initial state r, which has affinity with itself in every orientation, as will as with state pD on its west side. This allows the center of the square to be filled with tiles.

▶ **Theorem 12.** *For all $n > 0$, there exists a Tile Automata system that uniquely assembles an $n \times n$ square with,*

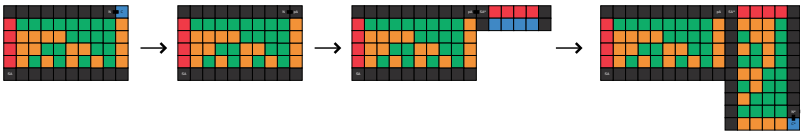


Figure 10 The transitions that take place after the first rectangle is built. The carry state transitions to a new state that allows a seed row for the second rectangle to begin growth

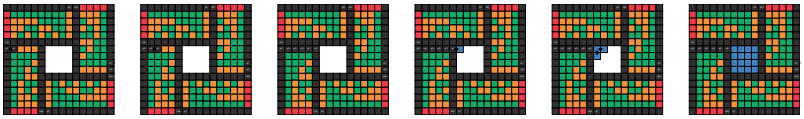


Figure 11 Once all 4 sides of the square build the pD state propagates to the center and allows the light blue tiles to fill in

- *Deterministic transition rules and* $\Theta\left(\left(\frac{\log n}{\log\log n}\right)^{\frac{1}{2}}\right)$ *states.*
- *Single-Transition rules and* $\Theta(\log^{\frac{1}{3}} n)$ *states.*
- *Nondeterministic transition rules and* $\Theta(\log^{\frac{1}{4}} n)$ *states.*

7 Future Work

This paper showed optimal bounds for uniquely building $n \times n$ squares in three variants of seeded Tile Automata without cooperative binding. En route, we proved upper bounds for constructing strings and rectangles. Serving as a preliminary investigation into constructing shapes in this model. This leaves many open questions:

- As shown in [5], even 1D Tile Automata systems can perform Turing computation. This behavior may imply interesting results for constructing $1 \times n$ lines. We conjecture, it is possible to achieve the optimal bound of $\Theta((\frac{\log n}{\log\log n})^{\frac{1}{2}})$ with deterministic rules.
- Our rectangles had a height bounded by $\mathcal{O}(\frac{\log n}{\log\log n})$, and none fell below the $k < \frac{\log n}{\log\log n}$ [2] bound for a thin rectangle. In Tile Automata without cooperative binding, is it possible to optimally construct $k \times n$ thin rectangles?
- We allow transition rules between non-bonded tiles. Can the same results be achieved with the restriction that a transition rule can only exist between two tiles if they share an affinity in the same direction?
- While we show optimal bounds can be achieved without cooperative binding, can we simulate so-called zig-zag aTAM systems? These are a restricted version of the cooperative aTAM that is capable of Turing computation.
- We show efficient bounds for constructing strings in Tile Automata. Given the power of the model, it should be possible to build algorithmically defined shapes such as in [33] by printing Komolgorov optimal strings and inputting them to a Turing machine.

References

1 Leonard Adleman, Qi Cheng, Ashish Goel, and Ming-Deh Huang. Running time and program size for self-assembled squares. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 740–748, 2001.

2 Gagan Aggarwal, Qi Cheng, Michael H Goldwasser, Ming-Yang Kao, Pablo Moisset De Espanes, and Robert T Schweller. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34(6):1493–1515, 2005.

3 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.

4 Bahar Behsaz, Ján Maňuch, and Ladislav Stacho. Turing universality of step-wise and stage assembly at temperature 1. In Darko Stefanovic and Andrew Turberfield, editors, *DNA Computing and Molecular Programming*, pages 1–11, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

5 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Verification and Computation in Restricted Tile Automata. In Cody Geary and Matthew J. Patitz, editors, *26th International Conference on DNA Computing and Molecular Programming (DNA 26)*, volume 174 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/12963`, `doi:10.4230/LIPIcs.DNA.2020.10`.

6 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.

7 Angel A Cantu, Austin Luchsinger, Robert Schweller, and Tim Wylie. Signal passing self-assembly simulates tile automata. In *31st International Symposium on Algorithms and Computation (ISAAC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

8 Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In David Doty and Hendrik Dietz, editors, *DNA Computing and Molecular Programming*, pages 155–172, Cham, 2018. Springer International Publishing.

9 Cameron Chalk, Eric Martinez, Robert Schweller, Luis Vega, Andrew Winslow, and Tim Wylie. Optimal staged self-assembly of general shapes. *Algorithmica*, 80(4):1383–1409, 2018.

10 Gourab Chatterjee, Neil Dalchau, Richard A. Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature Nanotechnology*, July 2017. URL: `https://www.microsoft.com/en-us/research/publication/spatially-localized-architecture-fast-modular-dna-computing/`.

11 Matthew Cook, Yunhui Fu, and Robert Schweller. Temperature 1 self-assembly: Deterministic assembly in 3d and probabilistic assembly in 2d. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 570–589. SIAM, 2011.

12 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.

13 Alberto Dennunzio, Enrico Formenti, Luca Manzoni, Giancarlo Mauri, and Antonio E Porreca. Computational complexity of finite asynchronous cellular automata. *Theoretical Computer Science*, 664:131–143, 2017.

14 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 *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 302–310. IEEE, 2012.

15 Nazim Fates. A guided tour of asynchronous cellular automata. In *International Workshop on Cellular Automata and Discrete Complex Systems*, pages 15–30. Springer, 2013.

16 Bin Fu, Matthew J Patitz, Robert T Schweller, and Robert Sheline. Self-assembly with geometric tiles. In *International Colloquium on Automata, Languages, and Programming*, pages 714–725. Springer, 2012.

17 David Furcy, Scott M. Summers, and Logan Withers. Improved Lower and Upper Bounds on the Tile Complexity of Uniquely Self-Assembling a Thin Rectangle Non-Cooperatively in 3D. In Matthew R. Lakin and Petr Šulc, editors, *27th International Conference on DNA Computing and Molecular Programming (DNA 27)*, volume 205 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/14671`, `doi:10.4230/LIPIcs.DNA.27.4`.

18 Oscar Gilbert, Jacob Hendricks, Matthew J Patitz, and Trent A Rogers. Computing in continuous space with self-assembling polygonal tiles. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 937–956. SIAM, 2016.

19 Eric Goles, P-E Meunier, Ivan Rapaport, and Guillaume Theyssier. Communication complexity and intrinsic universality in cellular automata. *Theoretical Computer Science*, 412(1-2):2–21, 2011.

20 Eric Goles, Nicolas Ollinger, and Guillaume Theyssier. Introducing freezing cellular automata. In *Cellular Automata and Discrete Complex Systems, 21st International Workshop (AUTOMATA 2015)*, volume 24, pages 65–73, 2015.

21 Leopold N Green, Hari KK Subramanian, Vahid Mardanlou, Jongmin Kim, Rizal F Hariadi, and Elisa Franco. Autonomous dynamic control of DNA nanostructure self-assembly. *Nature chemistry*, 11(6):510–520, 2019.

22 Daniel Hader and Matthew J Patitz. Geometric tiles and powers and limitations of geometric hindrance in self-assembly. *Natural Computing*, 20(2):243–258, 2021.

23 Jacob Hendricks, Matthew J. Patitz, Trent A. Rogers, and Scott M. Summers. The power of duples (in self-assembly): It's not so hip to be square. *Theoretical Computer Science*, 743:148–166, 2018. URL: `https://www.sciencedirect.com/science/article/pii/S030439751501169X`, `doi:https://doi.org/10.1016/j.tcs.2015.12.008`.

24 Ming-Yang Kao and Robert Schweller. Reducing tile complexity for self-assembly through temperature programming. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, page 571–580, USA, 2006. Society for Industrial and Applied Mathematics.

25 Pierre-Etienne Meunier, Matthew J. Patitz, Scott M. Summers, Guillaume Theyssier, Andrew Winslow, and Damien Woods. Intrinsic universality in tile self-assembly requires cooperation. In *Proceedings of the 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–771, 2014. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9781611973402.56`, `arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611973402.56`, `doi:10.1137/1.9781611973402.56`.

26 Pierre-Étienne Meunier and Damien Regnault. Directed Non-Cooperative Tile Assembly Is Decidable. In Matthew R. Lakin and Petr Šulc, editors, *27th International Conference on DNA Computing and Molecular Programming (DNA 27)*, volume 205 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:21, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/14673`, `doi:10.4230/LIPIcs.DNA.27.6`.

27 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*, STOC 2017, page 328–341, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3055399.3055446`.

28 Turlough Neary and Damien Woods. P-completeness of cellular automaton rule 110. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 132–143, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

29 Nicolas Ollinger and Guillaume Theyssier. Freezing, bounded-change and convergent cellular automata. *arXiv preprint arXiv:1908.06751*, 2019.

30 Matthew J. Patitz, Robert T. Schweller, and Scott M. Summers. Exact shapes and Turing universality at temperature 1 with a single negative glue. In *Proceedings of the 17th International Conference on DNA Computing and Molecular Programming*, DNA'11, page 175–189, Berlin, Heidelberg, 2011. Springer-Verlag.

31 Paul WK Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468, 2000.

32 Nicholas Schiefer and Erik Winfree. Time complexity of computation and construction in the chemical reaction network-controlled tile assembly model. In Yannick Rondelez and Damien Woods, editors, *DNA Computing and Molecular Programming*, pages 165–182, Cham, 2016. Springer International Publishing.

33 David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007.

34 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. URL: `https://www.science.org/doi/abs/10.1126/science.aan6558`, `arXiv:https://www.science.org/doi/pdf/10.1126/science.aan6558`, `doi:10.1126/science.aan6558`.

35 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.

36 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 *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 353–354, 2013.

37 Thomas Worsch. Towards intrinsically universal asynchronous ca. *Natural Computing*, 12(4):539–550, 2013.

APPENDIX B

APPENDIX B

SIMULATION OF MULTIPLE STAGES IN SINGLE BIN ACTIVE TILE SELF-ASSEMBLY

Simulation of Multiple Stages in Single Bin Active Tile Self-Assembly

Sonya C. Cirlos
University of Texas Rio Grande Valley

Timothy Gomez
Massachusetts Institute of Technology

Elise Grizzell
University of Texas Rio Grande Valley

Andrew Rodriguez
Texas State University

Robert Schweller
University of Texas Rio Grande Valley

Tim Wylie (timothy.wylie@utrgv.edu)
University of Texas Rio Grande Valley

Research Article

Keywords: Staged Self-assembly, Tile Automata, Context-Free Grammar, Freezing TA

Posted Date: December 21st, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3762430/v1

Additional Declarations: No competing interests reported.

Simulation of Multiple Stages in Single Bin Active Tile Self-Assembly

Sonya C. Cirlos[1], Timothy Gomez[2], Elise Grizzell[1], Andrew Rodriguez[3], Robert Schweller[1], Tim Wylie[1*†]

[1*]Department of Computer Science, University of Texas Rio Grande Valley, 1201 W. University Dr., Edinburg, TX, 78539, USA.
[2]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, MA, 02139, USA.
[3]Department of Computer Science, Texas State University, San Marcos, TX, 78666, USA.

*Corresponding author(s). E-mail(s): timothy.wylie@utrgv.edu;
Contributing authors: sonya.cirlos01@utrgv.edu; tagomez7@mit.edu;
elise.grizzell01@utrgv.edu; andrew.rodriguez@txstate.edu;
robert.schweller@utrgv.edu;
†These authors contributed equally to this work.

Abstract

Two significant and often competing goals within the field of self-assembly are minimizing tile types and minimizing human-mediated experimental operations. The introduction of the Staged Assembly and Single Staged Assembly models, while successful in the former aim, necessitate an increase in mixing operations later. In this paper, we investigate building optimal lines as a standard benchmark shape and building primitive. We show that a restricted version of the 1D Staged Assembly Model can be simulated by the 1D Freezing Tile Automata model with the added benefits of the complete automation of stages and completion in a single bin while maintaining bin parallelism and a competitive number of states for lines, patterned lines, and context-free grammars.

Keywords: Staged Self-assembly, Tile Automata, Context-Free Grammar, Freezing TA

1 Introduction

Many molecular programmers dream of designing single-pot reactions in which system molecules do the entirety of the computational work without any necessary intervention by the experimenter. This is arguably *true* self-assembly. Yet the power of experimenter intervention, in the form of mixing and splitting pots over a sequence of stages, yields power and efficiency in both theory and practice [18] that is currently unmatched even with some of the most powerful models of active self-assembly. This paper aims to address this gap in the case of 1-dimensional (1D) assembly by showing how an abstract modeling of operations of experimental stages, termed the Staged Assembly Model (SAM) [12], can be efficiently simulated by an abstract model of single-pot *active* self-assembly, termed Tile Automata (TA) [9].

Tile Automata generalizes *passive* tile assembly models (such as the two-handed tile assembly model [7]) by giving tiles dynamic states that update based on local pairwise rules, thus making it a model of *active* self-assembly. The Staged Assembly Model (SAM) generalizes tile assembly models by the modeling of experimenter-mediated operations, including the ability to store different portions of the system particles in separate containers or *bins*, and the ability to combine separate bins or split the contents of a bin among multiple bins, over a sequence of distinct *stages*. Previous results show that both models have substantially increased power over the basic tile self-assembly models they generalize. In particular, by offloading some of the computation onto an experimenter responsible for performing the required mixing operations of the system between stages, SAM can build complex shapes and patterns in near-optimal complexity with respect to tile types, bin counts, and stage counts [10–13, 20].

In answer to the long-standing open question of whether the substantial power of the SAM could be efficiently encoded into the reaction rules of an active single-pot system, this paper shows that in the case of 1-dimensional systems, any staged system can be encoded into a single-pot TA system with a comparable state and rule space to the tiles, bins, and stages of the SAM system it simulates. This result provides a corresponding corollary in TA for any results in 1D staged self-assembly. Further, this provides a new approach for programming 1D TA systems since designing staged systems is relatively simple with strong timing guarantees based on separate bins and stages, whereas programming complex TA systems from scratch can be daunting as the single-pot nature of the system requires careful attention to race conditions. As evidence of the power of this new result, we show how several previous results in TA now become simple corollaries of this new result. Further, we show how a general linear pattern can be constructed in TA using a number of states linear in the size of the smallest context-free grammar that produces the target pattern.

1.1 Staged Self-Assembly and Tile Automata

Algorithmic self-assembly emerged from a formalization of Wang Tiles to explore self-assembling structures. Defined by Winfree in [19], this was partially motivated by new DNA techniques that allow for the creation of DNA-based 'tiles' that can assemble into lattice structures at the nanoscale [22]. Further experimental work has investigated

active DNA-based components capable of complex tasks such as sorting molecules attached to a DNA origami surface [17].

The Staged Tile Assembly Model [12] generalizes the 2-Handed Assembly Model to allow growth to occur in multiple bins, mixing in a sequence described as stages, creating the capability to model experimental techniques, such as in [18] where 2D patterns are built with DNA origami tiles in multiple stages.

Tile Automata was introduced in [9] as a combination of hierarchical passive self-assembly systems and the active self-assembly of Cellular Automata systems where all *tiles* have a transitionable *state*. Affinity rules define which tiles can bond with each other based on their states and with how much strength. Starting from singleton tiles with states, any two producibles in the system may combine if there is enough affinity between adjacent tiles. Transition rules define state changes that may occur between two tiles once they are neighbors in an assembly.

Efficient line construction in Tile Automata was briefly studied in [5].

1.2 Related Work

Shape building was the first problem explored when the staged model was introduced [12]. In the staged model, a constant-sized set of glue types is sufficient to build any shape by encoding the description in the mix graph. The trade-off between the number of glues, bins, and stages was further investigated in later work with $1 \times n$, $\mathcal{O}(1) \times n$ [11], and general assemblies [10]. The complexity of verifying whether an assembly is uniquely produced is PSPACE-complete [6, 15].

A restricted class of systems in SAM, called Single Staged Assembly Systems (SSAS) in [13], requires each bin to only contain one terminal assembly built from two input assemblies. This restriction eliminates having multiple assemblies built in the same bin (*bin parallelism*). The size of the smallest SSAS that builds a 1D pattern $\mathcal{P}$ is equivalent (up to constant factors) to the size of the smallest Context-Free Grammar (CFG) that defines only $\mathcal{P}$. However, when bin parallelism is allowed, staged is more efficient than CFGs for a specific family of strings.

In [20], they built on previous results and define Polyomino Context-Free Grammars (PCFG), which generalize CFGs to 2D. The size of the smallest staged system that uniquely produces a patterned assembly is within a log factor of the smallest PCFG. In some cases, staged is much better.

One strength of Tile Automata is the possibility of being a "unifying" model, where multiple models can be connected through simulation results. The work that introduced the model [9] showed that the freezing model, where a tile may never repeat a state, simulates the non-freezing version of the model. Tile Automata was shown to simulate a model of programmable matter called Amoebots [2]. The chain of simulation was further extended in [8] where the Signal-Passing Tile Assembly Model (STAM) was shown to simulate Tile Automata. Work done in [3] shows how the 1D STAM can simulate a s stage 1D SSAS system using a single tile with $\mathcal{O}(s^4)$ glues types.

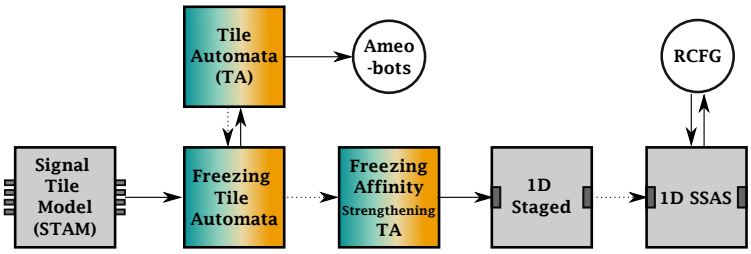


Fig. 1: Informal map of relations between models. Dotted line arrows indicate model is a special case of the previous. Solid lines indicate simulation results.

Tile Automata	Scale	States	Theorem
Freezing Strengthening	1	$\mathcal{O}(sbt)$	Thm.
Freezing Strengthening	2	$\mathcal{O}(sbg)$	Thm.
Strengthening	2	$\mathcal{O}(sg + bg)$	Open

Table 1: Restricted 1D Tile Automata can simulate 1D Staged model. We allow for 1D scaling. s is number of states, b is number of bins, g is the number of glues.

Aff Str	Cycle	Frz.	Det.	Single			Double		
Yes	Yes	No	ND	$\mathcal{O}(\lvert P\rvert^{\frac{1}{3}})$	2×3	[1]	$\mathcal{O}(\lvert P\rvert^{\frac{1}{4}})$	2×4	[1]
Yes	No	Yes	Det	$\mathcal{O}(\lvert P\rvert^{\frac{1}{2}})$	1×2	[1]	$\mathcal{O}(\lvert P\rvert^{\frac{1}{2}})$	1×1	[1]
No	Yes	Yes	Det	$\mathcal{O}(K_P)$	$\mathcal{O}(1) \times \mathcal{O}(1)$	[5, 8]	$\mathcal{O}(K_P)$	1×1	[5]
No	No	No	Det	$\mathcal{O}(K_P^{\frac{1}{2}})$	1×1	Thm. 4	$\mathcal{O}(K_P^{\frac{1}{2}})$	1×1	Thm. 4
Yes	No	No	Det	$\mathcal{O}(L_P^{\frac{1}{2}})$	$\mathcal{O}(1) \times 2$		$\mathcal{O}(L_P^{\frac{1}{2}})$	$\mathcal{O}(1) \times 1$	
Yes	No	Yes	ND	$\mathcal{O}(CF_P)$	1×1	Thm 2	$\mathcal{O}(CF_P)$	1×1	Thm. 2

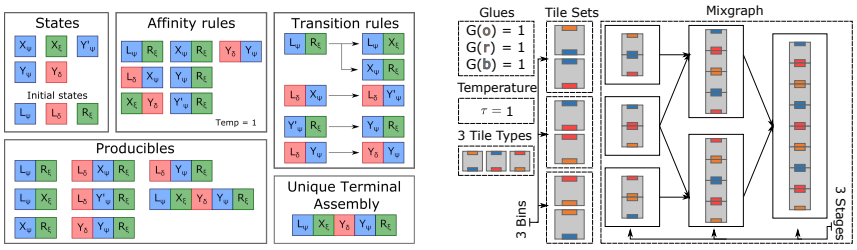
Table 2: Minimum number of states needed to construct a patterned rectangle over a constant number of colors representing the 1D pattern P in Affinity Strengthening Tile Automata with tiles not changing colors. K_P is the Kolmogorov complexity of the pattern P, CF_P is the size of the smallest Context Free Grammar that produces the singleton language $\{P\}$.

1.3 Our Contributions

We show that the 1D version of Freezing Affinity Strengthening Tile Automata can simulate the 1D staged assembly model, even with flexible glues (Section 3). The Tile Automata system uses $\mathcal{O}(sbt)$ states for a system with s stages, b bins, and t tile types.

Using this result we inherit the ability to simulate Context-Free Grammars from the staged model in [13] showing the same upper bound. For the line-building results, we inherit them from [12]. Additionally using results from [8], these results carry over to the STAM as well.

This is the full version of a paper presented at UCNC 2023. We include additional upper and lower bounds on pattern building in different versions of Tile Automata.



(a) A Tile Automata System (b) Staged Self-Assembly Example

Fig. 2: (a) 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 terminal. Note that none of the transition rules allow states to change color. (b) A simple staged self-assembly example. The system has 3 bins, 3 stages, and 3 tile types, assigned to bins, as shown in the mix graph. Only terminal assemblies can pass to a successive stage. The result of this system is the assembly shown in the bin in stage 3.

The result in Section 4 is a direct version of a Context-Free Grammar simulation which works in a slighty stronger version of Tile Automata, i.e., Theorem 2 works even in the case of Deterministic Single-Transitions. We additionally include bounds on building patterns in relaxed versions of Tile Automata and these results are outlined in Table 2.

2 Model and Definitions

We provide simplified definitions for 1D Tile Automata, then define 1D Staged Assembly as a generalization. Refer to previous work [1] and [12] for full definitions of the models.

2.1 The 1D Tile Automata model (TA)

In this dimensionally restricted version of the model, a *Tile Automata system*[1] is a triple (Σ, Π, Δ) where Σ is an alphabet of state types, Π is an affinity function, and Δ is a set of transition rules for states in Σ. An example 1D Tile Automata system is shown in Figure 2.

[1]Typical TA models are defined with a temperature parameter τ however, with consideration of solely 1D, eliminating the possibility of cooperative binding, we assume $\tau = 1$.

Tile. Let Σ be a set of *states* or symbols. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^1$ and has a state of $\sigma \in \Sigma$.

Assembly. An assembly A is a sequence of tiles $\{t_1, t_2, t_3, \ldots, t_{|A|}\}$. Let $A(i)$ and $A_\Sigma(i)$ represent the i^{th} tile and its state in assembly A, respectively. For a tile t in assembly A let $\rho_A(t)$ be the position of t in A.

Affinity Function. An *affinity function* Π takes an ordered pair in Σ^2 as input and outputs either 0 or 1. The *affinity strength* between two states for the ordered orientation is the binary output of the corresponding function. An assembly A is *stable* if, for every pair of tiles, $\Pi(A_\Sigma(i), A_\Sigma(i+1)) = 1$. Informally, if all adjacent tiles in assembly A have an affinity, A is stable. Two assembles, A and B are *combinable* if the concatenation of the two assemblies $AB = C$ is also a stable assembly.

Transition Rules. Transition rules allow states to change based on their neighbors. A *transition rule* is denoted $(\sigma_{1a}, \sigma_{2a}) \to (\sigma_{1b}, \sigma_{2b})$ with $\sigma_{1a}, \sigma_{2a}, \sigma_{1b}, \sigma_{2b} \in \Sigma$. If states σ_{1a} and σ_{2a} are adjacent to each other, they can transition to states σ_{1b} and σ_{2b}, respectively. An assembly A is *transitionable* to an assembly B if there exists two adjacent tiles $A(i), A(i+1) \in A$, two adjacent tiles $B(i), B(i+1) \in B$, a transition rule $(A_\Sigma(i), A_\Sigma(i+1)) \to (B_\Sigma(i), B_\Sigma(i+1)) \in \Delta$, and $A(j) = B(j)$ for all $j \neq i, i+1$.

Producibility. We define the set of producible assemblies starting from a set of initial assemblies Λ. For a given 1D Tile Automata system $\Gamma = (\Sigma, \Pi, \Delta)$ and initial assembly set Λ, the set of *producible assemblies* of Γ, denoted $\texttt{PROD}_\Gamma(\Lambda)$, is defined recursively:

- (Base) $\Lambda \subseteq \texttt{PROD}_\Gamma(\Lambda)$
- (Combinations) For any $A, B \in \texttt{PROD}_\Gamma(\Lambda)$ s.t. A and B are combinable into C, then $C \in \texttt{PROD}_\Gamma(\Lambda)$.
- (Transitions) For any $A \in \texttt{PROD}_\Gamma(\Lambda)$ s.t. A is transitionable into B using $\delta \in \Delta$, then $B \in \texttt{PROD}_\Gamma(\Lambda)$.

For a system Γ, we say $A \to_1^\Gamma B$ for assemblies A and B if A is combinable with some producible assembly to form B, if A is transitionable into B, or if $A = B$. Intuitively, this means that A may grow into assembly B through one or fewer combinations or transitions.
We define the relation $\to^\Gamma$ to be the transitive closure of $\to_1^\Gamma$, i.e., $A \to^\Gamma B$ means that A may grow into B through a sequence of combinations and transitions.

Terminal Assemblies. A producible assembly A of a Tile Automata system Γ is *terminal* provided A is not combinable with any producible assembly of Γ, and A is not transitionable to any producible assembly of Γ. Let $\texttt{TERM}_\Gamma(\Lambda) \subseteq \texttt{PROD}_\Gamma(\Lambda)$ denote the set of producible assemblies of Γ that are terminal.

Unique Assembly. A 1D TA system Γ, starting from initial assemblies Λ, *uniquely produces* a set of assemblies $\mathcal{A}$ if

- $\mathcal{A} = \texttt{TERM}_\Gamma(\Lambda)$,
- for all $B \in \texttt{PROD}_\Gamma(\Lambda)$, $B \to^\Gamma A$ for some $A \in \mathcal{A}$

2.2 Staged Assembly Model

Here, we define the Staged Assembly model using the definitions from above.

Tile Types and Glues. In the staged assembly model, tiles are defined by their glues. Let G be a set of glues. A *tile type* is an ordered pair of glues $(w, e) \in G^2$ where tile $t = (w, e)$ has west glue w and east glue e. The affinity function Π for the staged assembly model takes as input two tile types $t_1 = (a, b)$, $t_2 = (c, d)$ and outputs 1 if $b = c$ and 0 otherwise.

When allowing *Flexible Glues* we remove the restriction that Π outputs 0 when $b \neq c$ allowing for a general glue function. *Note this is equivalent to the affinity function of Tile Automata.*

Assembly. An assembly A in a staged assembly system is a sequence of tile types $\{t_1, t_2, t_3, \ldots, t_{|A|}\}$. Let $A(i)$ be the i^{th} tile type in assembly A.

Staged Assembly Systems. An *r-stage, b-bin mix-graph* $M_{r,b}$, is an acyclic r-partite digraph consisting of rb vertices $m_{i,j}$ for $1 \leq i \leq r$ and $1 \leq j \leq b$, and edges of the form $(m_{i,j}, m_{i+1,j'})$ for some i, j, j'. A *staged assembly system* is a duple $\Upsilon = (M_{r,b}, T)$ where $M_{r,b}$ is an r-stage, b-bin mix-graph, $T \subset G^2$ is a set of tiles types labeled from the set of pairs of glues G.

Two-Handed Assembly and Bins We define the assembly process in terms of bins[2]. Each bin can be considered an instance of a Tile Automata system without transition rules where $\Delta = \emptyset$. However, each bin has a different set of initial assemblies denoted as $\Lambda_{i,j}$ where i is the stage and j is the bin. Let T_j be the set of initial tile types in bin j.

1. $\Lambda_{1,j} = \{T_j\}$ (this is a bin in the first stage);
2. For $i \geq 2$, $\Lambda_{i,j} = \Big(\bigcup_{k:\ (m_{i-1,k}, m_{i,j}) \in M_{r,b}} \texttt{TERM}_\Upsilon(\Lambda_{i-1,k}) \Big)$.

Thus, the j^{th} bin in stage 1 is provided with the initial tile set T_j. Each bin in any later stage receives an initial set of assemblies consisting of the terminally produced assemblies' bins in the previous stage indicated by the edges of the mix-graph. The *output* of the staged system is the union of all terminal assemblies from each bin in the final stage. We say this set of output assemblies is *uniquely produced* if each bin in the staged system uniquely produces its respective set of terminal assemblies.

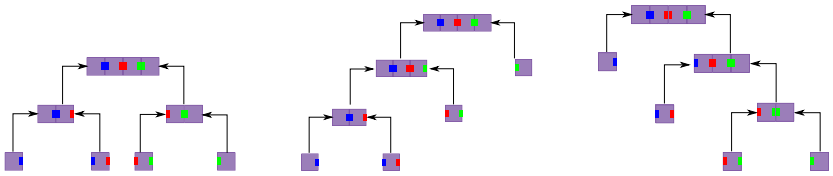
2.3 Assembly Trees

We may represent the assembly process in a single bin as an assembly tree in the staged model. An example tree can be seen in Figure 3a.

Definition 1 (Assembly Tree)**.** *An assembly tree T_A^b, for a producible assembly A in a bin b, is a binary tree where each node represents a subassembly of A. The root represents assembly A, and each leaf represents an initial assembly of b. Each node can be formed by combining the assemblies represented by the children.*

An assembly tree is a *Left-Handed* Assembly Tree if every assembly that attaches on the right side is an initial assembly. A *Right-Handed* Assembly Tree is the inverse

[2]Each bin may be seen as an instance of the 2-Handed Assembly Model.



(a) Assembly Tree (b) Left Handed (c) Right Handed

Fig. 3: Examples of assembly trees for the same assembly. (a) A balanced tree. (b) A left-handed assembly tree. (c) A right-handed assembly tree.

where every left assembly is an initial assembly. Examples of these two types of trees are in Figures 3b and 3c.

2.4 Colors and Patterns

In this section, we augment the Tile Automata model with the concept of a tile's color being based on the current state. Colors for Staged has been defined in [13]. For a set of color labels C, this is a partition of the states into $|C|$ sets. We only consider constant-sized C. Thus, the *color* of a tile t is the partition of the tile's state, denoted as $c(t)$.

Definition 2 (Pattern)**.** *A pattern P over a set of colors C is a partial mapping of $\mathbb{Z}$ to elements in C. Let $P(z)$ be the color at $z \in \mathbb{Z}$. A scaled pattern P^{hw} is a pattern replacing each pixel within a $1 \times w$ line of pixels.*

Definition 3 (Patterned Assemblies)**.** *We say a positioned assembly A' represents a pattern P if for each tile $t \in A'$, $c(t) = P(\rho_{A'}(t))$ and $dom(A') = dom(P)$. We say a positioned assembly B' represents a pattern P at scale $h \times w$ if it represents the scaled pattern P^{hw}.*

A system Γ uniquely assembles a pattern P if it uniquely assembles an assembly A, such that A contains a positioned assembly that represents P.

2.5 Tile Automata Restrictions

Here we define the relevant restrictions of Tile Automata. All but the last has been defined in previous work [1, 5, 8, 9]

Affinity Strengthening. *Affinity Strengthening* requires that any transition preserves affinities between tiles within assemblies. For each transition rule $(\sigma_a, \sigma_b) \rightarrow (\sigma_c, \sigma_d)$, $\Pi(\sigma_c, \sigma_d) = 1$. By limiting our focus to affinity strengthening systems, we

do not need to consider the scenario where a stable assembly becomes unstable (and would fall apart).

Freezing. In a freezing system, a tile may not transition to any state more than once. Thus, if a tile with state σ_a transitions into another state σ_b, it is not allowed to transition back to σ_a.

Bonded. Transitions only occur between tiles that have affinity with each other.

Single-Transition Tile Automata system. Γ is a Single-Transition Tile Automata system if for all transitions rules $(S_{1a}, S_{2a}, S_{1b}, S_{2b}, d)$ either $S_{1a} = S_{1b}$ or $S_{2a} = S_{2b}$.

Bonded, Single-Transition allows us to skip a couple steps in the simulation in the STAM from [8].

Deterministic Transition Rules. A system has deterministic transitions rules if for all pairs of states S_1, S_2 and direction $d \in \{v, h\}$ there only exists one transition rule between the states in that direction.

Color-Locked. A tile automata system is *Color-Locked* if for every transition rule $\delta = (S_{1a}, S_{2a}, S_{1b}, S_{2b}, d) \in \Delta$, $c(S_{1a}) = c(S_{1b})$ and $c(S_{2a}) = c(S_{2b})$, i.e. tiles are not allowed to change their color.

This restriction allows for transitions to be independent of the color, we can imagine this the color being inherent to the tile. These restrictions all together can model a signal tile carrying a chemical marker that cannot change, and transitions only expose more binding sites.

3 Simulation of General 1D Staged

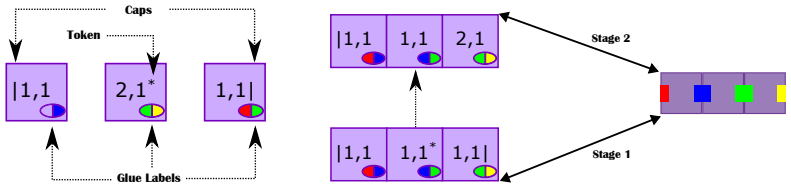
In this section, we show how to simulate all 1D staged systems with TA systems. First, we define what simulate means for these systems, followed by a high-level overview of our simulation, and then the details.

3.1 Simulation

Here, we utilize a simplified definition of simulation in which the set of final terminal assemblies, from the *target* staged system to be simulated, is exactly the same, under a mapping function, as the final terminal assemblies of the *source* TA system that is simulating it. This is a standard type of simulation used, and we omit technical definitions in this version. A stronger definition of simulation incorporates *dynamics*, in which assemblies may attach in the target system if and only if they attach in the source system. However, our approach focuses on simulating a restricted set of dynamics that are sufficient to ensure the production of all final (and partial) assemblies. We leave the problem of fully simulating the dynamics of a staged system as future work.

3.2 Overview

We create a Tile Automata system with initial tiles representing the initial tile types of the staged system. Each assembly in our Tile Automata system represents an assembly in a specific stage and bin. Each state is a pair consisting of a tile type t and a



(a) Tiles (b) Assemblies

Fig. 4: (a) Each of our Tile Automata states conceptually represents two glue labels that say which tile type they map to (a glue may be null, as in the leftmost state). They may also contain features such as the left/right cap or the active state token. (b) Assemblies map based on the glue labels on the Tile Automata states. Multiple Tile Automata assemblies represent the same Staged assembly, but sometimes in different stages.

stage-bin label representing t in that specific stage and bin. Some states will have an *active state token*(*) used to track the progress of the Tile Automata assembly in the assembly tree. We simulate only left- or right-handed assembly trees based on the parity of the stage number. The logic for the transition rules is described in Algorithm 1 using a *Glue-Terminal Table*. Each Tile Automata assembly builds according to the assembly trees of the staged system by having the token "read" the glues to decide if an assembly is terminal in a bin and needs to transition to the next stage.

3.3 Glue-Terminal Table

For the simulation to work, we need to know the glues used in each bin of the target system because we cannot "read" the absence of a glue/assembly in self-assembly. However, we can use the Glue-Terminal Table to construct the transition rules. This table stores which glues correspond with each bin.

Definition 4 (Glue-Terminal Table)**.** *For a staged system $\Upsilon = (M_{r,b}, T)$, the Glue-Terminal table $GT((s,b),g)$ is a binary $|M_{r,b}| \times G$ table with rows labeled with stage-bin pairs and columns labeled with glues. The entry $GT((s,b),g)$ is true (Used) if there exists at least two producible assemblies in bin b that attach using glue g in stage s. If it is false (Term.), the glue is never used in bin b for stage s.*

This table can be computed recursively by checking the glues of the that are assemblies in the previous bin. Computing terminal assemblies can be done much easier since it's 1D.

3.4 States and Initial Tiles

A state in our Tile Automata system has the following properties: each state has the first two properties and the second two properties are optional. The first label has sb possible options, the second has t, and the rest only increase the state space by a constant factor. This results in an upper bound on the states used of $\mathcal{O}(sbt)$.

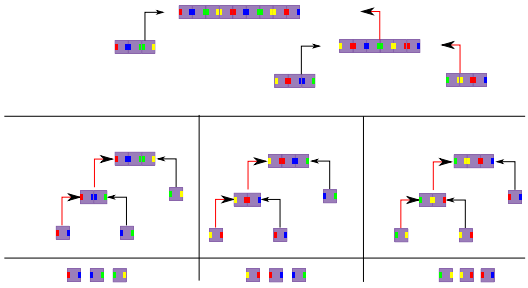
- **Stage-Bin Label.** Each state $(s,i)_t$ is labeled with a pair of integers (s,i) saying the state represents the i^{th} bin in stage s.

Algorithm 1 Algorithm to create transition rules for each pair of states in a Tile Automata system.

Data: Left state a and right state b, and glue-terminal table GT.
Result: Transition rule $(a, b) \to (a', b')$ if such a rule exists.
Let $L(\sigma)/R(\sigma)$ be the left/right glue label of the tile type σ maps to.
Let $\texttt{STAGE}(\sigma)$ be the stage σ is in. Let $\texttt{BIN}(\sigma)$ be the bin σ is in.
Let $\texttt{NEXT\_BIN}(\sigma)$ be the bin σ will be in the next stage.
Let $\texttt{HAS\_TOKEN}(\sigma)$ be *true* if σ contains a token, *false* otherwise.

if $R(a) \neq L(b)$ **then**
  Return null
if $\texttt{HAS\_TOKEN}(a) \wedge \texttt{STAGE}(a)$ is odd **then**
  if b has a right cap **then**
    if $GT((\texttt{STAGE}(b), \texttt{BIN}(b)), R(b)) = \textit{Used}$ **then**
      $a' \leftarrow a - *$; $b' \leftarrow b + *$; $b' \leftarrow b' - |$
    else if $GT((\texttt{STAGE}(b) + 1, \texttt{NEXT\_BIN}(b)), R(b)) = \textit{Used}$ **then**
      $a' \leftarrow a - *$; $b' \leftarrow b - |$ $\texttt{STAGE}(b') \leftarrow \texttt{STAGE}(b') + 1$; $\texttt{BIN}(b') \leftarrow \texttt{NEXT\_BIN}(b')$
    else
      $a' \leftarrow a$; $b' \leftarrow b$ $\texttt{STAGE}(a') \leftarrow \texttt{STAGE}(a') + 1$; $\texttt{BIN}(a') \leftarrow \texttt{NEXT\_BIN}(a')$ $\texttt{STAGE}(b') \leftarrow \texttt{STAGE}(b') + 1$; $\texttt{BIN}(b') \leftarrow \texttt{NEXT\_BIN}(b')$
  else
    $a' \leftarrow a - *$; $b' \leftarrow b + *$ $\texttt{STAGE}(b') \leftarrow \texttt{STAGE}(b') + 1$; $\texttt{BIN}(b') \leftarrow \texttt{NEXT\_BIN}(b')$
  Return $(a, b) \to (a', b')$
if $\texttt{HAS\_TOKEN}(b) \wedge \texttt{STAGE}(b)$ is even **then**
  if a has a left cap **then**
    if $GT((\texttt{STAGE}(a), \texttt{BIN}(a)), L(a)) = \textit{Used}$ **then**
      $b' \leftarrow b - *$; $a' \leftarrow a + *$; $a' \leftarrow a' - |$
    else if $GT((\texttt{STAGE}(a) + 1, \texttt{NEXT\_BIN}(a)), L(a)) = \textit{Used}$ **then**
      $b' \leftarrow b - *$; $a' \leftarrow a - |$ $\texttt{STAGE}(a') \leftarrow \texttt{STAGE}(a') + 1$; $\texttt{BIN}(a') \leftarrow \texttt{NEXT\_BIN}(a')$
    else
      $b' \leftarrow b$; $a' \leftarrow a$ $\texttt{STAGE}(b') \leftarrow \texttt{STAGE}(b') + 1$; $\texttt{BIN}(b') \leftarrow \texttt{NEXT\_BIN}(b')$ $\texttt{STAGE}(a') \leftarrow \texttt{STAGE}(a') + 1$; $\texttt{BIN}(a') \leftarrow \texttt{NEXT\_BIN}(a')$
  else
    $b' \leftarrow b - *$; $a' \leftarrow a + *$ $\texttt{STAGE}(a') \leftarrow \texttt{STAGE}(a') + 1$; $\texttt{BIN}(a') \leftarrow \texttt{NEXT\_BIN}(a')$
  Return $(a, b) \to (a', b')$

- **Glue Labels.** Each state $(s, i)_t$ represents a tile t from the staged system. We say this state has the glue labels of t when defining our affinity rules in Tile Automata. This label also defines our mapping from TA states to staged tiles in both directions.
- **Active State Token.** A state $(s, i)_t^*$ may have an Active State Token *. The token is used to enforce the left/right handed assembly trees by starting on one side of an assembly, and allowing attachment to other states with matching glue and stage-bin labels.



(a) Staged System

(s, b)	Red	Blue	Green	Yellow
$(1,1)$	Term	Used	Used	Term
$(1,2)$	Used	Used	Term	Term
$(1,3)$	Used	Term	Term	Used
$(2,1)$	Term	Term	Used	Used

(b) Glue-Terminal Table

Fig. 5: (a) Example Staged system to be simulated. (b) Glue-Terminal Table for shown staged system. In the table, s is the stage and b is the bin.

- **Caps.** A state may have a cap on one side, denoted $|s,i)_t$ or $(s,i|_t$. This means that on the side of the cap $|$, there are no affinity rules for that state. Until an assembly is ready to attach, it will have caps on its left and right most tiles.

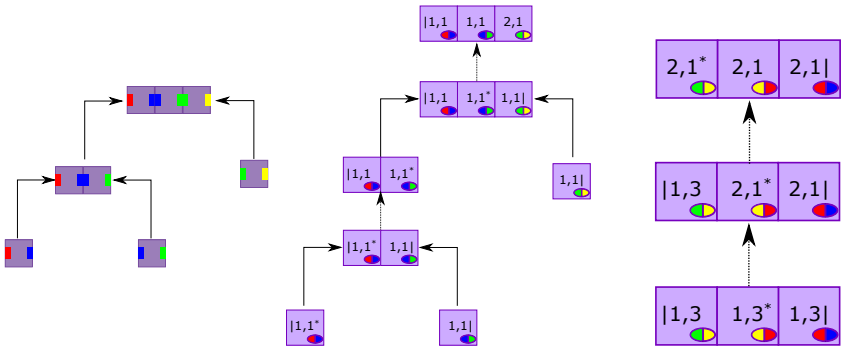
We create an initial state for each pair $b_{1,i}, t$ where $b_{1,i}$ is the i^{th} bin of the first stage and t is a tile input to that bin. If the left glue of the t is used in the $b_{1,i}$, then we include the state $(1,i_t|$, i.e., the right cap state. If the left glue is open, but the right glue is used, the tile is the first in a left-handed assembly tree. In this case, we include the token left cap state $|1,i_t^*)$.

If a tile is terminal in the first bin, we instead include an initial state representing the first bin where the state is consumed. For example, if a tile t is input to bin $(1,i)$ and is terminal, but its right glue is used in an attachment in bin $(2,j)$ (where there's an edge between $(1,i)$ and $(2,j)$), then we instead include an initial state $|2,j_t)$.

3.5 Bin Simulation

In any odd stage, we construct every terminal using a sequence of attachments representing a left-handed assembly tree. For even stages, we use a right-handed assembly tree. We control this with the token by defining our affinity rules such that every attachment occurs between one state with the token and one without a cap. We switch between the left and right handed trees to reduce the amount of times the token must walk back and forth on the assembly since the token ends on the opposite side each time.

We walk through an example of a bin in the first stage in Figure 6a. The token left cap state $|1,1_t^*)$ attaches to the right cap state $(1,1_{t'}|$ if t' attaches to the right



(a) $(1, 1)$ (b) $(1, 3)$

Fig. 6: (a) Example simulation of an assembly in stage 1. Notice the token moves leftward through the assembly as it builds to enforce a left handed assembly tree. (b) Transition for terminal assembly in bin $(1, 3)$. Since the rightmost glue is terminal in bin $(1, 3)$ the token changes the stage to 2 and starts moving left to remove the cap.

of t. These two states then transition. If the right glue of t' is used in the bin, the token moves to that state and removes the cap. This process can then repeat in the bin. Looking at the next tile t'', the right glue is unused, and thus, the assembly is terminal, and the transition should move it to the next stage, now changing directions as outlined in Figure 6b. The process for defining transitions is described in Algorithm 1; when given two states and the Glue-Transition table, a transition rule is returned if one would exist in the system. Note that this algorithm is non-deterministic as one bin may output to multiple bins in the next stage, so a pair of states may have multiple transition rules.

Theorem 1. *For any 1D staged system Υ with flexible glues, s stages, b bins, and t tile types, there exists a 1D Freezing Affinity-Strengthening Tile Automata system Γ with $\mathcal{O}(sbt)$ states that simulates Υ.*

Proof. Consider a staged system $\Upsilon = (M_{r,b}, T)$ with s stages, b bins and t tiles types. Tile Automata system $\Gamma = (\Sigma, \Pi, \Delta)$ which simulates Υ is defined and discussed below.

State complexity $\mathcal{O}(sbt)$. Each tile type in Υ requires a unique state in Γ for every bin in every stage, resulting in $s \cdot b \cdot t$ states. The additional state increase for the token and caps of each state is constant for a total of $\mathcal{O}(sbt)$ states.

Flexible Glues, Freezing and Affinity Strengthening. A state $\sigma_t \in \Sigma$ with tile type $t \in T$ has affinity with a state $\sigma'_t \in \Sigma$ with tile type $t' \in T$ if t attaches to t' in Υ. With the affinity function we can encode general glues so we can simulate flexible glues. For every transition rule $\delta \in \Delta$, δ does not alter the tile type a state represents since only the stage, bin, token, or cap are affected.

Every transition rule is freezing and either removes a cap, moves the token forward, or advances to the next stage. Once a state with a tile type t has lost its cap it can never regain it. In a single stage, the token may walk over each tile a maximum of 2 times as both sides of the assembly must be checked to decide if the assembly is

terminal. Note that this token walk involves adding an additional distinct state so the tiles do not visit the same state twice.

Simulation. We prove this is a correct simulation by induction on the size of the assemblies. The initial assemblies cover our base case for single tiles in Λ. The tile input in the first stage in Υ ensures each included assembly is in Λ. For the recursive case, assume every assembly $A \in \texttt{PROD}_\Upsilon$ with $|A| < x$ is simulated. Let b be the bin in which A is produced. A must be produced using two assemblies B and C, each of size $< x$, which are also in bin b. From our assumption, B and C have assemblies representing them- $B', C' \in \texttt{PROD}_\Gamma(\Lambda)$. Since B and C are produced in the same bin and have matching assemblies B' and C' with matching tokens, they may combine into an assembly A'. A will represent A since it has the same labels. □

3.6 Lines

Using Theorem 1, we provide an alternate proof from [5] of length-n lines with $\mathcal{O}(\log n)$ states.

Corollary 1. *For all $n \in \mathbb{N}$, there exists a freezing Tile Automata system that uniquely assembles a $1 \times n$ line in $\mathcal{O}(\log n)$ states.*

Proof. In [12], it is shown that there exists a staged assembly system that uniquely produces a $1 \times n$ line with 6 tile types, 7 bins, and $\mathcal{O}(\log n)$ stages. From theorem 1, there exists a Freezing Affinity-Strengthening Tile Automata system Γ with $\mathcal{O}(sbt)$ states that simulates any staged system Υ with s stages, b bins and t tile types. Therefore, simulating the staged assembly system from [12] can be done with $\mathcal{O}(\log n)$ states. □

4 Freezing Affinity Strengthening

While the results in the previous section imply that you may implement Context Free-Grammar (CFGs) by simulating 1D Staged, here we provide a direct simulation of CFGs. This direct simulation has the advantage of being deterministic and single transition. An example CFG is shown in Figure 7, along with the corresponding TA system in Figure 8. In addition to the freezing and affinity strengthening constraints, this result achieves the feature that tiles never undergo a change in their color throughout the assembly process. We denote rules that adhere to this constraint as *color-locked* rules.

4.1 Context-Free Grammars

A **context-free grammar (CFG)** is a set of recursive rules used to generate patterns of strings in a given language. A CFG is defined as a quadruple $G = (V, \Upsilon, R, S)$. V represents a finite set of non-terminal symbols and Υ is a finite set of terminal symbols. The symbol R is the set of production rules and S is a special variable in V called the start symbol. Production rules R of CFGs are in the form $A \to BC|a$, with V in the left-hand side and V and/or Υ on the right- hand side. A CFG derives a string through recursively replacing nonterminal symbols with terminal and non-terminal symbols based on its production rules.

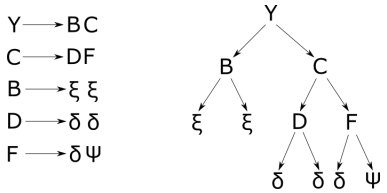


Fig. 7: A restricted context-free grammar (RCFG) G and its corresponding parse tree that produces a pattern P, $\xi\xi\delta\delta\delta\psi$. This is a deterministic grammar, producing only pattern P.

Minimum Context Free Grammars We define the size of a grammar G as the number of symbols in the right hand side rules. Let CF_P be the size of the smallest CFG that produces the singleton language $|P|$.

Restricted Context-Free Grammars (RCFG). In this work, we focus on the CFG class used in [13] which they name Restricted CFGs. These restricted grammars produce a singleton language, $|L(G)| = 1$ and thus are deterministic. This is the same concept of Context-Free Straight Line grammars from [4]. Each RCFG production rule R contains two symbols on its right-hand side. We can convert any other deterministic CFG to this form with only a constant factor size increase.

Figure 7 presents an example RCFG G and its parse tree that derives a pattern of symbols P, $\xi\xi\delta\delta\delta\psi$. The parse tree shows how internal nodes are non-terminal symbols and leaf nodes contain a terminal symbol whose in-order traversal derives the output string. Notice that since RCFG G is deterministic, each non-terminal symbol $N \in V$ has a unique subpattern $g(N)$ that is defined by taking N to be the start symbol S and applying the production rules. Here, the language or output pattern P of G can be denoted by $L(G) = g(S)$.

4.2 1D Patterned Assembly Construction

We describe our method of simulating a Restricted CFG G with Tile Automata to build a 1D patterned assembly that represents the pattern P derived from G.

Initial Tiles and Producibles. This Tile Automata system, Γ_G, begins with creating its initial tiles from the unique terminal symbols, Υ, in RCFG G. In Figure 7, the output pattern P derived from G has three unique terminal symbols ξ, δ, and ψ. Each unique Υ in G is mapped to a distinct color and remains locked to the symbol throughout the construction. From G's production rule parse tree, internal nodes have two child nodes consisting of two similar or different terminal symbols, Υ. Depending on the placement of the terminal symbols, the initial tiles are designated as L for left-hand side or R for right-hand side. Figure 8a depicts that an initial tile consists of an Υ symbol with its distinct color in an L or R state.

Following G's parse tree, the initial tiles can combine to build Γ_G's first set of producible assemblies. Grammar G's production rules can be encoded into system Γ_G by providing the affinity rules. If two terminal symbols in G connect to the same internal node in its parse tree, the initial tiles in Γ_G that represent the symbols combine to form a producible. The first set of producibles cannot bind to any other tile because

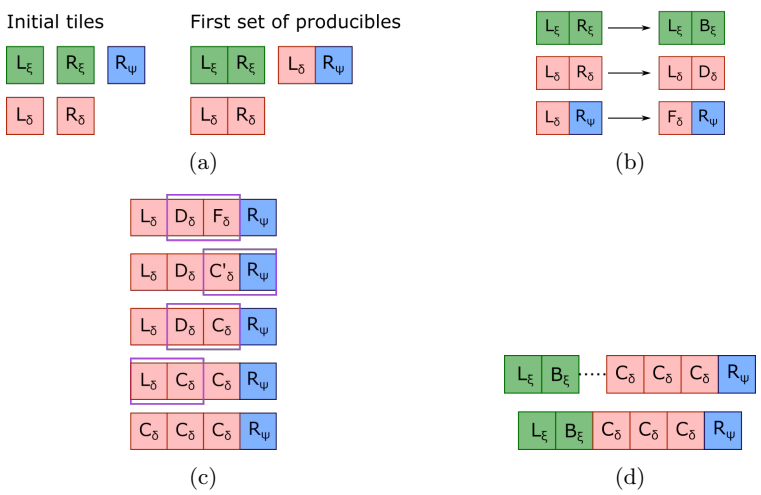


Fig. 8: Tile Automata system, Γ_G, assembling a 1D patterned assembly that represents the pattern P produced by the RCFG G shown in Figure 7. (a) Γ_G contains initial tiles from the unique terminal symbols of G. Grammar G's production rules are encoded in Γ_G as affinity rules, allowing initial tiles to form the first set of producibles. (b) Following G's rules, Γ_G's color-locked, one-sided transition rules are applied to the first set of producibles. (c) Subpattern assembly $L_\delta D_\delta F_\delta R_\psi$ transitions tiles towards captile R, marking visited tiles. Once the transitions reach captile R, we transition to the left of the subassembly to C_δ tiles, removing the marks along the way. (d) RCFG G production rule $Y \to BC$, directs Γ_G to combine B and C subassemblies to build the terminal patterned line assembly, representing pattern P from grammar G.

they are capped with L and R states, which we denote as *captiles*, and thus are stopped from growing, shown in Figure 8b. Note that these first producibles are subpatterns of P.

Uncapping Producibles. RCFG G production rules tell Γ_G how the first producible assemblies will combine to form larger subpatterns of P and ultimately represent the final patterned line assembly. In Γ_G, our first set of producibles are composed of L and R captiles. For these producibles to combine with each other, we apply one-sided, color-locked transition rules to uncap each producible, opening their left or right-hand side depending on the nonterminal symbols placement in grammar G's production rules. For example, in Figure 7 nonterminal C is composed of a D on the left-hand side and F on the right-hand side. In Figure 3.2b, the producible $L_\delta R_\psi$ represents G's terminal symbols $\delta\psi$ as well as nonterminal F. Because F sticks to D's right side, a one-sided transition rule is applied to producible $L_\delta R_\psi$ changing only the pink tile L_δ to a new tile F_δ, forming next producible $F_\delta R_\psi$. Here, the color-locked restriction in Γ_G applies because the new tile F_δ retains its color (pink) that is designated to the terminal symbol δ of P from G. This producible $F_\delta R_\psi$ is considered a right-handed subassembly because it is uncapped on its left side, allowing it to attach to

the right-hand side of the producible that represents nonterminal D. The rest of Γ_G's first producibles transition according to G's production rules as shown in Figure 8b.

Transition Walk. Γ_G recursively applies G's production rules to build the other subassemblies needed to represent pattern P. Grammar G's production rule $C \rightarrow DF$ tells Γ_G that there is affinity between D and F, directing producibles $L_\delta D_\delta$ and $F_\delta R$ to combine and form a new subpattern assembly $\delta\delta\delta\psi$ of P, shown at the top of Figure 8c. In Lemma 1, we show how every nonterminal in G is represented as a subpattern assembly produced by Γ_G. Subpattern assembly $L_\delta D_\delta F_\delta R_\psi$, represents nonterminal C from G and is capped with captiles L and R. From G's production rules in Figure 7, nonterminal symbol Y is composed of B on the left-hand side and C on right-hand side. To uncap the left side of subpattern $L_\delta D_\delta F_\delta R_\psi$, a series of one-sided, color-locked transition rules are applied to turn each tile into a C_δ tile making the subassembly uniform, depicted in Figure 8c. The adjacent tiles that have transition rules between them are outlined in purple, with the resulting tiles shown in the subassembly below it.

We apply the method of "walking" across 1D assemblies from [5] to uncap left or right sides of subassemblies. Subpattern assembly $L_\delta D_\delta F_\delta R_\psi$ must have an opened left side to attach to subassembly B, so we first transition tiles towards the right side, marking visited tiles with a prime notation. Once the transitions reach captile R, we begin to transition to the left of the subassembly to C_δ tiles, removing the prime notations along the way. As shown in Figure 8c, once producibles D and F combine, a one-sided, color-locked transition rule applies changing the F_δ tile for a temporary C'_δ tile, where the prime marks the tile as visited. Next, the adjacent C'_δ and R_ψ tiles transition to remove the prime from the C'_δ tile, producing subpattern $L_\delta D_\delta C_\delta R_\psi$. Another transition is applied between adjacent tiles $D_\delta C_\delta$ to form the fourth subassembly in Figure 8c. Finally, one more transition occurs between $L_\delta C_\delta$ to produce subpattern $C_\delta C_\delta C_\delta R_\psi$.

Patterned Line Assembly. Figure 8d depicts the subpattern assemblies created by Γ_G that represent nonterminal symbols B and C. According to the affinity rules of Γ_G, subassemblies B and C combine to form terminal assembly Y. Subassemblies for B and C attach and terminal assembly Y is constructed and capped with captiles L and R on its sides. This new terminal assembly Y represents G's pattern P, with each distinct colored tile representing unique terminal symbols of pattern P.

Definition 5 (Nonterminal Pattern)**.** *For a nonterminal $N \in V$, let $g(N)$ be a substring derived when N is the start symbol of grammar G.*

Lemma 1. *Each producible assembly in Γ_G, created from a RCFG $G = (V, \Upsilon, R, S)$ represents a subpattern $g(N)$ for some symbol N in $V \bigcup \Upsilon$.*

Proof. We will prove by induction that any producible assembly B represents a subpattern $g(N)$ for some symbol N in $V \bigcup \Upsilon$.

For the base case, if B is an initial tile, then B represents some terminal symbol $N \in \Upsilon$. For the inductive step, if B is a larger assembly, then we show B represents a nonterminal $N \in V$. We define the following two recursive cases. B is built from combining subassemblies C and D, we can assume these assemblies represent symbols N_C and N_D respectively. We know from how we defined our affinity rules if C and D can combine then there is some rule $N \rightarrow N_C N_D$. Then B represents the pattern $g(N) =$

$g(N_C) \oplus g(N_D)$. B is producible via transition from an assembly C, B must represent the same subpattern as C since the transition rules do not change the color. □

Theorem 2. *For any pattern P, there exists a Freezing Tile Automata system Γ with deterministic single transition rules that uniquely assembles P with $\mathcal{O}(CF_P)$ states and 1×1 scale. This system is cycle-free and transition rules do not change the color of tiles.*

Proof. By definition, there exists a CFG G that produces P with $|G| = CF_P$. We construct the system Γ_G. From Lemma 1, each producible assembly B must represent a subpattern $g(N)$ for some symbol N. The only terminal of Γ is the assembly representing the start symbol S since all other assemblies either can attach to another assembly or can transition. □

5 Optimal Patterns in Tile Automata

In this section we show that general Tile Automata can obtain Kolmogorov optimal state complexity at 1×1 scale. These first results are achieved by applying the efficient binary string construction from [1], and allowing the additional tiles used by the assembly to fall off, thus leaving only the string. We can then utilize the Turing machine from to simulate a universal Turing Machine. The Turing Machine in was designed to accept/reject an input, so we modify the Turing Machine to print P on the tape and halt.

Lemma 2. *For any binary pattern X there exists an affinity strengthening Tile Automata system that uniquely constructs an assembly representing X at scale,*

- 4×2 *with* $\mathcal{O}(|X|^{\frac{1}{4}})$ *states,*
- 3×2 *with* $\mathcal{O}(|X|^{\frac{1}{3}})$ *states using single-transition rules, and*
- 2×1 *with* $\mathcal{O}(|X|^{\frac{1}{2}})$ *states using deterministic single-transition rules and is cycle free.*

Proof. These constructions are provided in [1] which shows that there exists a method to encode the bits of a string in the transition rules of the system. Each construction takes advantage of a feature not available in the stricter class of systems. The model shown in this paper however does have seeded growth but a simple extension shows this works with 2-handed production. □

Theorem 3. *For any pattern P, there exists a Tile Automata system Γ that uniquely assembles P with $\Theta(K_P^{\frac{1}{4}})$ states at 1×1 scale.*

Proof. Given a pattern P, we first consider a Turing machine M that will print P. Using the process described in [5], we create a system $\Gamma_M = (\Sigma, \Pi, \Lambda, \Delta, \tau)$ that simulates M. When M has completed printing P, the buffer states B_L and B_R need to detach. We take Σ and create a copy Σ_{SR} which we modify by removing the accept/reject states in favor of *final states*. For every state $\rho \in \Sigma_{SR}$ where ρ composes P, we create $\rho_F \in \Sigma_{SR}$ with affinity only for every other final state. Starting with the rightmost tile that composes P, we add transition rules that will transition each tile with state ρ into their final state equivalent ρ_F. Since these final states have no affinity with the buffer states, tiles with those buffer states, and any other state not

considered a final state, will detach from the assembly. This detaching process begins with a transition rule between B_R and the rightmost tile with state ρ, turning ρ into ρ_F.

From Lemma 2, we encode Γ_M in a binary string $b(\Gamma_M)$ and use $b(\Gamma_M)$ to construct system Γ_S that uses $\Theta(K_P^{\frac{1}{4}})$ to assemble $b(\Gamma_M)$. [21] states there exists a universal Turing machine that uses linear space in the amount of space used by the machine being simulated. Γ will simulate a universal Turing machine with Γ_S being used to construct the input into Γ, giving us a system that uniquely assembles P with $\Theta(K_P^{\frac{1}{4}})$ states and 1×1 scale. □

5.1 Deterministic Single Transition Turing Machine

The Turing machine from [5] utilizes transition rules that change both tiles in the same step. While [8] shows a way to simulate double rules with single rules, we present a slight modification to the Turing machine construction to make it utilize single rules.

Lemma 3. *For any pattern P, there exists a Tile Automata system Γ with deterministic single-transition rules that uniquely assembles P with $\mathcal{O}(K_P)$ states and 1×1 scale. This system is cycle free.*

Proof. We create a Turing machine M that will print P. Using Turing machine M, we use the process described in [5] to create a system $\Gamma_D = (\Sigma, \Pi, \Lambda, \Delta, \tau)$ that simulates M utilizing double-transition rules. We then modify Σ, Δ, and Π into single-transition rule versions Σ_{SR}, Δ_{SR}, and Π_{SR} as follows.

Σ_{SR} and Π_{SR} will initially be a copy of Σ and Π respectively, while Δ_{SR} is populated with every single-transition rule in Δ. For every double-transition rule $\delta = (\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, d) \in \Delta$, we create an additional state $\omega \in \Sigma_{SR}$. The affinity strength of ω using Π_{SR} will be equal to the affinity strength of D using Π for all directions. We take δ and create 3 transition rules $\delta_{S1}, \delta_{S2}, \delta_{S3} \in \Delta_S$ defined below.

- $\delta_{S1} = (\mathsf{A}, \mathsf{B}, \mathsf{A}, \omega, d)$
- $\delta_{S2} = (\mathsf{A}, \omega, \mathsf{C}, \omega, d)$
- $\delta_{S3} = (\mathsf{C}, \omega, \mathsf{C}, \mathsf{D}, d)$

We use the *final states* described in the proof of Theorem 3 to modify Σ_{SR} in order to detach the buffer states. Using our modifications, we create a Tile Automata system $\Gamma = (\Sigma_{SR}, \Pi_{SR}, \Lambda, \Delta_{SR}, \tau)$ with deterministic single-transition rules that uniquely assembles P with $\mathcal{O}(K_P)$ states and 1×1 scale. □

Using Lemma 2we can encode the input to a universal Turing machine with square root the number of states with deterministic single transition rules.

Theorem 4. *For any pattern P, there exists a Tile Automata system Γ with deterministic single transition rules that uniquely assembles P with $\mathcal{O}(K_P^{\frac{1}{2}})$ states and 1×1 scale. This system is cycle free.*

Proof. We make some modifications to the process used in the proof of Theorem 3 to satisfy the deterministic single-transition rules. We create Γ_M using the method described in the proof of Lemma 3 and encode the system in a binary string $b(\Gamma_M)$. Γ_S is created using $b(\Gamma_M)$ which will use $\mathcal{O}(K_P^{\frac{1}{2}})$ as shown in Lemma 2. Γ will simulate a

universal Turing machine that uses the assembly built by Γ_S, giving us a system that uniquely assembles P with $\mathcal{O}(K_P^{\frac{1}{2}})$ states and 1×1 scale. □

Other methods for non-deterministic rules and with single and double rules give the following.

Theorem 5. *For any pattern P, there exists a Tile Automata system Γ with single transition rules that uniquely assembles P with $\Theta(K_P^{\frac{1}{3}})$ states and 1×1 scale.*

Proof. A deterministic single-rule TA system Γ_M can be constructed according to Lemma 3, and using an encoding $b(\Gamma_M)$, we make Γ_S which uses $\Theta(K_P^{\frac{1}{3}})$ states using Lemma 2 □

5.2 Freezing with Detachment

We do not directly consider Freezing and allowing detachment since the results of [9] shown that any non-freezing system can be simulated by a freezing system by replacing tiles. Also shown in the full version of [5] it was shown freezing Tile Automata with only height 2 assemblies can simulate a general Turing machine. The assembly can then fall apart to achieve 1×1 scale.

6 Affinity Strengthening

As shown in [5], Affinity Strengthening Tile Automata (ASTA) is capable of simulating Linear Bounded Automata (LBA) and that verification in ASTA is PSPACE-Complete. Thus, it makes sense to view this version of the model as the spaced-bounded version of Tile Automata, similar in power to LBAs or Context Sensitive Grammars. We select space-bounded Kolmogorov complexity as our method of bounding the state complexity since we can encode a string and simulate a Turing machine as in the previous section to get an upper bound. The concept of bounded Kolmogorov Complexity was explored in [14]. For these results, we consider building scaled patterns in which each pixel of the pattern is expanded to a $s \times O(1)$ box of pixels. Another way to view this upper bound is that for any algorithm α that outputs P in $f(|P|)$ space, we may construct an assembly representing P of size $\mathcal{O}(f(|P|)$, in $\mathcal{O}(|\alpha|)^{\frac{1}{4}}$ states, where $|\alpha|$ is the number of bits describing α for general Tile Automata. Similar bounds are shown for the other restrictions. It is interesting to point out that with a large enough scale factor we achieve Kolmogorov optimal bounds, including optimal scaled shape constructions as in [16].

6.1 Space Bounded Kolmogorov Complexity

Definition 6 (Space Bounded Kolmogorov Complexity)**.** *Given a pattern P, and a function $f : \mathbb{N} \to \mathbb{N}$ that outputs the space used by a Turing machine, let $KS_P(f(|P|))$ be the length of the smallest string that, when input to a universal Turing machine M_K, halts with the pattern P on the tape in $f(|P|)$ space.*

It was stated in [14] that there exists some optimal Turing machine, which we call M_K, that incurs only a constant multiplicative factor increase in the space used. We note for two space bounds $f(|P|)$ and $g(|P|)$, the value $KS_P(g(|P|)) \leq KS_P(f(|P|))$

(a) Assembly representing input string.

(b) Universal Turing machine running on input.

(c) The pattern is output on the tape.

(d) The pattern scales outward to fill the assembly.

Fig. 9: (a) It is possible to build assemblies representing binary strings with an efficient number of states. (b) We can then run a universal Turing machine on the input increasing the length of the assembly as needed. (c) The Turing machine will halt with the pattern output on the tape. (d) The pattern will then scale out to fill the assembly.

as using more space allows for more efficient computing of all pattern P, with $|P| < c$ for some constant c.

6.2 Construction

Figure 9a shows a sketch of the assembly for deterministic Tile Automata using the string constructions from [1] shown in Lemma 2. The single rule Turing machine can be modified to never break apart and only increase the tape length, similar to the PSPACE-hard reduction from [5]. Figure 9b shows an example Turing machine being run where the tape length is increased.

Once the pattern has been printed or assembled (Figure 9c), there are additional tiles in the assembly to deal with. However, since we cannot detach tiles, we scale the pattern. The first step is to expand the length of pattern. If we use s tape cells to print a pattern $|P|$, we scale each point in the pattern by $c \cdot |P|$. This is done with a simple algorithm implemented in the transition rules. Create a token state that starts at the leftmost state after the string is printed. Go to each 'pixel' and tell it expand once after first signaling the neighboring cells to move right (to prevent overwriting). We do this for each pixel in the pattern, push the other states, increase pixel size. The system repeats this process until all pixels of the pattern are fully expanded, and then they transition the tiles below them, which results in the patterned assembly of Figure 9d.

Theorem 6. *For any pattern P, scale factor $s > 0$, there exists an Affinity Strengthening Tile Automata system Γ with deterministic single transition rules that uniquely assembles P with $\mathcal{O}(KS_P(s|P|)^{\frac{1}{2}})$ states and $s \times 2$ scale. This system is cycle free.*

Proof. Let X be the string that when input to M, P is written to the tape in $s|P|$ space. Using the binary string building results from Lemma 2 we can encode X in $\mathcal{O}(|X|^{\frac{1}{2}})$ states. Then we run M using the single transition rule Turing machine described in the proof of Lemma 3. This will run and leave the pattern P on the tape states. Consider a second Turing machine M_{INC} scales up the pattern to fill the width of the tape. Each pixel is increased by the same amount. The states then copy the color to the state below it as well. This can be done in a constant number of states. The amount that each pattern scales by is $\frac{s|P|}{|P|} = |P| \cdot (s-1)$. □

Theorem 7. *For any pattern P, scale factor $s > 0$, there exists an Affinity Strengthening Tile Automata system Γ with single-transition rules that uniquely assembles P with $\mathcal{O}(KS_P(s|P|)^{\frac{1}{3}})$ states and $s \times 3$ scale.*

Proof. Again using the Single-Transition rule Turing machine from the proof of Lemma 3 and the string building result from Lemma 2, we can construct the input to the universal Turing machine M_K. The pattern P can be output in $s|P|$ space. We then scale up the pattern to fill the assembly. □

Theorem 8. *For any pattern P, scale factor $s > 0$, there exists an Affinity Strengthening Tile Automata system Γ that uniquely assembles P with $\mathcal{O}(KS_P(s|P|)^{\frac{1}{4}})$ states and $s \times 4$ scale.*

Proof. Lastly using the same method from Lemma 2 we can encode the input to the universal Turing machine in $|X|^{\frac{1}{4}}$ where $|X|$ is the length of the string. This results in an assembly of height 4 as resulting assembly will be of dimensions $|X| \times 4$. The string X can then be input to the Turing machine to print the pattern than scale up. □

7 Lower Bounds

We provide lower bounds for general Tile Automata under the three transition rule restrictions. We do this by showing a binary string encoding a Tile Automata system can be passed to a Turing machine to output a patterned assembly, from which the pattern P can be read and output. This means we cannot encode a system in less bits than the Kolmogorov Complexity K_P. We achieve similar bounds as [1] as we use the same system for binary string encoding.

For affinity strengthening we provide a lower bound based on the Space Bounded Kolmogorov Complexity defined in Section 6. As with the previous result, we show that a binary string encoding a system can be passed to a Turing machine that outputs the uniquely produced assembly representing the pattern P in $f(|P|)$ space. This means we cannot encode the system in less than $KS_P(f(|P|))$ bits. We give an upper bound of $f(n) = \mathcal{O}((s|P|)^2 \log^2 s|P|)$ in Lemma 4 to compute a pattern scaled by a factor of s. With this we base our lower bounds on $KS_P((s|P|)^2 \log^2 s|P|)$.

7.1 General

Theorem 9. *For any Pattern P over constant colors a Tile Automata system Γ that uniquely assembles P at any scale requires $\Omega(K_P^{\frac{1}{4}})$ states.*

Proof. A Tile Automata system $\Gamma = \{\Sigma, \Pi, \Lambda, \Delta, \tau = \mathcal{O}(1)\}$ can be encoded in $< c|\Sigma|^4$ bits for some constant c. We may store Π as a $|\Sigma| \times |\Sigma|$ table with each $\mathcal{O}(\log \tau)$ bit cell storing their binding strength which is at most τ. The initial tiles Λ can be encoded with a single bit for each state. Δ is the largest part of the encoding taking $2|\Sigma|^4$ bits . This can be stored as a 4D table where each cell contains two bits (v, h). The first bit at index $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ being whether or not the states (σ_1, σ_2) transition to (σ_3, σ_4) vertically and the second bit horizontally. The exact constant achieved is thus dependent on τ.

Consider a Turing machine M_{TA} that takes as input the binary description of a Tile Automata system Γ that uniquely assembles an assembly A and outputs the pattern of A as a string. We can assume M_{TA} can be described in constant bits. The producible assemblies of a Tile Automata system are recursively enumerable. Since we know that Γ uniquely produces P we know there exists a finite number of assemblies as well as the system must be bounded. This makes verifying the terminal assembly is decidable as there's only a finite number of possible Combinations, Breaks, and Transitions to check.

Let M_K be the fixed universal Turing machine to define K_P, assume there exists a system $\Gamma' = \{\Sigma', \Pi', \Lambda', \Delta', \tau' = \mathcal{O}(1)\}$ that uniquely produces the pattern P with $|\Sigma'| < (\frac{K_P}{c})^{\frac{1}{4}}$ states. Using our encoding method above encode Γ' as a binary string $b(\Gamma')$ with in $|b(\Gamma')| < K_P$. If we pass $b(\Gamma')$ along with an encoding of M_{TA} to the universal Turing Machine M_K it will simulate the algorithm and output the pattern P. This would mean that M_K can produce the pattern with less than $K_P + | < M_k > |$ bits which violates the Kolmogorov Complexity so this is not possible. □

Theorem 10. *For any Pattern P over constant colors, a Tile Automata system Γ with single transition rules that uniquely assembles P at any scale requires $\Omega(K_P^{\frac{1}{3}})$ states.*

Proof. We use the same argument for this proof but show the system can be encoded more efficiently. We can store our transition rules in a $\mathcal{O}(|\Sigma|^3)$ bit table. This is a 3D table where each cells stores 4 bits. The first two indices representing the starting states and the third is the target state. There is only one state since single transition rules only change one rule at a time. The table stores 4 bits in order to store whether they transitions vertically or horizontally, and whether the first or second tile changes to the other state. □

Theorem 11. *For any Pattern P over constant colors, a Tile Automata system Γ with deterministic transition rules that uniquely assembles P at any scale requires $\Omega\left((\frac{K_P}{\log K_P})^{\frac{1}{2}}\right)$ states.*

Proof. Deterministic rules can be encoded in $\mathcal{O}(|\Sigma|^2 \log |\Sigma|)$ bits. To achieve this, store the rules in a $|\Sigma| \times |\Sigma|$ table where each cell stores up to two other pairs of states which takes $\mathcal{O}(\log |\Sigma|)$ bits. We only need to store a constant number of pairs since each pair of states and orientation can only have a single rule. Note that this method can encode single or double transition rules with only a constant factor difference. Applying similar algebra as done for Theorem 9 we have $|\Sigma| = \Omega\left((\frac{K_P}{\log K_P})^{\frac{1}{2}}\right)$. □

7.2 Affinity Strengthening

In [5] it was shown that the Unique Assembly Verification Problem (UAV) for affinity strengthening Tile Automata is solvable in PSPACE. In Lemma we show that given a binary string $b(|\Gamma|)$ describing a directed Affinity Strengthening Tile Automata system, we can produce a description of the uniquely produced assembly A in $\mathcal{O}(|A|^2 \log^2 |\Sigma|)$ space. We then apply this fact in Theorem 12 to get a state complexity lower bound based on bounded-space Kolmogorov complexity.

Lemma 4. *Given a binary string $b(\Gamma)$ describing a directed Tile Automata system Γ, there exists an algorithm that outputs the uniquely produced assembly $\texttt{TERM}_\Gamma = \{A\}$, in $\mathcal{O}(|A|^2 \log^2 |\Sigma|)$ space.*

Proof. This can be done by making multiple calls to a subroutine that solves the unique assembly verification problem (UAV) for affinity strengthening Tile Automata. For each integer starting at $i = 1$, call the algorithm for UAV on each assembly B of size $|B| = i$. If the UAV algorithm returns yes, then return B since $A = B$, i.e. B is the uniquely produced assembly. Storing one of these assembly take $\mathcal{O}(|A| \log |\Sigma|)$ bits and we only need to have stored one at a time and the largest assembly we store is $|A|$ size.

The exact details of the algorithm are shown in [5]. This algorithm only stores a constant number of assemblies at a time each of up to size $2|A|$. We can store an assembly in $|A| \log |\Sigma|$ bits thus giving our bound. □

Theorem 12. *For all Patterns P, scale factor $s > 0$, an Affinity Strengthening Tile Automata system Γ that uniquely assembles P at scale $n \times m$ for $nm = s$ requires $\Omega(KS_P((s|P|)^2 \log^2 s|P|)^{\frac{1}{4}})$ states.*

Proof. We can use the same method for encoding Γ into a binary string $b(\Gamma)$ as done in Theorem 9 to achieve $|b(\Gamma)| = \mathcal{O}(|\Sigma|^4)$. We can pass $b(\Gamma)$ along with an algorithm that outputs the pattern P produced by Γ to the universal Turing Machine M_K. With this we can bound the length of the string, $|b(\Gamma)| \geq KS_P(f(|P|))$ where $f(|P|)$ is the space taken by the algorithm to output P.

From Lemma 4 we know we can output a description of the uniquely produced assembly A in $\mathcal{O}(|A|^2 \log^2 |\Sigma|)$ space and the pattern can be read and output. A naive implementation can give $|\Sigma| \leq |A|$ by assigning each tile a unique state. The size of the assembly is $|A| = s|P|$, so we can bound the space by the scale factor s and the pattern size $|P|$ giving us $\mathcal{O}((s|P|)^2 \log^2 s|P|)$. We therefore get $|\Sigma| = \Omega\left(KS_P((s|P|)^2 \log^2 s|P|)^{\frac{1}{4}}\right)$. □

Theorem 13. *For all Patterns P, scale factor $s > 0$, an Affinity Strengthening Tile Automata system Γ with single transition rules that uniquely assembles P at scale $n \times m$ for $nm = s$, requires $\Omega(KS_M(P, s|P|^3)^{\frac{1}{3}})$ states.*

Proof. We may encode a system with single transition rules in $|\Sigma|^3$ bits so we get a bound of $|\Sigma| = \Omega\left(KS_P((s|P|)^2 \log^2 s|P|)^{\frac{1}{3}}\right)$. □

Theorem 14. *For all Patterns P, scale factor $s > 0$, an Affinity Strengthening Tile Automata system Γ with deterministic transition rules that uniquely assembles P at scale $n \times m$ for $nm = s$, requires $\Omega\left((\frac{KS(P,|P|^3)}{\log KS_M(P,|P|^3)})^{\frac{1}{2}}\right)$ states.*

Proof. A deterministic Tile Automata system can be encoded $\mathcal{O}(|\Sigma|^2 \log |\Sigma|)$ bits. By performing the same steps as in Theorem 11 we get $|\Sigma| = \Omega\left((\frac{KS(P,|P|^3)}{\log KS_M(P,|P|^3)})^{\frac{1}{2}}\right)$. □

8 Conclusion

In this paper we show how to convert any 1D staged assembly system to an equivalent 1D freezing Tile Automata system. We then show how this generalizes some previous results. We then show how a similar techinque can be used to implement CFGs to build patterns. We then described a set of upper and lower bounds for pattern building based on previous work. There are many interesting directions for future work.

- What is the most efficient method to compute the glue-terminal table?
- Can we improve the number of states needed in the TA simulation? Could it be reduced to $\mathcal{O}(st + bt)$ or even $\mathcal{O}(sg + bg)$ where g is the number of glues in the system? What is the lower bound?
- Does allowing for 1D scaling help achieve better bounds?
- Can 1D staged simulate 1D freezing Affinity-Strengthening Tile Automata? I.e., are they equivalent? If so, how many tiles, bins, and stages are needed?
- What challenges arise when attempting to generalize this to 2D? The glue-terminal table must not only store whether or not an assembly is terminal based on its glues, but also its geometry.
- What is the lower bound for building patterns in 1D freezing Affinity-Strengthening Tile Automata? Are there languages that Tile Automata can assemble more efficiently than staged?

Declarations

Ethical Approval

Not applicable.

Competing interests

There are no competing interests that we are aware of in reference to this paper.

Authors' contributions

These authors contributed equally to this work.

Funding

No external funding was received.

Availability of data and materials

Data Availability Statement: No Data associated in the manuscript.

References

[1] Alaniz RM, Caballero D, Cirlos SC, et al (2022) Building squares with optimal state complexity in restricted active self-assembly. In: Proc. of the Symposium on Algorithmic Foundations of Dynamic Networks, pp 6:1–6:18

[2] Alumbaugh JC, Daymude JJ, Demaine ED, et al (2019) Simulation of programmable matter systems using active tile-based self-assembly. In: DNA Computing and Molecular Programming, Cham, DNA'19, pp 140–158

[3] Barad G, Amarioarei A, Paun M, et al (2019) Simulation of one dimensional staged dna tile assembly by the signal-passing hierarchical tam. Procedia Computer Science 159:1918–1927

[4] Benz F, Kötzing T (2013) An effective heuristic for the smallest grammar problem. In: Proc. of the 15th Annual Conf. on Genetic and Evolutionary computation, pp 487–494

[5] Caballero D, Gomez T, Schweller R, et al (2020) Verification and Computation in Restricted Tile Automata. In: 26th Inter. Conf. on DNA Computing and Molecular Programming, pp 10:1–10:18

[6] Caballero D, Gomez T, Schweller R, et al (2021) Covert computation in staged self-assembly: Verification is pspace-complete. In: 29th Annual European Symposium on Algorithms, ESA'21, pp 23:1–23:18

[7] Cannon S, Demaine ED, Demaine ML, et al (2013) Two Hands Are Better Than One (up to constant factors): Self-Assembly In The 2HAM vs. aTAM. In: 30th Inter. Sym. on Theoretical Aspects of Computer Science, pp 172–184

[8] Cantu AA, Luchsinger A, Schweller R, et al (2020) Signal passing self-assembly simulates tile automata. In: 31st Inter. Sym. on Algorithms and Computation, ISAAC'20, pp 53:1–53:17

[9] Chalk C, Luchsinger A, Martinez E, et al (2018) Freezing simulates non-freezing tile automata. In: DNA Computing and Molecular Programming, Cham, pp 155–172

[10] Chalk C, Martinez E, Schweller R, et al (2018) Optimal staged self-assembly of general shapes. Algorithmica 80(4):1383–1409

[11] Chalk C, Martinez E, Schweller R, et al (2019) Optimal staged self-assembly of linear assemblies. Natural Computing 18(3):527–548

[12] Demaine ED, Demaine ML, Fekete SP, et al (2008) Staged self-assembly: nanomanufacture of arbitrary shapes with o (1) glues. Natural Computing 7(3):347–370

[13] Demaine ED, Eisenstat S, Ishaque M, et al (2011) One-dimensional staged self-assembly. In: Proceedings of the 17th international conference on DNA computing and molecular programming, DNA'11, pp 100–114

[14] Longpré L (1986) Resource bounded kolmogorov complexity, a link between computational complexity and information theory. Tech. rep., Cornell University

[15] Schweller R, Winslow A, Wylie T (2019) Verification in staged tile self-assembly. Natural Computing 18(1):107–117

[16] Soloveichik D, Winfree E (2007) Complexity of self-assembled shapes. SIAM Journal on Computing 36(6):1544–1569

[17] Thubagere AJ, Li W, Johnson RF, et al (2017) A cargo-sorting DNA robot. Science 357(6356):eaan6558

[18] Tikhomirov G, Petersen P, Qian L (2017) Fractal assembly of micrometre-scale dna origami arrays with arbitrary patterns. Nature 552(7683):67–71

[19] Winfree E (1998) Algorithmic self-assembly of DNA. PhD thesis, California Institute of Technology

[20] Winslow A (2015) Staged self-assembly and polyomino context-free grammars. Natural Computing 14(2):293–302

[21] Woods D, Neary T (2009) The complexity of small universal turing machines: A survey. Theoretical Computer Science 410(4-5):443–450

[22] Woods D, Doty D, Myhrvold C, et al (2019) Diverse and robust molecular algorithms using reprogrammable dna self-assembly. Nature 567(7748):366–372

APPENDIX C

APPENDIX C

COVERT COMPUTATION IN THE ABSTRACT TILE SELF-ASSEMBLY MODEL

Covert Computation in the Abstract Tile-Assembly Model

Robert M. Alaniz
Department of Computer Science, University of Texas Rio Grande Valley

David Caballero
Department of Computer Science, University of Texas Rio Grande Valley

Timothy Gomez
Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology

Elise Grizzell
Department of Computer Science, University of Texas Rio Grande Valley

Andrew Rodriguez
Department of Computer Science, University of Texas Rio Grande Valley

Robert Schweller
Department of Computer Science, University of Texas Rio Grande Valley

Tim Wylie
Department of Computer Science, University of Texas Rio Grande Valley

Abstract

There have been many advances in molecular computation that offer benefits such as targeted drug delivery, nanoscale mapping, and improved classification of nanoscale organisms. This power led to recent work exploring privacy in the computation, specifically, covert computation in self-assembling circuits. Here, we prove several important results related to the concept of a hidden computation in the most well-known model of self-assembly, the Abstract Tile-Assembly Model (aTAM). We show that in 2D, surprisingly, the model is capable of covert computation, but only with an exponential-sized assembly. We also show that the model is capable of covert computation with polynomial-sized assemblies with only one step in the third dimension (just-barely 3D). Finally, we investigate types of functions that can be covertly computed as members of P/Poly.

2012 ACM Subject Classification Theory of computation → Computational complexity and cryptography

Keywords and phrases self-assembly, covert computation, atam

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding This research was supported in part by National Science Foundation Grant CCF-1817602.

Acknowledgements This research was supported in part by National Science Foundation Grant CCF-1817602.

1 Introduction

With the ability to manufacture nanoscale structures and to use DNA as building blocks for structures [28] or for data storage [10], there has been a great increase in the need to process and compute information at the same level. Thus, the study of self-assembling computation has been an important and active area of research over the last two decades.

Designing self-assembling systems that compute functions is an active and well-studied area of computational geometry and biology [4,19]. This ability to craft monomers capable of placing themselves — especially when doing precision construction and computation at scales where conventional tools are incapable of operating, e.g., the nanoscale — has tremendous

power. One of the few downsides to self-assembly computation is that the entire history of the computation is visible. In certain cases, this may be undesirable for privacy or security reasons, which we motivate below. Thus, we build on recent work [6, 7, 9] to explore *covert computation*, where we build Tile Assembly Computers (TACs) designed with the goal of obtaining the output of computation while obscuring the inputs and computational history. We do this by proving that covert computation is possible even in one of the simplest standard models of self-assembly: the Abstract Tile-Assembly Model (aTAM) [29].

Motivation. The development of covert computation as a model and method of designing self-assembling systems was driven by several areas of concern in cryptography, biomedical engineering, privacy, and might even help protect intellectual property in systems that use "products of nature," such as DNA, as they cannot be patented in the United States as of 2013 [14]. Covert computation has also emerged as a powerful complexity tool, being used to show the coNP-completeness of the *Unique Assembly Verification* problem in the negative glue aTAM [9], and the PSPACE-completeness of the Unique Assembly Verification problem in the Staged Assembly Model [6]. As this paper focuses on systems without detachment, there might also be important applications in implantable systems where even the possibility of displacement from free-floating DNA could cause unknown side effects or destabilization of the assembly [23].

1.1 Previous Work

The Abstract Tile-Assembly Model (aTAM) was first introduced in [29] and inherited the ability to perform Turing computation from Wang tiles. Since then, investigation into the model has led in many directions, such as Intrinsic Universality [18, 22], efficient assembly of shapes [25], and parallel computation [5, 24]. Many generalizations have also appeared, such as allowing for RNA tiles that can be deleted [1, 15], multiple stages of growth [6, 12, 16], and even negative glues [9, 17]. The aTAM is powerful because not only can the tile set store information, but work has also gone into using the seed [3], or even the temperature [11, 26], for making systems more complex.

Tile Assembly Computers were defined in [5, 24], and Covert Computation, as defined in the field of self-assembly, was first introduced in 2019 [9] for negative growth-only aTAM. In *negative* variations of tile self-assembly models, tiles are capable of not only attachment to but also detachment from an assembly if the remaining assembly is still stable. In negative *growth-only* aTAM, tiles are never allowed to detach even though there may be glues providing a repellent force, and the system must be designed so that detachment does not occur. This paper introduced the covert construction framework to answer an open complexity problem for Unique Assembly Verification (UAV) with negative growth-only glues in the aTAM model, showing it to be coNP-complete. Notably, without negative glues, the UAV problem is solvable in polynomial time [2].

Covert computation has been explored in two other models of self-assembly as well: Staged Self-Assembly [6] and Tile Automata [7]. The staged self-assembly model, one of the most powerful *passive* tile self-assembly models, abstracts the process of scientists mixing test tubes together by allowing multiple self-assembly processes occurring in separate 'bins' that may be combined in subsequent 'stages.' The authors show that 3-stages suffice for covert computation and used the techniques to show that the UAV problem directly relates the number of stages to a specific level of the polynomial hierarchy. Thus, with no restrictions on the number of stages, UAV in the staged model is PSPACE-complete. Covert Computation in the *active* self-assembly model of Tile Automata was shown to be rather simple as tiles in the model are capable of changing states (instead of having static glues), easily erasing

Class	Model	Size Of				Ref
		Input	Tile Set	Output	Assembly	
Bool. Circuits	Neg_{GO}	$\mathcal{O}(n)$	$\mathcal{O}(MCS)$	$\mathcal{O}(k)$	$\mathcal{O}(MCS)$	[9]
Bool. Circuits	3D	$\mathcal{O}(n)$	$\mathcal{O}(MCS)$	$\mathcal{O}(k)$	$\mathcal{O}(MCS)$	Thm. 1
Rev. Circuits*	2D	$\mathcal{O}(n + MSC)$	$\mathcal{O}(MCS)$	$\mathcal{O}(1)$	$\mathcal{O}(2^n)$	Thm. 2

Table 1 Known Covert Circuits for n-bit function $f(x)$. Let MCS be the minimum circuit size that computes $f(x)$. **Input Size** is the size of the input assembly. **Output Size** is the size of the output template, where we use k to describe the number of output bits. * currently only works for binary functions.

computational history.

1.2 Our Contributions

In this work, we further explore the problem of designing covert tile assembly computers (TACs) in the aTAM, focusing on TACs that have a polynomial size description. We provide two new covert computers in the aTAM with only positive glue strengths of $\{1, 2\}$ in Sections 3 and 4. The 3D construction uses a similar technique to the circuits in [9] by implementing a NAND gate using dual rail logic and backfilling. We refer to this covert TAC as having a strict polynomial size since the systems defined by the TAC all produce assemblies of polynomial size. This only uses a single-step into the third dimension, which is occasionally referred to as *just-barely* 3D [20, 21].

The covert TAC in Section 4 is in the standard 2D aTAM. The TAC is of polynomial size, but produces an exponential-size terminal assembly. This works by computing the function non-covertly using Toffoli gates, getting the output, reversing the computation to recover the input, then building the next and previous circuit assemblies until all possible circuits are built. We utilize the Toffoli gates' reversibility property to have a symmetrical circuit assembly that displays its input on both sides that we can increment or decrement (the input used) to start the next computation.

In Section 5 we explore the classes of decision problems solvable by polynomial size covert TACs. Table 1 gives an overview of known covert circuits for functions based on the input size. Since covert has been defined as a non-uniform model, meaning different input sizes have different tile sets, we look at non-uniform complexity classes as well. Namely, the class P/poly, the class of problems solvable by polynomial size circuits. We prove that if a problem is solvable by a 3D covert TAC, then it is in P/poly. This, taken with the result in Section 3, shows an equivalence between these two models of computation.

2 Definitions

We begin with an overview of the Abstract Tile-Assembly Model, then follow with a definition of Tile Assembly Computers and covert computation.

2.1 Abstract Tile Assembly Model

At a high level, the Abstract Tile-Assembly Model (aTAM) uses a set of *tiles* capable of sticking together to construct shapes. These tiles are typically squares (2D) or cubes (3D) with *glues* on each side where they may attach to one another. A glue is labeled to indicate its type, governing what other tiles it may bond with and the *strength* of the bond. A tile

with all of its labels is a *tile type*. A *tile set* contains all the tile types of the system. A single tile may attach at a location if the combined strength of the matching glues is greater than or equal to the *temperature* τ. An *assembly* is a shape made up of one or more combined tiles. Construction is started around a designated *seed* assembly S. Any assembly capable of being made from the seed is called a *producible* assembly. An assembly is *terminal* if no more tiles can attach. A terminal assembly is said to be *uniquely produced* if it is the only terminal assembly that can be made by a tile system. A tile system is formally represented as an ordered triplet $\Gamma = (T, s, \tau)$ of the tile set, seed assembly, and temperature parameter, respectively.

2.1.1 aTAM Formal Definitions

Tiles. Let Π be an alphabet of symbols called the *glue types*. A tile is a finite edge polygon with some finite subset of border points, each assigned a glue type from Π. Each glue type $g \in \Pi$ also has some integer strength $str(g)$. Here, we consider unit square tiles of the same orientation with at most one glue type per face, and the *location* to be the center of the tile located at integer coordinates.

Assemblies. An assembly A is a finite set of tiles whose interiors do not overlap. If each tile in A is a translation of some tile in a set of tiles T, we say that A is an assembly over tile set T. For a given assembly A, define the *bond graph* G_A to be the weighted graph in which each element of A is a vertex, and the weight of an edge between two tiles is the strength of the overlapping matching glue points between the two tiles. Only overlapping glues of the same type contribute a non-zero weight, whereas overlapping, non-equal glues contribute zero weight to the bond graph. The property that only equal glue types interact with each other is referred to as the *diagonal glue function* property, and is perhaps more feasible than more general glue functions for experimental implementation (see [13] for the theoretical impact of relaxing this constraint). An assembly A is said to be τ*-stable* for an integer τ if the min-cut of G_A has weight at least τ.

Tile Attachment. Given a tile t, an integer τ, and an assembly A, we say that t may attach to A at temperature τ to form A' if there exists a translation t' of t such that $A' = A \cup \{t'\}$, and the sum of newly bonded glues between t' and A meets or exceeds τ. For a tile set T, we use notation $A \rightarrow_{T,\tau} A'$ to denote there exists some $t \in T$ that may attach to A to form A' at temperature τ. When T and τ are implied, we simply say $A \rightarrow A'$. Further, we say that $A \rightarrow^* A'$ if either $A = A'$, or there exists a finite sequence of assemblies $\langle A_1 \dots A_k \rangle$ such that $A \rightarrow A_1 \rightarrow \dots \rightarrow A_k \rightarrow A'$.

Tile Systems. A tile system $\Gamma = (T, S, \tau)$ is an ordered triplet consisting of a set of tiles T called the system's *tile set*, a τ-stable assembly S called the system's *seed* assembly, and a positive integer τ referred to as the system's *temperature*. A tile system $\Gamma = (T, S, \tau)$ has an associated set of *producible* assemblies, $\texttt{PROD}_\Gamma$, which define what assemblies can grow from the initial seed S by any sequence of temperature τ tile attachments from T. Formally, $S \in \texttt{PROD}_\Gamma$ is a base case producible assembly. Further, for every $A \in \texttt{PROD}_\Gamma$, if $A \rightarrow_{T,\tau} A'$, then $A' \in \texttt{PROD}_\Gamma$. That is, assembly S is producible, and for every producible assembly A, if A can grow into A', then A' is also producible.

We further denote a producible assembly A to be *terminal* if A has no attachable tile from T at temperature τ. We say a system $\Gamma = (T, S, \tau)$ *uniquely produces* an assembly A if all producible assemblies can grow into A through some sequence of tile attachments. More formally, Γ *uniquely produces* an assembly $A \in \texttt{PROD}_\Gamma$ if for every $A' \in \texttt{PROD}_\Gamma$ it is the case that $A' \rightarrow^* A$. Systems that uniquely produce one assembly are said to be *deterministic*.

2.2 Covert Computation

Here, we provide formal definitions for computing a function with a tile system and the further requirements for the covert computation of a function. Our formulation of computing functions is that used in [9], which is a modified version of the definition provided in [24] to allow for each bit to be represented by a subassembly potentially larger than a single tile.

Tile Assembly Computers (TAC). Informally, a Tile Assembly Computer (TAC) for a function f consists of a set of tiles, along with a format for both input and output. The input format is a specification for how to build an input seed to the system that encodes the desired input bit-string for function f. We require that each bit of the input be mapped to one of two assemblies for the respective bit position: a sub-assembly representing "0" or a sub-assembly representing "1". The input seed for the entire string is the union of all these sub-assemblies. This seed, along with the tile set of the TAC, forms a tile system. The output of the computation is the final terminal assembly this system builds. To interpret what bit-string is represented by the output, a second *output* format specifies a pair of sub-assemblies for each bit. The bit-string represented by the union of these subassemblies within the constructed assembly is the output of the system.

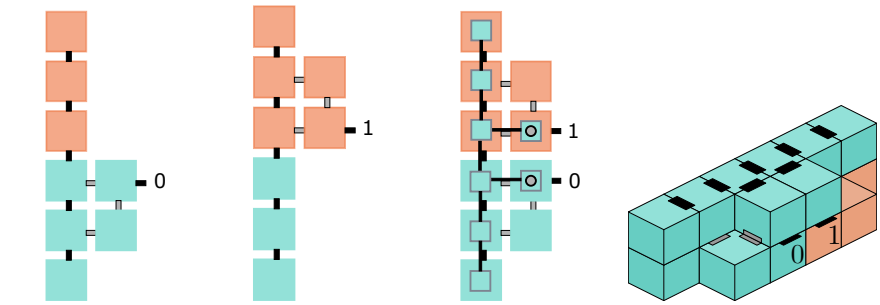
For a TAC to *covertly* compute f, the TAC must compute f and produce a unique assembly for each possible output of f. We note that our formulation for providing input and interpreting output is quite rigid and may prohibit more exotic forms of computation. Further, we caution that any formulation must take care to prevent "cheating" that could allow the output of a function to be partially or completely encoded within the input. To prevent this, a type of *uniformity* constraint, akin to what is considered in circuit complexity [27], should be enforced. We now provide the formal definitions of function computing and covert computation.

Input/Output Templates. An n-bit input/output template over tile set T is a sequence of ordered pairs of assemblies over T: $A = (A_{0,0}, A_{0,1}), \ldots, (A_{n-1,0}, A_{n-1,1})$. For a given n-bit string $b = b_0, \ldots, b_{n-1}$ and n-bit input/output template A, the *representation* of b with respect to A is the assembly $A(b) = \bigcup_i A_{i,b_i}$. A template is valid for a temperature τ if this union never contains overlaps for any choice of b and is always τ-stable. An assembly $B \supseteq A(b)$, which contains $A(b)$ as a subassembly, is said to represent b as long as $A(d) \nsubseteq B$ for any $d \neq b$. We refer to the size of a template as the size of the largest assembly defined by the template.

Function Computing Problem. A *tile assembly computer* (TAC) is an ordered quadruple $\Im = (T, I, O, \tau)$ where T is a tile set, I is an n-bit input template, and O is a k-bit output template. A TAC is said to compute function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2^k$ if for any $b \in \mathbb{Z}_2^n$ and $c \in \mathbb{Z}_2^k$ such that $f(b) = c$, then the tile system $\Gamma_{\Im,b} = (T, I(b), \tau)$ uniquely assembles a set of assemblies which all represent c with respect to template O.

Covert Computation. A TAC *covertly* computes a function $f(b) = c$ if 1) it computes f, and 2) for each c, there exists a unique assembly A_c such that for all b, where $f(b) = c$, the system $\Gamma_{\Im,b} = (T, I(b), \tau)$ uniquely produces A_c. In other words, A_c is determined by c, and every b where $f(b) = c$ has the exact same final assembly.

Polynomial-Sized Tile Assembly Computers. We say a TAC is polynomial size if the input template, tile set, and output template are all polynomial in n. However, this requirement still allows the producible assemblies to be exponentially larger. We say a TAC is *strictly* polynomial size if the produced assemblies are also polynomial in size.



(a) Bit Assembly Template 0 **(b)** Bit Assembly Template 1 **(c)** Bit Assembly Output Template **(d)** Bit Assembly Output Template (isometric view)

Figure 1 Input assemblies and their respective input templates. The blue squares represent the bit set to zero, and the orange squares represent a bit set to one. Grey glues are strength-1, black glues are strength-2.

3 3-Dimensional Covert Circuits

In this section, we show how to perform covert computation in the aTAM using 3 dimensions. The computation behaves similarly to the covert circuit construction in [9] by building NAND gates and FANOUTs using dual rail logic. We start with showing a NOT that switches which wire is "on", then extending to a NAND by utilizing cooperative binding.

The main difference between the two constructions is when *backfilling* occurs, which is the process of filling in the unused dual rail line once that line is no longer needed. Here, we do not backfill as we go, rather, we fill in the assembly once the computation is complete.

3.1 Input Assemblies

Our input assembly consists of n 1×6 columns with two of four tiles attached on the right (Figures 1a and 1b). The top two tiles will be included when the input is 1, and the bottom two tiles if the input is 0. These tiles have enough attachment strength to be stable when both are present, however, since the tiles only have strength 1 bonds, they may not attach alone. This initially prevents the growth of the other bit, which is not placed until the computation is complete, further elaboration of this process is described in section 3.5.

3.2 Wires and NOT Gates

Bit information is represented and transferred using a wire. A wire is constructed using two rows of tiles (Figure 2a), each representing a binary value of 0 or 1. This dual rail system initially grows only one of the rows from the input assembly based off the input and then builds into the gates. Before the circuit finishes growing, only one row of each wire will be constructed, and at the end, the other wire row will be built.

Gates such as the NOT grow off the wires. An example of a NOT gate can be seen in Figure 2b, notice how we utilize the third dimension to cross the wires over each other. This gate swaps the position of the rows of tiles; a row that represents a 0 will now be in the upper row and represent a 1. At the end of each gate is a diode gadget that was used in previous work [9]. The gadget is a 2×2 subassembly that grows only in one direction. If the first tile is placed, the whole thing will be first. If the last tile is placed, nothing else grows since it connects using two strength 1 glues. This prevents errors caused by "backward" growth.

(a) Dual Rail Logic **(b)** NOT Gate

Figure 2 (a) We use dual rail gates. The input glue of 1 grows the orange tiles and 0 grows the blue. (b) A NOT gate is implemented by crossing the wires over each other.

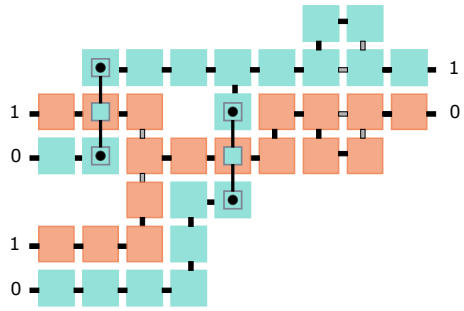


Figure 3 Full NAND Gate construction in the full circuit. The tiles in orange represent tiles that will be built from an input of 1 input, while the blue tiles come from an input of 0.

3.3 NAND Gates

We construct a NAND gate using the NOT gate and cooperative binding. The full NAND gate can be seen in Figure 3. If either input to a NAND gate is 0, the output is always 1. This can be seen in Figures 4a, 4b, and 4c. If any blue tile is placed, the 1 output of the gate will be built. If both inputs are 1, the 0 output can be constructed using cooperative binding.

One thing to note in the case of one output being 0 and the other being 1 is that the blue tiles will be placed along the other wire. However, this will not cause any issues since it can only build back up to the output of the previous gate due to the diode gadget.

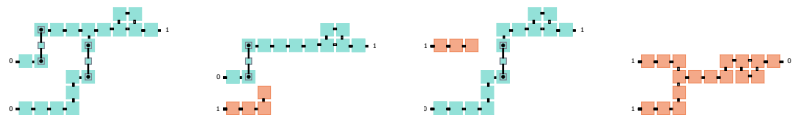
3.4 Fan Out and Crossover

Two other gadgets that assist in creating circuits are the fan out and crossover gadgets. The fan out (Figure 5a) splits a wire in order to copy the value to two gates. It does this by having each tile path split, and then use the third dimension to swap the positions.

The crossover gadget (Figure 5c) allows for the creation of non-planar circuits. Using the third dimension, a wire can go over another wire in order to reach its input. While such 3D crossovers simplify constructions greatly, we note that such crossovers are not necessarily needed, as planar circuits can simulate such crossovers using XOR gates [9].

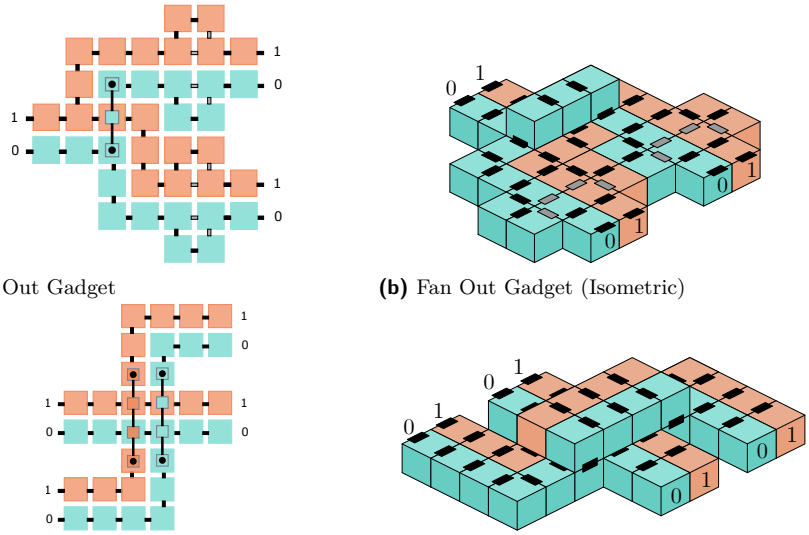
3.5 Backfilling and Target Assemblies

In order to perform covert computation, there must exist a unique assembly for each output. The gray tile at the end of the circuit in Figure 6a is one of two flag tiles that denotes the output of the circuit. Once this tile is placed, a row of tiles is built back towards the input (Figure 6b). Once the input assembly is reached, the tiles above the input are placed, thus



(a) Input 00 **(b)** Input 01 **(c)** Input 10 **(d)** Input 11

Figure 4 Growth of possible inputs to a NAND gate. The gate will stay like this after computing, before the history is hidden.



(a) Fan Out Gadget **(b)** Fan Out Gadget (Isometric)

(c) Crossover Gadget **(d)** Crossover Gadget (Isometric)

Figure 5 (a) A fan out gadget. (b) Isometric view of the fan out gadget. (c) While a crossover is not required for universal computation, we can easily implement one by using the 3rd dimension. (d) Isometric view of the crossover gadget.

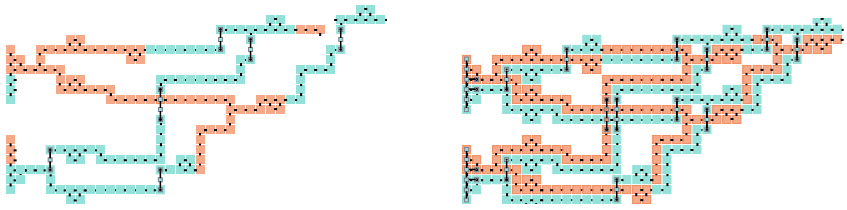
allowing for the input assemblies to be filled in. This causes the entire circuit to be filled out, which hides the original input and computation history.

▶ **Theorem 1.** *For any n-bit function f that is computable by a Boolean circuit, there exists a Tile Assembly Computer $\Im$ which covertly computes f in the 3D aTAM with only positive glues. Further, $\Im$ is strictly polynomial in n.*

Proof. We can construct the tile set T_c from the circuit c that computes f. Arrange the gates and wires on the square grid using $\mathcal{O}(n^2)$ space, and scale up each gate and wire by a constant factor. Wires are scaled up by a factor of 2 to account for the dual rail logic wires. The gates are scaled up by a factor depending on which gate it is, however, all the gates we present are only a constant size. This creates assembly $A_{c,Full}$.

We now show that $\Im$ computes f. Consider an n-bit input x to f, using the input template create seed assembly A_x. Each gate will grow from A_x, computing the circuit on each input. Since backfilling does not occur until the circuit finishes computing, we guarantee only the correct outputs grow from the final gate. The circuit is computed covertly since the output then grows back to the start of the circuit and places the unused inputs. ◀

The 3rd dimension is vital in this construction to allow signals to cross over for the NOT gate. Notice the part of the NAND gadget that is computing the AND gate and how the



(a) Computed Circuit **(b)** Output Assembly

Figure 6 Example structures of the computation circuit of an XOR using NOTs and NANDs. The circuit before backfilling is on the left, and the final output is shown on the right side. (a) A circuit once the output is computed. (b) Once the output grows backward, the other input bits are placed.

diode uses cooperative binding. Additionally, it would not be possible to build the full input gadget to allow the circuit to backfill. The positions that must be filled will be blocked on one side by the input assembly and on the other by wire. The backfilling here is used differently than in [8] since there each gate would backfill its input wires. There the negative glues were used to allow the tiles to cross over signals to build a NOT gate.

4 Exponential Assembly Covert Computer in 2D

In this section, we show that covert computation is possible in 2D in the standard aTAM, where the input can be described in polynomial size, yet the final terminal assembly is exponential in size. Thus, while we are able to achieve strictly polynomial-sized covert computation in 3D, we achieve (non-strict) polynomial-sized covert computation in 2D.

This construction is possible by first computing the function using reversible Toffoli gates, and then replicating and computing the circuit for all possible inputs. Once the output of the original input is placed, the Toffoli gate reverses its computation to build a mirror of the circuit with the input replicated on both the right and left. The output builds an assembly arm used to place tiles on either side of the assembly to increment and decrement the mirrored inputs based on the binary value of the original input, thus seeding a new input for exponential growth in each direction. Thus, for a 4-bit input, it builds the circuit for all 2^4 possible inputs after it builds the output template.

4.1 Toffoli Gate

The Toffoli gate is a 3-bit reversible universal logic gate (Figure 7a), we denote the inputs A, B, C, and the outputs A', B', C'. The first two input and output bits map to each other: $A = A'$ and $B = B'$. The third output flips the C bit if both A and B are 1. Logically expressed, this is $C \otimes (A \wedge B) = C'$.

We can express an n-bit d-depth reversible circuit as a $n \times d$ grid where each row represents a wire, and each column is a layer of gates and wires. Each gate can be represented by tiles computing the elementary 2-bit AND and XOR and implementing a fan out, as shown in Figure 7b.

4.2 Covert Circuit

The input template is a specific tile for each bit. Given an n-bit string, we create a $n \times 1$ bit assembly with stability-granting left and bottom circuit construction scaffolds, as shown in

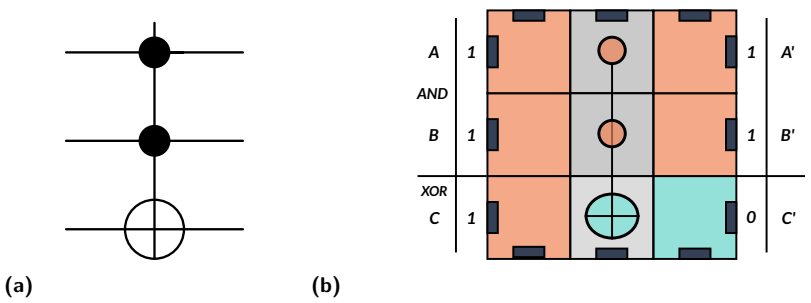


Figure 7 (a) Logical representation of Toffoli gate. (b) A Toffoli gate on a grid can be represented by the three vertical 'cells' of elementary logic gates.

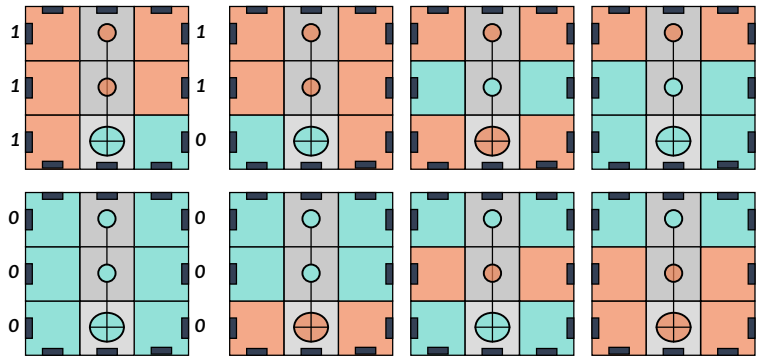


Figure 8 All possible computations of a single Toffoli gate. 1 (orange), 0 (blue). $111 \to 110, 110 \to 111, 101 \to 101, 100 \to 100$, Row 2: $000 \to 000, 001 \to 001, 010 \to 010$, and $011 \to 011$.

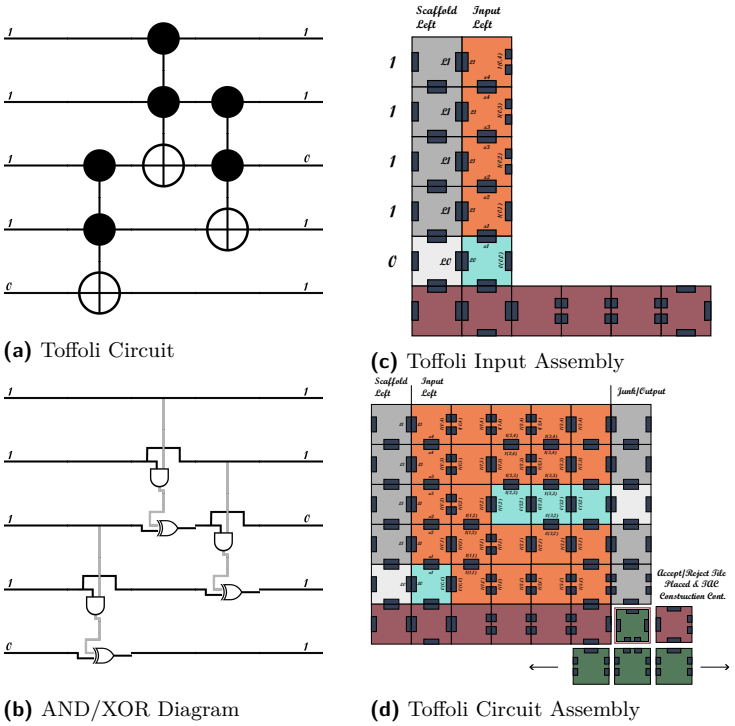
Figure 9c.

The circuit assembly is a $n \times (d+2)$ rectangle. Each Toffoli gate is a 3×1 subassembly. Three possible computations of a single Toffoli gate are shown in Figure 8. Typically, these gates must be reversible, meaning the circuit may grow from the east or west but produce the same assembly We note that the gate itself is not covert, and the "covertness" comes from the full construction.

An example Toffoli circuit is shown in Figure 9a along with the logical representation in Figure 9b. A constructed circuit assembly in one direction can be seen in Figure 9d.

4.3 Increment/Decrement Input to Next Circuit Logic

After completion of a circuit, three columns of tiles are built: mark for increment (left), copy or flip (center), and mark for decrement (right). The order of growth of these columns depends on the starting direction. Growing from the left to increment input to the next circuit or from the right to decrement it. Cooperatively with those columns, below the output arm begins its extension to transmit the outcome, accept or reject, of the original circuit. This arm extension continues to the center circuit output outcome tile location. From here, the circuit construction scaffold, previously provided in the input template, may loop back to the edge of the circuit so the new input scaffold and bits may place as illustrated in Figure 11. The circuit growth continues normally from that point forward, with the exception of the output tile placement.



(a) Toffoli Circuit

(b) AND/XOR Diagram

(c) Toffoli Input Assembly

(d) Toffoli Circuit Assembly

Figure 9 (a) Example 5-bit Toffoli Circuit. (b) The Toffoli circuit represented with AND and XOR gates. (c) Example Input Assembly. For each bit (1 or 0), we place the scaffold (grey or white) and input bit tile (orange or blue). The bottom is a row of circuit construction scaffold tiles (maroon). (d) The Toffoli Circuit Assembly built in one direction. The (green) tile below the output/junk column represents the (positive) output and will allow the output control row to place.

4.4 Output Assembly

Once the output is built, the rows below have d tiles attached in the east and west directions that encode the output. Through cooperative attachment, tiles are placed to allow the strings to increment/decrement, as described above. The final terminal assembly contains every possible computation.

▶ **Theorem 2.** *For all functions $f(x)$ that are computable by a n-bit reversible circuit R, there exists a polynomial tile assembly computer $\Im = (T, I, O, 2)$ that covertly computes $f(x)$ and has an output assembly of size $\mathcal{O}(2^n)$.*

Proof. If there exists a n-bit reversible circuit R that computes $f(x)$, we construct tile assembly computer $\Im = (T, I, O, 2)$ as follows. From the circuit R that computes f, we design a circuit R' to compute f with Toffoli gates as described in section 4.2. Using R' and the developed input increment/decrement logic for circuit replication, we construct a tile set T_c.

We create the input assembly I by converting the n-bit input string x to tiles L_i in scaffold left (figure 9c) and associated input, and a bottom row of tiles called the left circuit construction scaffold.

From here, the left assembly will grow into figure 9d, once the output is determined to be 'accept' or 'reject', the output indicator tile is placed, and the original output indicator arms

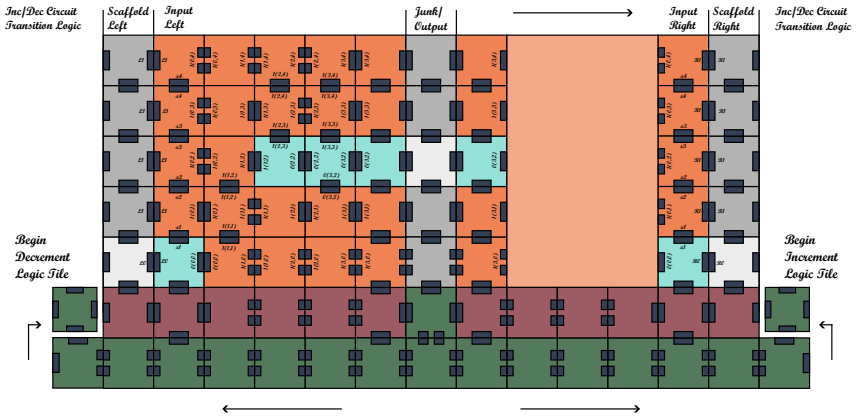


Figure 10 An example of a symmetrical circuit that has built both sides and is placing begin decrement and increment logic tiles.

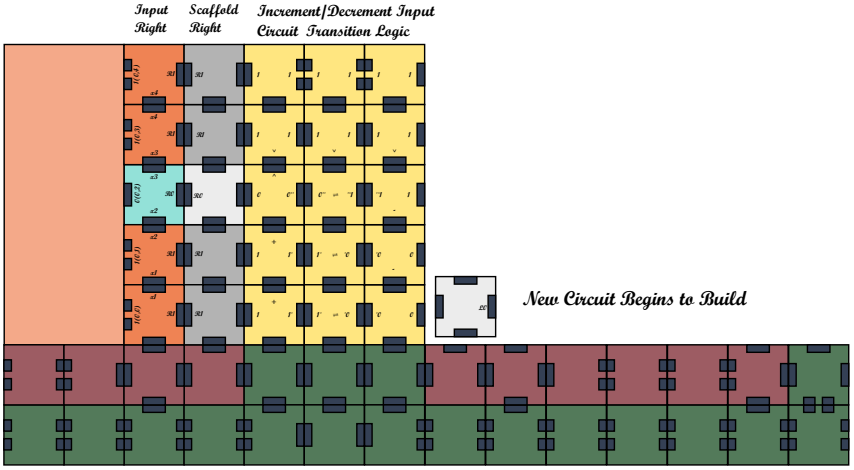


Figure 11 An example of a new circuit created by incrementing the output from a previously built assembly.

grow to allow the Right Assembly the ability to grow as well as place begin decrement and increment logic tiles on the bottom left and right sides of the completed assembly respectively, as seen in figure 10.

All n-bit computations of $f(y)$ for y less than original input x will be computed to the left of the original assembly, and all $x_n > x$ after being decremented and incremented using the reversible and symmetric logic in yellow from figure 11. Growth is halted by the INC/DEC logic at overflow in either direction.

The ability to grow further left/right circuit construction scaffolds is dependent on the output arms from the original output indicator arms growing to the center of the circuit about to begin construction where the output accept/reject indicator tile would place, preserving the output status for every circuit built in the TAC.

As there are only two possible assemblies that can be built, accept all or reject all, the Tile Assembly Computer is polynomial size in description and exponential in output size. ◀

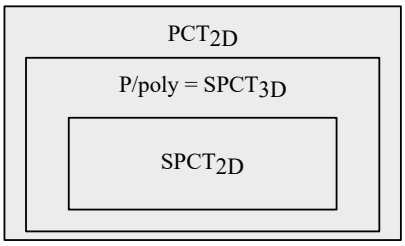


Figure 12 Diagram showing important classes defined in this section and their relation to P/poly. Note that none of these containments are known to be proper.

We have shown that if the output assembly is allowed to be exponential in size, that covert computation is possible in the aTAM, even in two dimensions. However, in practice, this is not usually a plausible solution. Given that Unique Assembly Verification is in P [2], it is unlikely that covert computation is possible with a strictly polynomial-size TAC.

▶ **Conjecture 3.** *There does not exist a strictly polynomial-size Tile Assembly Computer in the 2D Abstract Tile-Assembly Model.*

5 Polynomial-Sized Covert Circuits

In this section, we define and investigate complexity classes based on decision problems computable by polynomial-sized covert computers. We start by introducing the class P/poly and defining three classes of covertly computable problems: the class of problems covertly computable by a strictly polynomial 3D system (SPCT_{3D}), the class of problems computable by a strictly polynomial 2D system (SPCT_{2D}), and the class of problems computable by a (non-strict) polynomial 2D system (PCT_{2D}). We show how these classes relate to each other, including the result that P/poly is equal to SPCT_{3D}. Our results in this section are summarized in Figure 12.

5.1 Complexity Classes

The class P/poly is a well-studied complexity class defined as the class of problems solvable by a polynomial-sized circuit. One note about this class is it puts no requirement on the circuit other than that it exists. This has an equivalent definition as the problems solvable by a polynomial-time Turing machine with a polynomial advice string. We can think of this as the Turing machine being given a description of the circuit and evaluating it. Here, the advice string or circuit must be identical for all inputs of length n. [1]

▶ **Definition 4** (P/poly)**.** *The class of problems solvable by a polynomial-sized Boolean circuit. Alternatively, defined as the problems solvable by a polynomial-time Turing machine $M < x, a_{|x|} >$, where x is the input and $a_{|x|}$ is an advice string that is based only on the length of x. That is, if two inputs x, y have the same size $|x| = |y|$, then they must use the same advice string.*

We define the following three complexity classes to categorize the functions that are computable by polynomial-size covert TACs.

[1] Under this definition, every unary language is in this class, including UHALT.

▶ **Definition 5** ($\mathsf{SPCT_{3D}}$). *The class of problems solvable by a strict polynomial sized covert tile assembly computer in the 3D Abstract Tile-Assembly Model.*

Formally, a language L is in $SPCT_{3D}$ if there exists a sequence of covert TACs $C = \{C_1, C_2, ...\}$ such that the i^{th} TAC, C_i, is strictly polynomial in i and if it correctly computes all $x \in L$ where $|x| = i$.

▶ **Definition 6** ($\mathsf{SPCT_{2D}}$). *The class of problems solvable by a strict polynomial sized covert tile assembly computer in the 2D Abstract Tile-Assembly Model.*

▶ **Definition 7** ($\mathsf{PCT_{2D}}$). *The class of problems solvable by a polynomial sized covert tile assembly computer in the 2D Abstract Tile-Assembly Model.*

5.2 Strict Polynomial Size Equivalence

To show equivalence between P/poly and $\mathrm{SPCT_{3D}}$, we first define the 2-Promise Unique Assembly Verification problem, a modified version of Unique Assembly Verification where we are given two assemblies, a and b, rather than a single target. The problem asks to separate two cases: accept if an assembly containing a as a subassembly is produced, and reject if an assembly containing b is produced. We assume it is promised that one of these cases is true. This problem is solvable in polynomial time since you only need to attach tiles until one of the two assemblies is produced (Lemma 9).

▶ **Definition 8** (2-Promise Unique Assembly Verification problem). ***Input:*** *Assemblies a, b and an aTAM system (T, s, τ) which is promised to uniquely produce one of two assemblies, A or B, such that $a \subseteq A$ and $b \subset B$.* ***Output:*** *'Yes', if Γ uniquely assembles A, and 'No', if Γ uniquely assembles B.*

▶ **Lemma 9.** *The 2-Promise Unique Assembly Verification problem is solvable in polynomial time in the 3D aTAM.*

Proof. Call greedy grow (from [2]) to get maximal producible assembly C. If Γ uniquely assembles C and $a \subseteq C$, return 'yes'. Otherwise, return 'no'. ◀

Equipped with the algorithm for the 2-promise problem, and taking the description of a covert computer as an advice string, it follows that we can compute the seed assembly from the input template, and the two possible output assemblies from the output template, and then run the algorithm for the 2-Promise UAV problem (Lemma 10). This puts any problem solvable by a polynomial-sized covert circuit in the class P/poly. The other direction of equivalence is given by the 3D covert computer constructions.

▶ **Lemma 10.** *If a language L is computable by a strict polynomial-sized covert tile assembly computer in the 3D aTAM, then L is in P/poly.*

Proof. Let $\Im_n(T, I, O, \tau)$ be the covert computer for the strings in language L of size n. Since $\Im_n$ is of strict polynomial size, we can encode the tile set, input/output templates, and temperature in $poly(n)$ bits. Thus, $\Im_n$ will be our advice string for membership in P/poly. Further, we are only considering decision problems. Thus, there are only two output templates which we denote as a_a and b_r for accept and reject, respectively.

Consider a Turing machine given the string x and covert circuit $\Im_{|x|} = (T, I, (a_a, b_r), \tau)$ that does the following:

- Convert x to an assembly $I(x)$ using the input template.

- Call the algorithm for 2-Promise UAV on input $((T, I(x), \tau), a_a, b_r)$.
- If the algorithm accepts then $x \in L$, else $x \notin L$

This Turing machine essentially runs the covert computer on x and then checks the output by seeing which template is included in the final assembly. ◀

▶ **Theorem 11.** *The classes $SPCT_{3D}$ and P/poly are equivalent.*

Proof. By Lemma 10, if a language is in P/poly there is a Boolean circuit of polynomial size which computes it, giving us P/poly $\subseteq$ PCT$_{3D}$. In Theorem 1 we show that if there exists a Boolean circuit, there exists a strictly polynomial sized covert computer that computes the circuit. ◀

5.3 Polynomial Sized 2D Covert Circuits

Here, we use previous constructions to show that the class of polynomial sized 2D covert circuits is at least as strong as strict polynomial covert circuits. That is every language in $SPCT_{3D}$ is in PCT_{2D}.

▶ **Theorem 12.** *If a language L is in P/poly then L is in PCT_{2D}*

Proof. In Lemma 10 we show that if a language is in P/poly there is a Boolean circuit of polynomial size which computes it. Any Boolean circuit can be turned into a reversible circuit, thus by Theorem 2, if there exists a reversible circuit, there exists a polynomial tile assembly computer that computes it in 2D. ◀

6 Conclusion and Future Work

Previous work in the aTAM required negative glues in order to build covert Tile Assembly Computers. We have provided two new covert computers in the aTAM with only positive glue strengths, one in (just-barely) 3D and one in 2D with an exponential-sized output assembly. These covert TACs add new tools to the field that may find use in future complexity results, or in future applications related to privacy, cryptography, or biological computation. We have further initiated the study of covert computers in the context of known complexity classes, showing connections to the well-studied class P/poly. These results motivate future work to find functions that can be covertly computed in the 2D aTAM with strict polynomial size, such as (perhaps) Branching Programs.

Some additional specific directions for future work are as follows. We show the containment of the class of strict polynomial computers to be in P/poly. Can this be improved? Could we possibly use the P/poly log space analogue L/poly? What about in smaller classes, such as covert computers with non-cooperative binding or at temperature-1?

References

1 Zachary Abel, Nadia Benbernou, Mirela Damian, Erik D. Demaine, Martin L. Demaine, Robin Flatland, Scott D. Kominers, and Robert Schweller. Shape replication through self-assembly and rnase enzymes. In *Proceedings of the 2010 Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'10, pages 1045–1064, 2010. doi:10.1137/1.9781611973075.85.

2 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.

3 Andrew Alseth and Matthew J. Patitz. The need for seed (in the abstract tile assembly model). In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA'23, pages 4540–4589, 2023.

4 Spring Berman, Sándor P Fekete, Matthew J Patitz, and Christian Scheideler. Algorithmic foundations of programmable matter (dagstuhl seminar 18331). In *Dagstuhl Reports*, volume 8. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

5 Yuriy Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Comp. Sci.*, 378:17–31, 2007.

6 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Covert computation in staged self-assembly: Verification is pspace-complete. In *Proceedings of the 29th European Symposium on Algorithms*, ESA'21, 2021.

7 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Verification and computation in restricted tile automata. *Natural Computing*, 2021. doi:10.1007/s11047-021-09875-x.

8 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.

9 Angel A. Cantu, Austin Luchsinger, Robert T. Schweller, and Tim Wylie. Covert computation in self-assembled circuits. *Algorithmica*, 83:531 – 552, 2019.

10 Luis Ceze, Jeff Nivala, and Karin Strauss. Molecular digital data storage using dna. *Nature Reviews Genetics*, 20(8):456–466, 2019.

11 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.

12 Cameron T. Chalk, Eric Martinez, Robert T. Schweller, Luis Vega, Andrew Winslow, and Tim Wylie. Optimal staged self-assembly of general shapes. *Algorithmica*, 80(4):1383–1409, 2018. doi:10.1007/s00453-017-0318-0.

13 Qi Cheng, Gagan Aggarwal, Michael H. Goldwasser, Ming-Yang Kao, Robert T. Schweller, and Pablo Moisset de Espanés. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34:1493–1515, 2005.

14 Supreme Court. Ass'n for molecular pathology v. myriad, 2013.

15 Erik Demaine, Matthew Patitz, Robert Schweller, and Scott Summers. Self-assembly of arbitrary shapes using rnase enzymes: Meeting the kolmogorov bound with small scale factor. *Symposium on Theoretical Aspects of Computer Science (STACS2011)*, 9, 01 2010. doi:10.4230/LIPIcs.STACS.2011.201.

16 Erik D. Demaine, Sándor P. Fekete, Christian Scheffer, and Arne Schmidt. New geometric algorithms for fully connected staged self-assembly. *Theoretical Computer Science*, 671:4 – 18, 2017. Computational Self-Assembly. URL: http://www.sciencedirect.com/science/article/pii/S030439751630679X, doi:https://doi.org/10.1016/j.tcs.2016.11.020.

17 David Doty, Lila Kari, and Benoît Masson. Negative interactions in irreversible self-assembly. *Algorithmica*, 66(1):153–172, 2013.

18 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 *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 302–310. IEEE, 2012.

19 Pim WJM Frederix, Ilias Patmanidis, and Siewert J Marrink. Molecular simulations of self-assembling bio-inspired supramolecular systems and their connection to experiments. *Chemical Society Reviews*, 47(10):3470–3489, 2018.

20 David Furcy, Samuel Micka, and Scott M. Summers. Optimal program-size complexity for self-assembled squares at temperature 1 in 3d. *Algorithmica*, 77(4):1240–1282, March 2016. doi:10.1007/s00453-016-0147-6.

21 David Furcy, Scott M. Summers, and Logan Withers. Improved Lower and Upper Bounds on the Tile Complexity of Uniquely Self-Assembling a Thin Rectangle Non-Cooperatively in 3D.

In Matthew R. Lakin and Petr Šulc, editors, *27th International Conference on DNA Computing and Molecular Programming (DNA 27)*, volume 205 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/14671`, `doi:10.4230/LIPIcs.DNA.27.4`.

22 Daniel Hader, Aaron Koch, Matthew J Patitz, and Michael Sharp. The impacts of dimensionality, diffusion, and directedness on intrinsic universality in the abstract tile assembly model. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2607–2624. SIAM, 2020.

23 Adam M Kabza, Nandini Kundu, Wenrui Zhong, and Jonathan T Sczepanski. Integration of chemically modified nucleotides with dna strand displacement reactions for applications in living systems. *Wiley Interdisciplinary Reviews: Nanomedicine and Nanobiotechnology*, 14(2):e1743, 2022.

24 Alexandra Keenan, Robert Schweller, Michael Sherman, and Xingsi Zhong. Fast arithmetic in algorithmic self-assembly. *Natural Computing*, 15(1):115–128, Mar 2016.

25 Paul WK Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468, 2000.

26 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.

27 Heribert Vollmer. *Introduction to Circuit Complexity*. Springer Berlin Heidelberg, 1999. `doi:10.1007/978-3-662-03927-4`.

28 Klaus F Wagenbauer, Christian Sigl, and Hendrik Dietz. Gigadalton-scale shape-programmable dna assemblies. *Nature*, 552(7683):78–83, 2017.

29 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.

APPENDIX D

APPENDIX D

REACHABILITY IN RESTRICTED CHEMICAL REACTION NETWORKS

Reachability in Restricted Chemical Reaction Networks

Robert M. Alaniz
University of Texas Rio Grande Valley, Edinburg, TX, USA

Bin Fu
University of Texas Rio Grande Valley, Edinburg, TX, USA

Timothy Gomez
Massachusetts Institute of Technology, Cambridge, MA, USA

Elise Grizzell
University of Texas Rio Grande Valley, Edinburg, TX, USA

Andrew Rodriguez
University of Texas Rio Grande Valley, Edinburg, TX, USA

Robert Schweller
University of Texas Rio Grande Valley, Edinburg, TX, USA

Tim Wylie
University of Texas Rio Grande Valley, Edinburg, TX, USA

Abstract

The popularity of molecular computation has given rise to several models of abstraction, one of the more recent ones being Chemical Reaction Networks (CRNs). These are equivalent to other popular computational models, such as Vector Addition Systems and Petri-Nets, and restricted versions are equivalent to Population Protocols. This paper continues the work on core *reachability* questions related to Chemical Reaction Networks; given two configurations, can one reach the other according to the system's rules? With no restrictions, reachability was recently shown to be Ackermann-complete, which resolved a decades-old problem.

In this work, we fully characterize monotone reachability problems based on various restrictions such as the allowed rule size, the number of rules that may create a species (k-source), the number of rules that may consume a species (k-consuming), the volume, and whether the rules have an acyclic production order (*feed-forward*). We show PSPACE-completeness of reachability with only bimolecular reactions in two-source and two-consuming rules. This proves hardness of reachability in a restricted form of Population Protocols. This is accomplished using new techniques within the motion planning framework.

We give several important results for feed-forward CRNs, where rules are single-source or single-consuming. We show that reachability is solvable in polynomial time as long as the system does not contain special *void* or *autogenesis* rules. We then fully characterize all systems of this type and show that with void/autogenesis rules, or more than one source and one consuming, the problems become NP-complete. Finally, we show several interesting special cases of CRNs based on these restrictions or slight relaxations and note future significant open questions related to this taxonomy.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases Chemical Reaction Networks, reachability, hardness

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The popularity of molecular computation and the need to model distributed reactions has given rise to several models of abstraction and multiple areas of research. Many of these models arose naturally in different fields decades apart, yet mathematically are nearly

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:24

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

equivalent. The focus of this paper is on Chemical Reaction Networks (CRNs) [4, 5], a model equivalent [8] to Vector Addition Systems (VASs) [16] and Petri-Nets [20]. Further, the Population Protocols model [2] is just a restricted version of these models, limited by the number of input and output elements in each operation.

Although these models may be substantively equivalent, we focus on CRNs for two reasons: first, due to the simplicity and convenience of the system definition. Specifically, expressing the production operations through reaction rules gives a straightforward rubric to measure and characterize the power of the system. The second is because CRNs, in some form, are also the oldest formulation of these types of distributed systems. Even though the reaction rule description is more intuitive in many cases, in our formal definitions, we rely on a matrix interpretation for notational convenience with precision, similar to Vector Addition Systems.

From the inceptions of each of these models, two of the most fundamental questions have always been *production* and *reachability*. Production asks if some species/element can ever be produced. Reachability simply asks if, given a system and initial configuration, whether it will even reach another specific configuration. As the expectation of behaviors and the attainability of end goals are necessary in the design of any of these systems, production and reachability are at the heart of system design. Despite the fundamental nature of reachability questions, most remain open or have only recently been solved, while production is better understood overall. Since many results are applicable, or equivalent, between these models, we briefly survey the reachability question in relation to each of the models.

1.1 An Overview of Related Models

While reachability was proven EXPSPACE-hard 1976 in [18] and decidable in 1981 [19], completeness wasn't proven until 2021 when the unrestricted version of the reachability problem was proven to be Ackermann-complete [9, 17] for the equivalent models of Vector Addition Systems, Petri-Nets, and Standard CRNs.

Chemical Reaction Networks. The desire and attempt to bring a mathematical abstraction to model chemical reactions has been well-documented for over a hundred years, but began in its modern incarnation during the 1960s as chemical reaction network theory [4, 5]. Given that CRNs are motivated by actual chemical processes, there is a reasonably distinct split between the purely mathematical and applied experimental work in the research literature. Applied work typically uses stochastic versions of the model. In *Proper* CRNs, where reactions do not change the system's total volume, reachability and production are known to be PSPACE-hard even with catalytic reactions [13].

Vector Addition Systems. Vector Addition Systems (VAS) have an initial vector of non-negative integers and a set of operational integer vectors. The system is allowed to add these vectors however desired, as long as none of the values in the initial/counter vector ever become negative. The related questions of coverage, asking if there exists a reachable configuration with at least the given number of each species, and boundness, asking if the set of reachable configurations is finite, were shown to be in EXPSPACE [22]. Other relevant reachability results include, VASs with dimension ≤ 5 were proven decidable [15], as the set of reachable configurations is a semilinear set. In VAS with states (VASS), it was also shown that there exists 6-dimensional VASs whose reachable configurations can form non-semilinear functions. In 2-Dimensional VASS, reachability is PSPACE-complete if the values of the input and target vector are encoded in binary and NL-complete if encoded in unary [6].

Petri-Nets. Petri-Nets were first formally introduced in [20] in 1962 to visually describe chemical processes. Petri-Nets provide a graphical representation of reactions by having *places*

General CRNS					
Cons.	*Source*	*Rule Size*	*Volume*	*Membership*	*Theorem*
$\mathcal{O}(q)$	$\mathcal{O}(q)$	$(2,2)$	U/B	PSPACE-complete	[13]
2	2	$(2,2)$	U/B	PSPACE-complete	Thm. 13
2	2	$(1,2)$ and $(2,1)$	U/B	PSPACE-hard	Thm. 15
j	k	$(1,1)$	U/B	NL-hard	Thm. 18
1	1	$(k,1)$	U/B	P	Thm. 20
Feed-Forward					
Cons.	*Source*	*Rule Size*	*Volume*	*Membership*	*Theorem*
2	2	$(1,2)$ or $(2,2)$ or $(2,1)$	U/B	NP-complete	Thm. 26
j	1	No Void	U/B	P	Thm. 32
1	k	No Autogenesis	U/B	P	Thm. 33
1	1	Any	U/B	P	Cor. 34
Feed-Forward with Void/Autogenesis Rules					
Cons.	*Source*	*Rule Size*	*Volume*	*Membership*	*Theorem*
3	0	$(3,0)$	U/B	NP-complete	Thm. 35
j	0	$(2,0)$	U	P	Thm. 37
j	0	$(2,0)$	B	P (Bipartite)	Thm. 39

Table 1 Table of reachability results. The *Volume* column indicates the input encoding of the volume, and thus U is for unary and B is for binary. Rule sizes (a,b) indicates a elements interact as input to produce b elements as output. q denotes the number of states in the Turing machine.

with edges to *transitions*, which have edges to other places. Weights accompanying these edges correspond to consumption within reactions between different elements. Reachability in Petri-Nets has continued to be an active area of research, with numerous extensions and restrictions on the graph structure. Immediate observation Petri-Nets (IO Nets), introduced in [13], place a restriction on the transitions such that each transition involves 3 places: the source, the destination, and the observed place. The observed place acts as a 'catalyst' in the transition. By simulating a bounded-tape Turing machine, reachability in IO nets is shown to be PSPACE-hard [13].

Population Protocols. Created to model distributed and decentralized computing with agents that are highly resource-limited, Population Protocols was introduced in 2004 [1, 2]. In the model, at most two agents may interact at any time and may choose to change state based on this communication. Since only two agents interact at a time, the model can be viewed as a restricted form of CRNs, VASs, and Petri-Nets. Due to the limited nature of the agents, all reactions are volume-preserving with two inputs and two outputs. Reachability was proven to be PSPACE-hard in [13] via a limited version of IO Petri-Nets.

1.2 Our Contributions

In this paper, we improve on nearly all known results related to monotone volume-restricted CRNs, where each reaction preserves the volume of the system, based on several aspects of the rules. An overview of the major results is listed in Table 1. The results in the three sub-tables roughly correspond to the major results in Sections 3, 4, and 5, respectively.

Proper/General CRNs. The general reachability problem in CRNs is Ackermann-complete [9, 17], and thus we focus on the restriction of *proper* CRNs, in which each reaction/rule preserves or decreases system volume, putting the problem in PSPACE. Practical considerations motivate the study of small reactions, such as bimolecular reactions in which up to 2 molecules are used as products (such as in Population Protocols). Although

this was recently shown in [13], the number of rules producing and consuming any given species was based on the size of the input. We show that reachability is PSPACE-complete with bimolecular reactions even when at most two rules consume or produce a species (Theorem 13). Thus, reachability in population protocols is PSPACE-complete under this restriction (2-source, 2-consuming). In Corollaries 14 and 15, this reduction extends to the universal reachability problem, and reachability for non-monotone volume. For production, our PSPACE-complete result is tight with regards to rule size as we show NL-completeness for $(1, 1)$ rules in Theorem 18. We also provide several related smaller results.

Feed-Forward CRNs. We fully characterize, based on rule size, void/autogenesis rules, and the number of source/consuming rules, all CRN systems that are feed-forward, which is where reactions of the CRN permit an ordering such that products of later reactions do not occur as reactants in earlier rules [7]. We show that, without void/autogenesis rules, the feed-forward property alone moves the reachability problem into the class NP (Theorem 24). We show that reachability is polynomial-time solvable for a feed-forward system if it is *either* 1-consuming *or* 1-source and uses no *void* rules (rules that produce no species) or *autogenesis* rules (rules that consume no species). We prove that relaxing these restrictions makes reachability hard by showing that the problem in feed-forward systems is NP-complete in 2-consuming, 2-source systems (Theorem 26), and if they have void or autogenesis rules.

Consuming and Source. A k-consuming CRN is one where the number of rules that consume or "use up" a species is bound by k. k-source systems limit the number of rules that a species may be sourced from, or created by, the CRN. Non-Competitive CRNs [25] are a special case of 1-consuming where any rule that is consumed is not allowed to be a catalyst. Consuming and source metrics are not only interesting from a theoretical perspective since reachability is hard even for small k, but also from a implementation aspect. When implementing CRNs with DNA strand displacement, one method implements reactions as larger DNA complexes that bind to smaller DNA strands complexes [23]. Limiting source and consuming rules make the DSD systems less complex as we are limiting the number of complexes the strands must interact with.

Void and Autogenesis Rules. Our final consideration is the case of systems that utilize a particularly powerful class of rules: *void* rules which do not produce any species types, and *autogenesis* rules which do not consume any species types. We show that reachability is NP-complete even if system reactions are all void rules or all autogenesis rules of size $(3, 0)$ or $(0, 3)$, respectively (Theorem 35). We then explore the case of $(2, 0)$ void rule systems and show that reachability is polynomial-time solvable if the volume of the input configuration is polynomial bounded (Theorem 37), *or* if the CRN is *bipartite* (Theorem 39). For other $(2, 0)$ void rule systems, we leave the complexity of reachability as an open problem.

2 Preliminaries

Basics. Let $\Lambda = s_1, s_2, \ldots, s_{|\Lambda|}$ denote some ordered alphabet of *species*. A configuration over Λ is a length-$|\Lambda|$ vector of non-negative integers, denoting the number of copies of each present species. A *rule* or *reaction* has two multisets, the first containing one or more *reactant* (species) used to create resulting *product* (species), the second multiset. We represent each rule as an ordered pair of configuration vectors $R = (R_r, R_p)$. R_r contains the minimum counts of each reactant species necessary for reaction R to occur, where reactant species are either *consumed* by the rule in some count or leveraged as *catalysts* (not consumed); in some cases a combination of the two. The product vector R_p has the count of each species *produced* by the *application* of rule R, effectively replacing vector R_r. The species corresponding to

the non-zero elements of R_r and R_p are termed *reactants* and *products* of R, respectively.

The *application* vector of R is $R_a = R_p - R_r$, which shows the net change in species counts after applying rule R once. For a configuration C and rule R, we say R is applicable to C if $C[i] \geq R_r[i]$ for all $1 \leq i \leq |\Lambda|$, and we define the *application* of R to C as the configuration $C' = C + R_a$. For a set of rules Γ, a configuration C, and rule $R \in \Gamma$ applicable to C that produces $C' = C + R_a$, we say $C \rightarrow^1_\Gamma C'$, a relation denoting that C can transition to C' by way of a single rule application from Γ. We further use notation $C \rightsquigarrow_\Gamma C'$ to signify the transitive closure of $\rightarrow^1_\Gamma$ and say C' is *reachable* from C under Γ, i.e., C' can be reached by applying a sequence of applicable rules from Γ to initial configuration C. We use the following notation to depict a rule $R = (R_r, R_p)$:

$\sum_{i=1}^{|\Lambda|} R_r[i]s_i \rightarrow \sum_{i=1}^{|\Lambda|} R_p[i]s_i$

For example, a rule turning two copies of species H and one copy of species O into one copy of species W would be written as $2H + O \rightarrow W$.

▶ **Definition 1** (Discrete Chemical Reaction Networks). *A discrete chemical reaction network (CRN) is an ordered pair* (Λ, Γ) *where* Λ *is an ordered alphabet of species, and* Γ *is a set of rules over* Λ.

The primary computational problem we consider in this paper is the *reachability* problem. We consider additional problems in the paper, such as determining if it is possible to produce a given amount of a particular species from an initial configuration of a CRN, as well as universal reachability, which asks if the target configuration is reachable for all reaction application sequences.

▶ **Definition 2** (Reachability Problem.). *Given a CRN* (Λ, Γ)*, an initial configuration* I*, and a destination configuration* D*, the* Reachability Problem *is to compute whether or not* D *is reachable from* I *with respect to* Γ.

▶ **Definition 3** (Production Problem). *Given a CRN* (Λ, Γ)*, an initial configuration* I*, a species* $s_i \in \Lambda$*, and a positive integer* k*, decide if there exists a reachable configuration* B *such that* $B[i] \geq k$.

▶ **Definition 4** (Universal Reachability Problem.). *Given a CRN* (Λ, Γ)*, an initial configuration* I*, and a destination configuration* D*, the* Universal Reachability Problem *is to compute whether* D *is reachable from all configurations* M *that are reachable from* I *with respect to* Γ.

Primary Restrictions. We consider the reachability problem under several different restrictions defined below. Fig. 1 provides examples of the various restrictions on the model.

▶ **Definition 5** (Feed-Forward). *A CRN* (Λ, Γ) *is* feed-forward *if* Γ *permits an ordering on the rules such that the products of any given rule never occur as reactants for earlier rules of the ordering [7].*

Each rule in a system produces some species and consumes others. The following metric places a maximum bound on the number of rules that either produce a given species (j-source) or consume a given species (j-consuming).

▶ **Definition 6** (j-source, j-consuming). *A species* s_i *is* consumed *in rule* $R = (R_r, R_p)$ *if* $R_r[i] > R_p[i]$, produced *if* $R_r[i] < R_p[i]$*, and is a* catalyst *in rule* R *if* $R_r[i] = R_p[i] > 0$. *A CRN* (Λ, Γ) *is* j*-source if for all species* $s \in \Lambda$*,* s *is produced in at most* j *distinct rules in* Γ. *A CRN* (Λ, Γ) *is* j*-consuming if for all species* $s \in \Lambda$*,* s *is consumed in at most* j *distinct rules in* Γ. *We use the terms* single-source *and* single-consuming *for the special cases of* 1*-source and* 1*-consuming CRNs, respectively.*

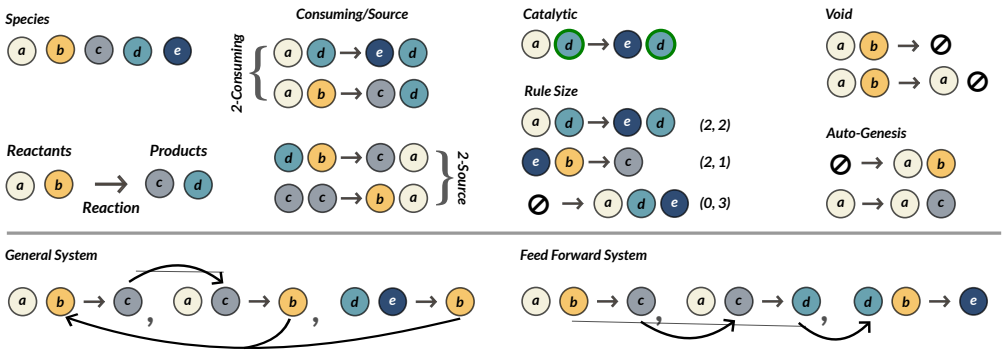


Figure 1 Example CRN rules to demonstrate the primary restrictions.

The next concept is a special class of rules that either produce nothing (void rules) or consume nothing (autogenesis rules). This could potentially be motivated by evaporation, the ability to pass through a membrane or the spontaneous appearance of ions in a vacuum.

▶ **Definition 7** (Void and Autogenesis rules). *A rule $R = (R_r, R_p)$ is a* void *rule if $R_a = R_p - R_r$ has no positive entries. A rule is an* autogenesis *rule if R_a has no negative values.*

Additional Restrictions. We also consider the complexity of reachability and production with respect to the *size* of rules.

▶ **Definition 8.** *The* size/volume *of a configuration vector C is* $\texttt{volume}(C) = \sum C[i]$.

▶ **Definition 9** (size-(i,j) rules). *A rule $R = (R_r, R_p)$ is said to be a size-(i,j) rule if $(i,j) = (\texttt{volume}(R_r), \texttt{volume}(R_p))$. A reaction is bimolecular if $i = 2$ and unimolecular if $i = 1$.*

▶ **Definition 10** (Volume Decreasing, Increasing, Preserving). *A rule $R = (R_r, R_p)$ of size-(i,j) is said to be volume decreasing if $i > j$, volume increasing if $i < j$, and volume preserving if $i = j$. A CRN (Λ, Γ) is said to be volume decreasing (respectively increasing, preserving) if all rules in Γ are volume decreasing (respectively increasing, preserving). Note: In previous work, volume preserving has been called* Proper.

A special subset of CRN systems studied in the literature is Population Protocols, in which agents bump into each other and adjust their state according to rules. This model is equivalent to a CRN that is limited to exactly volume 2 for both the reactants (the two agents that bump into each other) and the products (representing the two new states of the agents after the collision).

▶ **Definition 11** (Population Protocols). *A CRN (Λ, Γ) in which all rules in Γ are size-$(2,2)$ is called a population protocol.*

3 Reachability in General CRNs

The main result of this section is PSPACE-completeness of the reachability problem with 2-source, 2-consuming, size-$(2,2)$ reactions in Theorem 13. En route, we prove that production is PSPACE-complete with Theorem 12. We extend this reduction in two ways. First, in Corollary 14, we prove our reduction holds for the universal reachability problem, and second, we show the reduction holds for non-monotonic volume with rule sizes of $(1,2)$ and $(2,1)$ in Theorem 15. We follow with several interesting, yet minor results. Detailed proofs can be found in Appendix A.

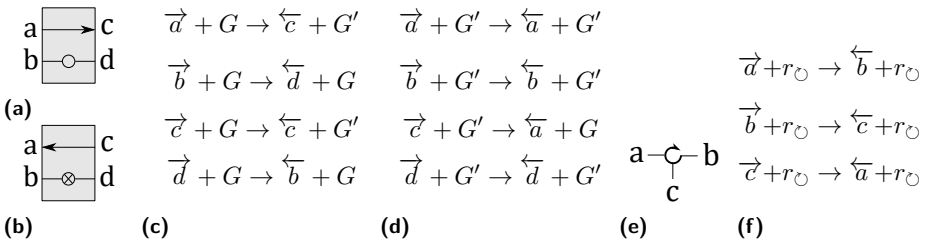


Figure 2 (a) Unlocked state of a Toggle-Lock gadget represented by species G. (b) Locked state of a Toggle-Lock gadget represented by species G'. (c-d) Reactions which implement a single gadget. (c) represents a successful traversal and (d) represents the 'bound-back' reactions. The arrow is incoming or outgoing from port. (e) The rotate gadget. (f) Rules for the rotate gadget.

3.1 Gadget Reconfiguration Framework

Our main result is based on the motion planning problem through Toggle-Lock and Rotate gadgets [11]. Motion planning is PSPACE-hard with only a rotate gadget and any reversible gadget with interacting tunnels, a class that includes the Toggle-Lock [10]. The motion planning problem considers two input configurations of gadgets, a start location for the agent, and a target location, and asks if the agent can reach the target location. [1]

A gadget consists of a set of labeled ports and states describing the gadget's legal traversals, which may change the state of the gadget. The Toggle-Lock Gadget has an unlocked and locked state, shown in Figure 2a and 2b, respectively. The top path is directed based on the state of the gadget, and the agent must follow the direction. Traversing this path changes the state of the gadget. Traversal of the bottom tunnel can only occur in the unlocked state; this can be done in either direction and does not change the state of the gadget. The rotate gadget only has one state that sends the agent to the next port, going clockwise. A motion planning system consists of a set of gadgets, a set of wires denoting port connections, and an initial signal location.

We are simulating a 0-player gadget framework where the agent makes no choices. The agent is directed down the wire and turns around if the gadget is not traversable in the current state, corresponding to a deterministic model of computation. An early version of this result appeared in the short abstract [14], which reduced from a different gadget and used the 1-player framework studied in [3, 10, 12]. However, the reduction was not constant source or constant consuming.

3.2 Production and Reachability

We prove production is PSPACE-complete using the framework described above. The target species is the agent species representing the target wire.

▶ **Theorem 12.** *Production in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.*

We extend the reduction above to work for reachability by taking advantage of the fact that the toggle-lock is reversible. Once we reach the target species, we flip the rotate catalyst to allow the agent to move counterclockwise through a rotate gadget, allowing us

[1] In [11] this problem is called the reachability problem. To avoid confusion, we refer to this as the motion planning problem.

to undo all the changes to the gadget states. When the agent reaches the starting location again, the system is in the initial configuration except with the opposite rotate catalyst. Reconfiguration for the 1-player version of the motion planning framework was shown to be PSPACE-complete using a similar technique where the agent changes the state of the final gadget, then undoes all of its previous movements [3]. This reduction extends to the universal reachability problem since there is only one reaction at each step that can be performed.

▶ **Theorem 13.** *Reachability in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.*

▶ **Corollary 14.** *Universal reachability in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.*

3.3 Volume Related Results

Here, we look at several restrictions related to rule size, and consequently, volume.

Non-Monotone Volume. We extend the reduction to utilize smaller rules. We show PSPACE-hardness of production and reachability when allowing both $(1, 2)$ and $(2, 1)$ rules. The CRN has both volume increasing and decreasing rules, which means it is non-monotone, and thus, these problems are not known to be in PSPACE. To prove this, we add an intermediate species for each reaction. We replace the reaction $\overrightarrow{a} + G \to \overrightarrow{c} + G'$ with the two reactions $\overrightarrow{a} + G \to \overrightarrow{aGc}$ and $\overrightarrow{aGc} \to \overrightarrow{c} + G'$.

▶ **Corollary 15.** *Production and reachability in 2-source, 2-consuming CRNs is PSPACE-hard with rules of size* $(2, 1)$ *and* $(1, 2)$*.*

Unary Encoded Volume. When a system is volume-increasing or volume-decreasing, the reaction sequence length is polynomial in the volume of the system, so we achieve the following theorem.

▶ **Theorem 16.** *Reachability is in NP for volume-increasing and volume-decreasing CRNs when the volume is encoded in unary.*

Unimolecular Reactions. Unimolecular reactions are of the form $A \to B$, i.e. preserving rules of size 1. If we are limited to only this type of reaction, production is NL-complete. The NL-hardness result works for reachability as well.

▶ **Theorem 17.** *Production with rules of size* $(1, 1)$ *is NL-complete.*

We only show hardness for this case. The naïve non-deterministic algorithm does not work when the input values of the vectors are encoded in binary. We would need to track the entire configuration in-between each step. We leave exact membership for future work.

▶ **Theorem 18.** *Reachability and production in CRNs is NL-hard with rules of size* $(1, 1)$*.*

3.4 Results Related to Single-Consuming or Single-Source

Although the majority of the results in this section relate to rule size, we also look at rule restrictions based on the number of source and consuming rules. Our main result shows that reachability in a 2-consuming, 2-source general CRN system is PSPACE-complete. In Sections 4 and 5, we fully characterize these systems if the rules are feed-forward. Here, we address a few interesting results for CRNs that are not feed-forward.

Production is NP-Hard. Here, we state that production is NP-hard in single-consuming CRNs. We reduce from 3-SAT by creating species for the variables and clauses, as well as creating rules to assign variables and satisfy clauses.

▶ **Theorem 19.** *Production in single-consuming CRNs is NP-hard.*

Algorithm for $(k, 1)$ rules. Here, we state the existence of a polynomial-time algorithm for single-source single-consuming simple CRNs where each rule is of the form $a_1 + \cdots + a_k \to b$, and satisfies the requirements that $k \geq 2$ and $a_1, \cdots, a_k, b$ are different species.

▶ **Theorem 20.** *There is a polynomial-time algorithm to solve the reachability problem for single-source and single-consuming simple CRNs.*

4 Reachability in Feed-Forward CRNs

Having established PSPACE-completeness for general CRNs, we consider the feed-forward restriction, in which rule sets do not have cycles. Feed-forward CRNs are motivated in that they allow functional composition of CRNs [25]. We characterize the complexity of reachability for feed-forward CRNs under the assumption that the system does not contain either void or autogenesis rules, leaving a focused consideration of void and autogenesis rules for Section 5. Detailed proofs can be seen in Appendix Section B.

We first show NP-completeness of feed-forward systems in Section 4.1, even in the case of size-$(2, 2)$ rules (i.e., bimolecular rules / Population Protocols), while at the same time being only 2-source and 2-consuming. We then show a polynomial-time solution to reachability in Section 4.2 for any feed-forward system that is either 1-source or 1-consuming, thus giving a complete characterization of feed-forward reachability.

4.1 NP-completeness for Bimolecular Reactions

In this section, we show that reachability (and production) in feed-forward systems is NP-complete for bimolecular reactions (rules of size at most $(2, 2)$), even when the CRN is 2-source and 2-consuming. This hardness result is tight because a decrease to either 1-source or 1-consuming, as shown in Section 4.2, implies a polynomial time solution to reachability. We start with proof of membership in NP, followed by NP-hardness from a reduction from the Hamiltonian Path problem.

NP Membership. A key property of feed-forward systems is that any sequence of rule applications can be reordered such that all system rules are applied consecutively. This new ordering is still a valid sequence of applicable rules that reaches the same final configuration.

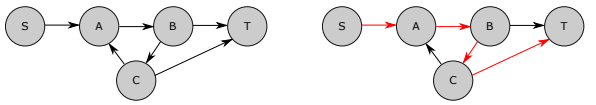
▶ **Definition 21** (Ordered Application)**.** *A sequence of reactions is an ordered application if all the applications of any given rule take place right after each other in a contiguous sequence. An example of this is $R_1, R_1, \ldots, R_1, R_2, \ldots, R_2, R_3$.*

▶ **Lemma 22.** *Let $C = (\Lambda, \Gamma)$ be a feed-forward CRN with a feed-forward ordering $F = \{R_0, R_1, \ldots, R_{|R|-1}\}$ over Γ. Given configurations c, c', and a sequence of reactions $S = \{\ldots, R_j, R_i, \ldots\}$ in Γ that converts c to c' where $i < j$, then the sequence $S' = \{\ldots, R_i, R_j, \ldots\}$, i.e., the sequence obtained by swapping the two rules R_i and R_j, also transforms $c \to c'$.*

▶ **Corollary 23.** *A configuration D is reachable from a configuration I with a feed-forward CRN if and only if D is reachable from I by an ordered application of rules.*

▶ **Lemma 24.** *The reachability problem is in NP for feed-forward CRNs that do not use autogenesis rules.*

NP-Hardness. We now show the reachability problem is NP-complete for feed-forward CRNs even for size $(2, 2)$-rules (bimolecular reactions, Population Protocols) and for 2-source, 2-consuming systems. We show this by a reduction from the Directed Hamiltonian Path



$$\begin{array}{lllll} \{S_0^* & A & B & C & T\} \\ \{S^v & A_1^* & B & C & T\} \\ \{S^v & A^v & B_2^* & C & T\} \\ \{S^v & A^v & B^v & C_3^* & T\} \\ \{S^v & A^v & B^v & C^v & T^4\} \end{array}$$

(a) Graph **(b)** Hamiltonian Path **(c)** Configurations

Figure 3 Our starting configuration $c = \{S_0^*, A, B, C, T\}$. Our goal configuration is $c' = \{S^v, A^v, B^v, C^v, T^4\}$. Each vertex must be changed to the visited state to reach the target, and the T must be the last vertex.

problem with vertices of in-degree and out-degree of at most 2 [21]. We reduce from bounded degree in order to achieve bounded source/consuming. For each vertex X in the graph $G = (V, E)$, we include $2 + |V|$ states: an initial state X, a visited state X^v, and $|V|$ signal states X_i^*. The additional signal states are added so the system is feed-forward. An example reduction is shown in Figure 3. We encode the edges of the example graph in the rules as follows,

$$\text{Rules } R = \left\{ \begin{array}{c|c|c} S_i^* + A \to S^v + A_{i+1}^* & B_i^* + C \to B^v + C_{i+1}^* & B_i^* + T \to B^v + T_{i+1}^* \\ A_i^* + B \to A^v + B_{i+1}^* & C_i^* + A \to C^v + A_{i+1}^* & C_i^* + T \to C^v + T_{i+1}^* \end{array} \right\}$$

▶ **Definition 25** (HAMPATH). *Given a graph $G = \{V, E\}$ and two nodes $s, t \in V$, does there exist a path from s to t that visits each node precisely once?*

Given this reduction, any Hamiltonian path of graph G has a corresponding sequence of rules that end with every vertex, other than T, represented with the *visited* state, and T represented with the signal state matching the count of the vertices. Conversely, the only way to reach such a configuration corresponds directly to a Hamiltonian path of G from S to T, yielding Theorem 26. With minor modifications, the reduction can be adapted to achieve the corollaries below.

▶ **Theorem 26.** *Reachability is NP-complete for feed-forward CRNs with size $(2, 2)$ rules that are 2-source and 2-consuming.*

▶ **Corollary 27.** *Reachability is NP-complete for feed-forward CRNs that are 2-source and 2-consuming with all rules of size $(2, 1)$.*

▶ **Corollary 28.** *Reachability is NP-complete for feed-forward CRNs that are 2-source and 2-consuming and with all rules of size $(1, 2)$.*

▶ **Corollary 29.** *Production is NP-complete for feed-forward CRNs with either size $(2, 2)$ rules, size $(2, 1)$ rules, or size $(1, 2)$ rules, that are 2-source and 2-consuming.*

Species-based Restrictions. For a CRN to be j-consuming, all species must be consumed in only j rules. We can relax this restriction to be j-consuming/k-source per species, meaning a rule is consumed in only j rules OR produced in only k rules. Each species is still bounded in one of the ways, but not both. We then show reachability, with the species-based restriction of k-consuming/1-source, is NP-hard by a reduction from the 3-Dimensional Matching (3DM) problem.

▶ **Definition 30** (Three Dimensional Matching Problem (3DM)). *The 3DM problem takes as input a hypergraph $H = (X, Y, Z, T)$ where X, Y, Z are three disjoint sets, and $T \subseteq X \times Y \times Z$ is a set of* hyperedges. *The output is whether or not there exists a subset of T that covers all vertices in H without any overlap.*

▶ **Corollary 31.** *Reachability in CRNs with each species being k-consuming/1-source or 1-consuming/k-source is NP-complete even with only one species being different than the others and the system being feed-forward without void/autogenesis rules.*

4.2 Feed-Forward, Single-Consuming/Single-Source

In this section, we establish Theorem 32 that shows the reachability problem is polynomial-time solvable for feed-forward, single-source rule sets that do not use void rules. We then extend this into Theorem 33 to show that reachability in feed-forward, single-consuming systems without autogenesis rules is also polynomial time solvable. Lastly, we give Corollary 34 that states the reachability problem for feed-forward, single-source, and single-consuming rule sets is polynomial-time solvable. Additional explanation and proofs are in Appendix B.2.

▶ **Theorem 32.** *The reachability problem is solvable in polynomial time for a rule set* Γ *that is feed-forward, single-source, and without void rules.*

▶ **Theorem 33.** *The reachability problem is solvable in polynomial time for a ruleset* Γ *that is feed-forward, single-consuming, and without autogenesis rules.*

▶ **Corollary 34.** *The reachability problem is solvable in polynomial time for a rule set* Γ *that is feed-forward, 1-source, and 1-consuming with no further restrictions on the rule set.*

5 Void and Autogenesis Rules

In our consideration of feed-forward CRNs, we omitted two classes of rules: *void* rules that consume reactants without creating any products and *autogenesis* rules that create products without consuming any reactants. One reason for separating these rules is that their lack of conservation of mass might mean they are not feasible in some experimental settings. Another important reason is that their inclusion alone substantially impacts the complexity of problems such as reachability.

Here, we explore reachability in the scenario where *all* rules are void rules or, conversely, all rules are autogenesis rules. We show that void rules (or autogenesis rules) alone imply the NP-completeness of reachability, even if such systems are both feed-forward and 0-source. We specifically show NP-completeness for size $(3, 0)$ void rules. We then explore the complexity of reachability with size $(2, 0)$ void rules and provide a polynomial time solution when the system's volume is encoded in unary. We further show a polynomial time solution for binary encoded volume for a restricted class of *bipartite* $(2, 0)$ CRNs. We leave the remaining general case of reachability with $(2, 0)$ void rules as an open question. We note by Definition 52, that we may prove results for void only rules, and they are equivalent for autogenesis rules. Detailed proofs can be found in the Appendix Section C.

Size $(3, 0)$ **Void Rules /** $(0, 3)$ **Autogenesis Rules.** We show that reachability is NP-complete by a reduction from 3-Dimensional Matching (3DM) (Definition 30).

▶ **Theorem 35.** *Reachability for CRNs with only rules of size* $(3, 0)$ *is NP-complete.*

▶ **Corollary 36.** *Reachability for CRNs with only rules of size* $(0, 3)$ *is NP-complete.*

Size $(2, 0)$ **rules with Unary Encoding.** As with $(3, 0)$ rules, $(2, 0)$ may also be reduced by matching. This time bipartite.

▶ **Theorem 37.** *Reachability in CRNs is in P with rules of size* $(2, 0)$ *if configuration counts are encoded in unary.*

Size $(2, 0)$ **rules with Binary Encoding** We now consider $(2, 0)$ with binary encoded species counts, which permits a potentially exponential configuration volume, making the algorithm of Theorem 37 no longer polynomial time. In this scenario, we consider a new restriction in which the CRN rules are bipartite:

▶ **Definition 38** (Bipartite CRN). *A bipartite CRN* (Λ, Γ) *is one in which the species* Λ *can be partitioned into two disjoint sets* Λ_1 *and* Λ_2 *such that for each rule* $R \in \Gamma$*, there are at most 2 reactants of* R *and they do not occur within the same partition of* Λ*.*

Determining if a CRN is bipartite can be solved in polynomial time by a bipartite graph detection algorithm. If the CRN is bipartite, we reduce the problem to the maximum flow problem. Although this algorithm only works with bipartite CRNs, we conjecture that the problem is in P, and leave it as an important open question related to general matching in weighted graphs.

▶ **Theorem 39.** *Reachability is polynomial-time solvable for bipartite CRNs with* $(2, 0)$ *rules.*

▶ **Conjecture 40.** *Reachability is polynomial-time solvable for CRNs with* $(2, 0)$ *rules.*

6 Conclusion

With the complexity of the general reachability problem solved recently, this paper resolves several restricted cases of the problem. We prove hardness for several open problems or improve the known results. These include answering reachability in Population Protocols, showing that non-increasing volume CRNs are PSPACE-complete with rules of size two (improved from 5), proving that feed-forward systems are NP-complete and giving a polynomial algorithm if it is single-source or single-consuming without void/autogenesis rules, and showing how void and autogenesis rules affect the complexity of feed-forward systems. Additionally, we provide several other results related to these restrictions.

Related Problems. While we give many results, this is by no means the end of the investigation into the computational complexity of restricted Chemical Reaction Networks, as this work can be extended in multiple ways. Do the reachability and production problems ever have different complexities for the same version of the model? When is universal reachability a harder problem than reachability? What about the version of production where we want to produce k copies of a given species rather than just a single copy?

Reachability. For a complete characterization of reachability for all parameters, here we note some open problems and future directions of investigation.

- Reachability for 1-consuming *and* 1-source CRNs with the feed-forward property has membership in P. Is the feed-forward restriction required for this result, or does it hold for a 1-consuming and 1-source proper CRN as well?
- We show the problem of reachability is PSPACE-complete with rules of size $(2, 2)$ (as previously shown in [13]). Is reachability also hard with smaller rule sizes? Can we achieve the same result with $(2, 1)$ rules and a binary encoded volume?
- What is the smallest catalytic (2,2) system that is PSPACE-complete?
- Production with $(1, 1)$ rules is NL-complete. However, we do not know if reachability is easy as well. Is there a polynomial time algorithm to decide unimolecular reactions?
- With a non-monotone volume, we no longer have membership in PSPACE- we only have an Ackermann upper bound. With constant rule size, is reachability still Ackermann-hard?
- Reachability for $(2, 0)$ rules with the volume encoded in binary is an open problem. This is a generalized matching problem on a weighted graph. We know the problem is in P for bipartite CRNs or with unary volume. Is the problem still easy in the general case?

References

1. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, page 290–299, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1011767.1011810.
2. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distribed Computing*, 18(4):235–253, mar 2006. doi:10.1007/s00446-005-0138-3.
3. Joshua Ani, Erik D Demaine, Yevhenii Diomidov, Dylan Hendrickson, and Jayson Lynch. Traversability, reconfiguration, and reachability in the gadget framework. In *Intl. Conf. and Work. on Algorithms and Comp.*, 2022.
4. Rutherford Aris. Prolegomena to the rational analysis of systems of chemical reactions. *Archive for Rational Mechanics and Analysis*, 19(2):81–99, jan 1965. doi:10.1007/BF00282276.
5. Rutherford Aris. Prolegomena to the rational analysis of systems of chemical reactions ii. some addenda. *Archive for Rational Mechanics and Analysis*, 27(5):356–364, jan 1968. doi:10.1007/BF00251438.
6. Michael Blondin, Matthias Englert, Alain Finkel, Stefan Göller, Christoph Haase, Ranko Lazić, Pierre Mckenzie, and Patrick Totzke. The reachability problem for two-dimensional vector addition systems with states. *Journal of the ACM (JACM)*, 68(5):1–43, 2021.
7. 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 '14, page 313–326, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2554797.2554827.
8. Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. *Programmability of Chemical Reaction Networks*, pages 543–584. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-540-88869-7_27.
9. Wojciech Czerwiński and Łukasz Orlikowski. Reachability in vector addition systems is ackermann-complete. In *62nd Annual Symposium on Foundations of Computer Science*, FOCS'21, pages 1229–1240. IEEE, 2021.
10. Erik D Demaine, Isaac Grosof, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In *9th Intl. Conf. on Fun with Algorithms (FUN 2018)*, 2018.
11. Erik D Demaine, Robert A Hearn, Dylan Hendrickson, and Jayson Lynch. Pspace-completeness of reversible deterministic systems. *arXiv preprint arXiv:2207.07229*, 2022.
12. Erik D Demaine, Dylan H Hendrickson, and Jayson Lynch. Toward a general complexity theory of motion planning: Characterizing which gadgets make games hard. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, 2020.
13. Javier Esparza, Mikhail Raskin, and Chana Weil-Kennedy. Parameterized analysis of immediate observation petri nets. In *Application and Theory of Petri Nets and Concurrency*, pages 365–385. Springer International Publishing, 2019. doi:10.1007/978-3-030-21571-2_20.
14. Bin Fu, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Reachability in population protocols. In *The Japan Conference on Discrete and Computational Geometry, Graphs, Games (JCDCG^3)*, 2022.
15. John Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.
16. Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi:https://doi.org/10.1016/S0022-0000(69)80011-5.
17. Jérôme Leroux. The reachability problem for petri nets is not primitive recursive. In *62nd Annual Symposium on Foundations of Computer Science*, FOCS'21. IEEE, 2021.
18. Richard J. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.

19 Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, page 238–246, New York, NY, USA, 1981. Association for Computing Machinery. doi:10.1145/800076.802477.

20 Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, 1962.

21 J Plesník. The np-completeness of the hamiltonian cycle problem in planar diagraphs with degree bound two. *Information Processing Letters*, 1979.

22 Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.

23 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.

24 Chris Thachuk and Anne Condon. Space and energy efficient computation with dna strand displacement systems. In *International Workshop on DNA-Based Computers*, 2012.

25 Marko Vasić, Cameron Chalk, Austin Luchsinger, Sarfraz Khurshid, and David Soloveichik. Programming and training rate-independent chemical reaction networks. *Proceedings of the National Academy of Sciences*, 119(24):e2111552119, 2022.

A Appendix for Section 3

A.1 Gadget Reconfiguration Framework

Directed Wire. Two ports are connected by a wire. We label each wire with a symbol such as a and include two species $\overrightarrow{a}, \overleftarrow{a}$ to denote the direction moved along the wire. We refer to these as the agent species.

Toggle-Lock and Rotate Gadgets. We have two species for each Toggle-Lock that we call the *gate catalysts*. We represent the unlocked gate using the species G, and the locked state with species G'. Traversing the toggle tunnel requires the gadget to be in the correct state for the toggle. The reaction changes the state of the gate catalyst and the agent species. The bottom tunnel can only be traversed if the gate is in the unlocked state. However, this does not change the state of the gate, so the G species acts a catalyst that must be present to traverse the gadget. Figures 2c and 2d shows the rules used to implement a toggle lock.

We implement rotate gadgets with a single rotate clockwise species $r_{\circlearrowright}$. The rotate gadget diagram is shown in Figure 2e. The signal state changes to the outgoing direction of the next wire in the clockwise ordering of ports using the rules shown in Figure 2f. Note that each reaction consumes and creates exactly one agent species yielding the following observation:

▶ **Observation 41.** *Any reachable configuration in the reduction only contains a single agent species.*

▶ **Theorem 12.** *Production in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.*

Proof. Membership in PSPACE for Proper CRNs was shown in [24]. Given an instance of the motion planning problem with toggle-locks and rotate gadgets, create a CRN ruleset and configuration as described above. Our starting configuration of the CRN encodes the start location and starting states of the gadgets. The target species we wish to produce is the agent species for the target location in the motion planning problem.

The agent may only traverse a toggle-lock gadget if the gate catalyst is in the correct state. The rotate gadget sends the signal to the correct next state. Once the agent reaches the target wire, the agent species representing it is produced. If the agent reaches the target location, then reactions apply representing the path of the agent to produce the target species.

From Observation 41, there only exists one agent species and the reactions encode only valid traversals through the gadgets. If the target species is produced, then the sequence of reactions to produce it represents the path of the agent though the system of gadgets. Each gate catalyst is only produced and consumed in the reaction that implements the toggle tunnel. Each agent species is directed, so it also is only produced and consumed in one rule.

Note that if an agent ever attempts to cross a gadget in a state that does not allow that traversal, the configuration will no longer have any valid reactions. This is the equivalent to an illegal move in the motion planning problem and the agent cannot progress, so the game is over. ◀

▶ **Theorem 13.** *Reachability in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.*

Proof. The reduction from Theorem 12 can be extended to show the reachability problem is PSPACE-hard as well. We add an additional species $r_{\circlearrowleft}$ which is the rotate counterclockwise catalyst. The target configuration is the same as the initial configuration except with the

counterclockwise catalyst $r_{\circlearrowleft}$. If the target wire is a, we add the rule $\overrightarrow{a} + r_{\circlearrowleft} \to \overleftarrow{a} + r_{\circlearrowleft}$, which changes the direction of the rotate catalyst and turns the agent around on the wire. Since the toggle lock gadget is reversible, the agent will undo all of its moves and return to the start configuration. Since the gadget system has the property of being reversible, this backwards traversal is possible. ◀

▶ **Corollary 14.** *Universal reachability in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.*

Proof. From Observation 41, we know there only exists one agent state in the system at a time. Since the system is single-consuming, the agent state is only consumed in a single rule. These two points mean there only exists a single move sequence, so if the target configuration is reachable, it is universally reachable. ◀

A.2 Volume Related Results

▶ **Corollary 15.** *Production and reachability in 2-source, 2-consuming CRNs is PSPACE-hard with rules of size* $(2, 1)$ *and* $(1, 2)$.

Proof. The reduction behaves the same way as in Theorems 12 and 13, however, here, we either have a single agent species, or a single intermediate species. ◀

▶ **Theorem 16.** *Reachability is in NP for volume-increasing and volume-decreasing CRNs when the volume is encoded in unary.*

Proof. When the volume of a system is strictly increasing or decreasing, any reaction sequence between I and D is bounded by of size $\leq |I - D|$. The sequence can then be given as a 'yes' certificate for reachability. ◀

▶ **Theorem 18.** *Reachability and production in CRNs is NL-hard with rules of size* $(1, 1)$.

Proof. Given a directed graph G we create a set of species and reactions as follows. For each node $v \in G$ we create a species. For each edge $(a, b) \in E$ we create a reaction $a \to b$.

We reduce from the directed path problem: given two nodes $s, t \in G$, does there exist a path between s and t? Let our initial configuration be a single copy of the species representing s and our target configuration be a single copy of t. At each step the species will represent the current node in the path to reach t if and only if there exists a path. ◀

▶ **Theorem 17.** *Production with rules of size* $(1, 1)$ *is NL-complete.*

Proof. Non-deterministically select a species i with a positive count in the initial configuration, check if the target species is reachable from i, if yes then the target species is producible. Checking reachability is in NL, since the reactions can be viewed as directed edges. NL-hardness comes from Theorem 18. ◀

A.3 Results Related to Single-Consuming or Single-Source

▶ **Theorem 19.** *Production in single-consuming CRNs is NP-hard.*

Proof. Consider an instance of 3SAT where X is the set of variables and C is the set of clauses. For each $x_i \in X$ we include $x_i, \overline{x_i}, x_i^T, x_i^F$ in our species set. The starting configuration includes a copy of each of the species $x_i, \overline{x_i}$. For each $c_j \in C$ with $c_j = (x_a, x_b, x_c)$, we include $c_j^a, c_j^b, c_j^c, c_j^{SAT}, SAT_j$. The starting configuration includes a copy of each species c_j^a, c_j^b, c_j^c and SAT_0. The target species we wish to produce is $SAT_{|C|}$.

Two assignments species T, F are included. One copy of each of these species is included in the starting configuration that acts as a catalyst. There are two rules for each x_i: $T + x_i + \overline{x_i} \to T + x_i^T + \overline{x_i}$ for assigning true, and $F + x_i + \overline{x_i} \to F + x_i + x_i^F$ for assigning false. The T and F catalysts are used in order to not have two rules with the same reactants. The rules change one of the species to the true or the false species, and the other can not be created. We add a rule $c_j^a + x_a^T \to c_j^{SAT} + x_a^T$ for positive literals and $c_j^a + x_a^F \to c_j^{SAT} + x_a^F$ for negated literals. To verify each clause we include the rules, $SAT_j + c_j^{SAT} \to SAT_{j+1}$

The target $SAT_{|C|}$ is only producible by starting with SAT_0 and applying the rule $SAT_j + c_j^{SAT} \to SAT_{j+1}$ exactly $|C|$ times to verify the satisfaction of each clause. ◀

A.3.1 A Polynomial-Time Algorithm for Single-Source and Single-Consuming Simple CRNs

In this section, we present a polynomial-time algorithm for the reachability problem of CRNs where each species has at most one source and is at most one consuming, and each rule generates at most one species.

Preliminaries. A CRN(Λ, Γ) is a $(1, 1, 1)$-CRN if it is 1-source and 1-consuming, and each rule $a_1 + \cdots + a_k \to b$ satisfies that $k \geq 2$ and the species $a_1, \cdots, a_k, b$ are different from each other. A $(1, 1, 1)$-CRN is also called *single source and single consuming simple* CRN.

Let $G = (V, E)$ be a directed graph with $|V| = n$. A directed graph $G = (V, E)$ is weakly connected if the undirected graph $G' = (V, E')$ is connected, where E' is the set of undirected edges (u, v) with $u \to v$ as a directed edge in $G = (V, E)$. A vertex v is a *leaf* in $G = (V, E)$ if there is one edge $u \to v$ coming to it. A vertex v is an *almost leaf* if every edge $u \to v$ in E has u as a leaf.

Let (Λ, Γ) be a $(1, 1, 1)$-CRN. Let each species of the CRN be a vertex in a directed graph $G = (V, E)$. There is an edge $u \to v$ if there is a rule that has u on the left and v on the right side. We say $G = (V, E)$ is *derived* from the CRN (Λ, Γ).

The reachability problem for a directed graph $G = (V, E)$ derived from a CRN (Λ, Γ), is to find a way to apply rules to reach configuration $(j_1, \cdots, j_n)$ from the starting configuration $(i_1, \cdots, i_n)$, where vertex t has i_t copies in the beginning and j_t copies in the end.

Let $G = (V, E)$ be directed a graph. We say G is an *almost-cycle* if it contains a unique directed cycle C and every vertex has an edge directed to a vertex in the cycle C.

▶ **Lemma 42.** *Let $G = (V, E)$ be derived from a $(1, 1, 1)$-CRN (Λ, Γ) and be weakly connected. Then $G = (V, E)$ has at most one directed cycle. Furthermore, if $G = (V, E)$ does not have a directed cycle, then it is a directed tree with a root that every vertex has a directed path to. If $G = (V, E)$ does have a directed cycle, then every vertex has a directed path to a vertex in the cycle.*

Proof. This is because (Λ, Γ) is a $(1, 1, 1)$-CRN. Assume there are two different directed cycles C_1 and C_2 in G. We discuss two cases.

- Case 1. There is a common vertex v in both C_1 and C_2. We have that v has two outgoing edges from v. This contradicts that G is derived from a $(1, 1, 1)$-CRN since this implies the CRN is 2-consuming.
- Case 2. There is a common vertex between C_1 and C_2. We can find a weak path connecting C_1 and C_2, where a weak path is a set of directed edges in G and form a regular path when they are converted into undirected edges. One vertex on the weak path must have two outgoing edges. This contradicts that G is derived from a $(1, 1, 1)$-CRN since this implies the CRN is 2-consuming.

Therefore, G has at most one directed cycle, or a single vertex graph. If it is an almost-cycle, we are done with the proof.

Now assume that G is neither an almost-cycle nor a single vertex graph. Let $v \to v_1 \to \cdots \to v_k$ be a longest directed path without any vertex in the directed cycle. We have that v_1 is an almost leaf. Otherwise, the directed path would not be the longest. Assume that v_1 is generated by $v, u_1, \cdots, u_t$ in (Λ, Γ) (it has a rule $v + u_1 + \cdots + u_t \to v_1$). We transform $G = (V, E)$ into $G' = (V \setminus \{v, u_1, \cdots, u_t\}, E \setminus \{v \to v, u_1 \to v, \cdots, u_t \to v\})$. G' is derived from another $(1, 1, 1)$-CRN (Λ', Γ') that is transformed from (Λ, Γ) by removing the rule $v + u_1 + \cdots + u_t \to v$ and $v, u_1, \cdots, u_t$ from (Λ, Γ). This is proven via a simple induction. ◀

▶ **Lemma 43.** *Let $G = (V, E)$ be derived from a $(1, 1, 1)$-CRN (Λ, Γ). Let u be an almost leaf (not a vertex on a directed cycle) in $G = (V, E)$. Then the reachability problem for $G = (V, E)$ can be transformed into $G' = (V \setminus U, E \setminus E(U))$, where U is the set of vertices v entering u ($(v \to u) \in E$) and $E(U)$ is the set of edges ($(v \to u) \in E$) entering u in $G = (V, E)$.*

Proof. For an almost leaf u not in a cycle, there is a unique way to apply the rules in order to reach the target configuration. If u has i_1 copies in the input configuration, j_1 copies in the target configuration, we must have $i_1 \leq j_1$. Otherwise, the target configuration is not reachable. If rule $v_1 + \cdots + v_t \to u$ is the only one to generate u, then we have to apply these rules $j_1 - i_1$ times. After applying this rule $j_1 - i_1$ times, we check if the number of copies of those $v_1, \cdots, v_t$ are equal to their target values, respectively. ◀

▶ **Lemma 44.** *There is a polynomial-time algorithm to solve the reachability problem for an almost-cycle graph derived from a $(1, 1, 1)$-CRN.*

Proof. Let $G = (V, E)$ be an almost-cycle graph with a unique C in it. Let v be a vertex in C. The vertex v appears in a rule $R : v + v_1 + \cdots + v_t \to u$. It is easy to see that none of $v_1, \cdots, v_t$ is the cycle C. If v_i is in C, we have u with both v and v_i going toward it in C, which is a contradiction. Thus, the number of copies of each v_i is decreasing. Therefore, we know the number of times the rule R should be used. For each v in C, we compute the number of copies (C_v) that v is consumed and the number (P_v) of copies v are produced. The difference $C_v - P_v$ is how many times (noted g_v) we enter and leave v.

Assume that the target configuration is reachable from the current configuration. Let it start from a vertex v in the cycle C. If it does not finish the first g iterations, we can adjust the operations until it finishes the first g iterations. Assume that the broken point is at vertex v'.

We can decide the number of iterations to apply rules along the cycle C. It will be $g = \min\{g_v\}$ if $g_v > 0$ for each vertex v in C. Without loss of generality, let $g = g_{v_1}$.

Assume that the target configuration is reachable from the current configuration. Let it start from a vertex v_i in the cycle C. If it does not finish the first g iterations, we can adjust the operations so that it finishes the first g iterations along the cycle C. Assume that the broken point is at vertex v_j. The next operation is at a vertex v_k with $|j - k| > 1$. We apply the rules $R_1, R_2 \cdots, R_{s-1}, R_s$, where R_s is the rule to generate v_{j+1}. The operations can be transformed to $R_s, R_1, R_2 \cdots, R_{s-1}$.

After applying the rules along the cycle C with g iterations, we compute $g'_v = g_v - g_{v_1}$, $C'_v = C_v - g_{v_1}$ and $P'_v = P_v - g_{v_1}$ for all vertices v in C. We have $g'_{v_1} = 0$. Let $v_1, v_2, \cdots, v_n$ be the n vertices in the cycle C according the direction of the cycle. For $i = 2, 3, \cdots, n, 1$, we apply the rule P'_{v_i} times to generate v_i, and another rule P_{v_i} times to produce v_i. At the end, we check if we fail or reach the target configuration. Thus, we have a polynomial-time algorithm for the reachability of $(1, 1, 1)$-CRNs. ◀

▶ **Theorem 20.** *There is a polynomial-time algorithm to solve the reachability problem for single-source and single-consuming simple CRNs.*

Proof. Let $(i_1, \cdots, i_n)$ be the configuration of the input species, where i_t is the number of copies of the species t. Let $(j_1, \cdots, j_n)$ be the target configuration of species in the end of process. We need to find a polynomial-time solution for each weakly connected component. Without loss of generality, we assume G only has one weakly connected

- Case 1. There is no directed cycle in G. By Lemma 43, we can shrink the graph G until it becomes a trivial case.
- Case 2. There is one directed cycle C in G. By Lemma 43, we can shrink the graph G until it becomes an almost-cycle, and then by Lemma 44, we know reachability is decidable in polynomial-time.

Therefore, we have a polynomial-time algorithm for the reachability problem for $(1, 1, 1)$-CRNs. ◀

B Appendix for Section 4

B.1 NP-completeness for Bimolecular Reactions

▶ **Lemma 22.** *Let $C = (\Lambda, \Gamma)$ be a feed-forward CRN with a feed-forward ordering $F = \{R_0, R_1, \ldots, R_{|R|-1}\}$ over Γ. Given configurations c, c', and a sequence of reactions $S = \{\ldots, R_j, R_i, \ldots\}$ in Γ that converts c to c' where $i < j$, then the sequence $S' = \{\ldots, R_i, R_j, \ldots\}$, i.e., the sequence obtained by swapping the two rules R_i and R_j, also transforms $c \to c'$.*

Proof. Let X denote the configuration obtained by applying the rules of S up to just before the application of rule R_j. Let $R_j = (R_r^j, R_p^j)$ and $R_i = (R_r^i, R_p^i)$. As S is a valid sequence of rule applications, $X - R_r^j$ is a non-negative vector. Due to the feed-forward ordering F, the reactants R_r^i do not occur as products in R_p^j, and thus $X - R_r^j - R_r^i$ is also non-negative, implying that rule R_i and R_j can be applied in either order. ◀

▶ **Corollary 23.** *A configuration D is reachable from a configuration I with a feed-forward CRN if and only if D is reachable from I by an ordered application of rules.*

▶ **Lemma 24.** *The reachability problem is in NP for feed-forward CRNs that do not use autogenesis rules.*

Proof. We utilize the ordered application from Corollary 23 as a polynomial-sized certificate. Denote this certificate as $A = \langle a_1, a_2, a_3, \ldots, a_k \rangle$ where each a_i is an nonnegative integer denoting the number of applications of rule R_i in the feed-forward ordering. While sequence A could include a large (exponential) number of total rule applications, we bound the number of applications by bounding the volume of the system at any given point during the application of these rules, which both shows that the sequence can be represented in size polynomial in the input size n of the reachability problem, and verified in polynomial time, thus showing membership in NP.

Let V_i denote the volume of the CRN after the complete application of all a_i instances of rule R_i in the feed-forward ordering, with $V_0 = V(I)$ denoting the volume of the initial given configuration I. Note that since there are no autogenesis rules, we know that $a_i \leq V_{i-1}$, since each rule must exhaust at least one count of some species in the system. Let $V(p_i)$ denote the volume of the product of the i^{th} rule in the feed-forward ordering, and let $V(p)$ denote the largest $V(p_i)$ plus 1, noting that each application of R_i increases the system

volume by at most $V(p_i) \leq V(p)$. Thus, $V_i \leq V_{i-1} + a_i V(p_i) \leq V_{i-1} + V_{i-1}V(p_i) \leq V_{i-1}V(p)$. This recurrence equation solves to $V_i \leq V(0)V(p)^i$. If we let n denote the input size to the reachability problem, we know that $V(0) \leq 2^n$, and $V(p) \leq 2^n$, and therefore $V_i \leq 2^n \cdot 2^{i \cdot n} \leq 2^{n^2+n}$. Thus, the total volume of the system at any point, as well as each a_i, can be represented in a polynomial number of bits in n. We make a nondeterministic path to guess nonnegative integers $a_1, a_2, \cdots a_n$ in the $[0, 2^{n^2+n}]$, verify the system has enough reactants to support a_i applications of each rules r_i in the sequence, and compute the configuration $C = C_0 + a_1 H_1 + a_2 H_2 \cdots + a_k H_k$, where C_0 is the initial configuration vector, and H_i is the application vector of rule R_i (see Section 2). Finally, we check if C is the same as the target configuration. Since all integers involved in these calculations are stored using a polynomial number of bits, this computation can be performed in polynomial time. ◀

▶ **Lemma 45.** *The reachability problem is in NP for feed-forward CRNs that do not use void rules.*

Proof. This follows from Lemma 24 and Lemma 53. ◀

▶ **Theorem 26.** *Reachability is NP-complete for feed-forward CRNs with size* $(2, 2)$ *rules that are 2-source and 2-consuming.*

Proof. If there exists a Hamiltonian path P in G, there exists a sequence of rules r_P, each moving the * to a subsequent agent matching the sequence of vertices in P and further setting the previous agent to the visited state. Such a rule sequence ends with a single visited state for each vertex in the graph other than T and a $T_{|V|}$ state.

Conversely, if a sequence of rules r that results in the target configuration exists, then a Hamilton path exists as each rule changes the agent location to the visited state, which can no longer change state, and moves the * to the next agent. The sequence of * species in the configuration represents the Hamilton path.

Finally, note that this system is feed-forward since an agent starts in the initial state, changes to a signal state, then to the visited state. We can ensure the ordering since the signal states are numbered. ◀

▶ **Corollary 27.** *Reachability is NP-complete for feed-forward CRNs that are 2-source and 2-consuming with all rules of size* $(2, 1)$.

Proof. This is obtained by simply removing the visited states in the previous reduction. ◀

▶ **Corollary 28.** *Reachability is NP-complete for feed-forward CRNs that are 2-source and 2-consuming and with all rules of size* $(1, 2)$.

Proof. This follows from the above corollary combined with Lemma 53 (defined in Section 4.2). ◀

▶ **Corollary 29.** *Production is NP-complete for feed-forward CRNs with either size* $(2, 2)$ *rules, size* $(2, 1)$ *rules, or size* $(1, 2)$ *rules, that are 2-source and 2-consuming.*

Proof. In the previous reduction, the species $T_{|V|}$ is only producible if it is reached after reaching all other species, implying the previous reduction above holds for the production problem. ◀

▶ **Definition 30** (Three Dimensional Matching Problem (3DM)). *The 3DM problem takes as input a hypergraph $H = (X, Y, Z, T)$ where X, Y, Z are three disjoint sets, and $T \subseteq X \times Y \times Z$ is a set of* hyperedges. *The output is whether or not there exists a subset of T that covers all vertices in H without any overlap.*

▶ **Corollary 31.** *Reachability in CRNs with each species being k-consuming/1-source or 1-consuming/k-source is NP-complete even with only one species being different than the others and the system being feed-forward without void/autogenesis rules.*

Proof. We reduce from the 3DM problem, which is hard even when each vertex is covered by at most 3 hyper edges. Let $H = (X, Y, Z, T)$ be an input to the 3DM problem. From this, create an input to the reachability problem as follows. Let $\Lambda = \{S_v | v \in X \bigcup Y \bigcup Z\} \bigcup \{a\}$, and let $\Gamma = \{S_x + S_y + S_z \to a | (x, y, z) \in T\}$. The initial configuration I is the configuration in which each species has count 1 except species a with count 0. Let D be the configuration in which each species has count 0 except a, which has count $|X| = |Y| = |Z|$. Then D is reachable from I under (Λ, Γ) if and only if H has a three-dimensional matching.

Species a is never consumed, but is produced by all $k = |X| = |Y| = |Z|$ of the rules. Thus, it is 0-consuming/k-source. All other species are never produced, and are consumed in 3 different rules. Thus, they are 3-consuming/0-source. Finally, since species A is never consumed, any ordering of the rules is feed-forward, and there are no void or autogenesis rules used. By Lemma 24, the problem is in NP. ◀

‘

B.2 Feed-Forward, Single-Consuming/Single-Source

We now introduce some machinery for constructing an efficient algorithm for solving reachability in the special case of feed-forward, single-source rule sets with no void rules.

▶ **Definition 46** (Leaf Rule.). *A rule $R \in \Gamma$ is a* leaf rule *for Γ if the products of R do not occur as reactants within any* other *rule of the system Γ (however, the products of a leaf rule may occur as reactants within the same leaf rule). A leaf rule could also be a void rule.*

▶ **Lemma 47.** *A feed-forward, non-empty rule set has at least one leaf rule.*

Proof. The final rule in the feed-forward ordering must be a leaf rule, as its product may not occur as a reactant for any previous rule of the system, which includes all rules other than itself. ◀

▶ **Lemma 48.** *If Γ is a feed-forward, single-source rule set without void rules, then so is any subset of Γ.*

Proof. This follows from the definitions of feed-forward, single-source, and void rules. ◀

▶ **Definition 49** (Pruned Configuration.). *Consider a configuration I, a target configuration D, and feed-forward rule set Γ with non-void leaf rule $R = (R_r, R_p)$. If there exists a non-negative integer x such that $D(i) \setminus xR_a(i) = I(i)$, for all i where $R_p(i) \neq 0$, and R is applicable to $D \setminus xR_a$, we say that D is prunable towards I with respect to rule R, and define the* pruning *of D towards I with respect to R to be the configuration $\texttt{Prune}(D, I, R) = D \setminus xR_a$. If no such non-negative integer x exists, then we say that D is inconsistent with I for rule R. Note that the integer x, and thus the configuration $\texttt{Prune}(D, I, R) = D \setminus xR_a$, are unique as long as R is not a void leaf rule.*

▶ **Lemma 50.** *If a configuration D is inconsistent with a configuration I for any leaf rule $R \in \Gamma$, then D is not reachable from I with a rule set Γ.*

Proof. As R is a leaf rule, the counts of the product species in rule R are only affected by rule R among the rules of Γ. Therefore, if D is reachable from I, it must be possible to generate the counts of these species specified by D by some number of applications x of rule R. If no such integer exists, which is the definition of inconsistent, then the species counts for R's products cannot equal the counts specified by D, making D unreachable. ◀

▶ **Lemma 51.** *For a rule set Γ and configuration D that is consistent with I for non-void leaf rule $R \in \Gamma$, then D is reachable from I with a rule set Γ if and only if $D' = \texttt{Prune}(D, I, R)$ is reachable from configuration I with a rule set $\Gamma \setminus R$.*

Proof. We first show that if $D' = \texttt{Prune}(D, I, R)$ is reachable with a rule set $\Gamma \setminus R$, then so is D with a rule set Γ. This is because once D' is reached, we know from the definition of $D' = \texttt{Prune}(D, I, R)$ that there exists a non-negative integer x such that x applications of rule R to configuration D' yields D, implying that D is reachable.

For the other direction, suppose we can reach D. From the sequence of rule applications that reaches D, there must be exactly some non-negative integer x applications of rule R. Create a modified sequence of configurations by omitting these x operations, and you get configuration $D' = \texttt{Prune}(D, I, R)$ by application of rules from $\Gamma \setminus R$, meaning D' is reachable from $\Gamma \setminus R$. ◀

▶ **Theorem 32.** *The reachability problem is solvable in polynomial time for a rule set Γ that is feed-forward, single-source, and without void rules.*

Proof. Let I denote the starting configuration, D denote the destination configuration, and (Λ, Γ) be a feed-forward CRN without void rules for a given reachability instance. The following recursive algorithm solves reachability for a feed-forward, single-source rule set with no void rules.

As a base case, if the input system has 0 rules, then D is reachable if and only if $I = D$. Otherwise, identify a leaf rule R from Γ, which must exist by Lemma 47. Check if D is consistent with I for rule R. If not, return false, which is the correct answer by Lemma 50. If it is, then let $D' = \texttt{Prune}(D, I, R)$ denote the pruning of D towards I for rule R and return the result of recursively solving reachability with initial configuration I, ruleset $\Gamma \setminus R$, and destination configuration D', which is a valid input to this algorithm as $\Gamma \setminus R$ is assured to be a feed-forward, single-source rule set without void rules by Lemma 48, and is assured to yield the correct result by Lemma 51. In total, this algorithm executes $|\Gamma|$ prune operations. Further, as the system is both feed-forward and without void rules, a polynomial number of prune operations preserves the property that the count of any system species after each pruning is small enough to be represented in only a polynomial number of bits (the argument for this is the same as the proof of Lemma 24). Therefore, the polynomial number of arithmetic operations used to compute a prune can each be completed in polynomial time, and so each prune can be performed in polynomial time, and therefore the algorithm overall finishes in polynomial time. ◀

▶ **Definition 52.** *For a rule set Γ, let $\overleftarrow{\Gamma}$ be the* reverse *of Γ where $\overleftarrow{\Gamma} = \{(a, b) | (b, a) \in \Gamma\}$.*

▶ **Lemma 53.** *For any two configurations A, B and rule set Γ, B is reachable from A in Γ if and only if A is reachable from B in $\overleftarrow{\Gamma}$.*

We now consider the case of reachability in feed-forward, single-consuming systems without autogenesis rules. Our approach is to *reverse* the given rule set Γ and apply our algorithm for Theorem 32.

▶ **Theorem 33.** *The reachability problem is solvable in polynomial time for a ruleset Γ that is feed-forward, single-consuming, and without autogenesis rules.*

Proof. Given a CRN (Λ, Γ) where Γ is feed-forward, single-consuming, and without autogenesis rules, along with initial configuration I, and destination D, generate rule set $\overleftarrow{\Gamma}$. Note that $\overleftarrow{\Gamma}$ must be single-source as Γ is single-consuming, and must have no void rules since Γ has no autogenesis rules, and must be feed-forward since Γ is feed-forward. We can therefore determine if I is reachable from D under $\overleftarrow{\Gamma}$ in polynomial time by Theorem 32, which gives the answer to our original reachability problem by Lemma 53. ◀

Here we consider the case of reachability in feed-forward, 1-source, and 1-consuming systems with no further restrictions on the rule set. A single-consuming rule set will contain at most one void rule, allowing void rules to be considered when pruning a configuration.

▶ **Corollary 34.** *The reachability problem is solvable in polynomial time for a rule set Γ that is feed-forward, 1-source, and 1-consuming with no further restrictions on the rule set.*

Proof. Let I denote the initial configuration, let D denote the target configuration, and let (Λ, Γ) be a feed-forward, 1-source, and 1-consuming CRN for a given reachability instance. With a single-consuming rule set Γ, any rule R, void or otherwise, may be used to prune D if there exists some non-negative integer x such that $D(i) \setminus xR_a(i) = I(i)$. By the definition of single-consuming, the integer x and the configuration $\texttt{Prune}(D, I, R) = D(i) \setminus xR_a(i)$ remain unique.

It follows that the recursive algorithm used in Theorem 32 solves reachability for a feed-forward, single-source, and single-consuming rule set by allowing void rules when pruning. ◀

C Appendix for Section 5

▶ **Theorem 35.** *Reachability for CRNs with only rules of size $(3, 0)$ is NP-complete.*

Proof. We reduce from the 3DM problem. Let $H = (X, Y, Z, T)$ be an input to the 3DM problem. From this, create an input to the reachability problem as follows. Let $\Lambda = \{S_v | v \in X \bigcup Y \bigcup Z\}$, and let $\Gamma = \{S_x + S_y + S_z \to \varnothing | (x, y, z) \in T\}$. Let configuration I be the configuration in which each species has count 1 and let D be the configuration in which each species has count 0. Then D is reachable from I under (Λ, Γ) if and only if H has a three-dimensional matching. ◀

▶ **Corollary 36.** *Reachability for CRNs with only rules of size $(0, 3)$ is NP-complete.*

Proof. We show this by reduction from the reachability problem with size $(3, 0)$ rules. Consider an instance of the reachability problem with an input of a CRN (Λ, Γ), an initial configuration I, and a destination configuration D in which rules in Γ are exclusively sized $(3, 0)$ void rules. Let $\Lambda' = \Lambda$, $\Gamma' = \overleftarrow{\Gamma}$, initial configuration $I' = D$, and target configuration $D' = I$. Observe that Γ' consists of rules only of size $(0, 3)$. By Lemma 53, D' is reachable from I' under ruleset Γ' if and only if D is reachable from I under ruleset Γ. ◀

▶ **Theorem 37.** *Reachability in CRNs is in P with rules of size $(2, 0)$ if configuration counts are encoded in unary.*

Proof. We show this by reducing reachability in this scenario to the two-dimensional matching problem, which has an established polynomial-time solution. We first consider the configuration $X = I - D$, creating a graph from this configuration. For each non-zero count species $X(i) > 0$, $X(i)$ vertices are added to the graph of type i. For each rule $i + j \to \emptyset$, we add edges to the graph connecting all vertices of type i to all vertices of type j. Then we have that X can reach the empty configuration if and only if the created graph has a perfect two-dimensional matching, and thus I reaches D if and only if such a matching exists. ◀

▶ **Definition 38** (Bipartite CRN). *A bipartite CRN (Λ, Γ) is one in which the species Λ can be partitioned into two disjoint sets Λ_1 and Λ_2 such that for each rule $R \in \Gamma$, there are at most 2 reactants of R and they do not occur within the same partition of Λ.*

▶ **Theorem 39.** *Reachability is polynomial-time solvable for bipartite CRNs with $(2, 0)$ rules.*

Proof. We show this by reducing reachability for a $(2, 0)$ rule bipartite CRN to the maximum network flow problem. Consider an input $(2, 0)$-rule bipartite CRN (Λ, Γ) with partitions Λ_1 and Λ_2, input configuration I, and output configuration D. From this, generate a max-flow instance as follows: for each $s \in \Lambda$, let the network contain a corresponding vertex v_s. For each rule, $a + b \to \emptyset \in \Gamma$, add an infinite capacity edge between vertices v_a and v_b. For each $x_i \in \Lambda_1$, add an edge from the source vertex to vertex v_{x_i} of capacity $I[i] - D[i]$, and for each $y_i \in \Lambda_2$, add an edge from v_{y_i} to the sink vertex of capacity $I[i] - D[i]$. The maximum-flow of this network is equal to the configuration volume $I - D$ if and only if D is reachable from I under Γ, and therefore reachability can be computed in polynomial time using the Edmonds-Karp maximum-flow algorithm. ◀

APPENDIX E

APPENDIX E

COMPUTING THRESHOLD CIRCUITS WITH VOID REACTIONS IN STEP CHEMICAL REACTION NETWORKS

Computing Threshold Circuits with Void Reactions in Step Chemical Reaction Networks

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

We introduce a new model of *step* Chemical Reaction Networks (step CRNs), motivated by the step-wise addition of materials in standard lab procedures. Step CRNs have ordered reactants that transform into products via reaction rules over a series of steps. We study an important subset of weak reaction rules, *void* rules, in which chemical species may only be deleted but never changed. We demonstrate the capabilities of these simple limited systems to simulate threshold circuits and compute functions using various configurations of rule sizes and step constructions, and prove that without steps, void rules are incapable of these computations, which further motivates the step model. Additionally, we prove the coNP-completeness of verifying if a given step CRN computes a function, holding even for $O(1)$ step systems.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases CRN, Chemical Reaction Network, Threshold Circuits, Void Reactions

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Category Student Paper. The circuit constructions given, and the presentations were all created and primarily written by five undergraduate students. Specifically, Sections 3 and 4. The images, examples, and tables were also created by them. The introduction and lower bounds were mainly done by graduate students (two MS and one PhD).

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:21

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

1 Introduction

Chemical Reaction Networks (CRNs) are one of the most established and longest studied models of self-assembly [5, 6]. CRNs originate in attempting to model chemical interactions as molecular species that react and create products from the reaction. This can be represented as an original number of each species and a set of replacement rules. The fundamental nature of the model is evident in the independent inception of equivalent models in multiple areas of research through other motivations [17], such as Vector Addition Systems (VASs) [26] and Petri-Nets [32]. Further, Population Protocols [2] are a restricted version where the number of input and output elements are each two.

Step CRNs. We propose and investigate an important but straightforward extension to the CRN model motivated by the desire to reflect standard laboratory and medical practices. The *Step* CRN models augments the CRN model with a sequence of discrete steps where an additional specified amount of chemical species is combined with the existing CRN after running the system to completion. Our goal is to explore the computational power of Step CRNs using highly restricted classes of CRN rules that would otherwise be computationally weak. In particular, we consider the problem of implementing the computationally universal class of *Threshold Circuits* using only *void* rules.

Void Rules. We study the computational power of Step CRNs under an extremely simple subset of CRN rules termed *void* rules [1]. General CRN rules are powerful since they allow the removal, addition, and replacement of species. Impressively, these rules have successful experimental implementations using DNA strand replacement mechanisms [37]. However, implementing this level of generality requires sophisticated, and large, DNA complexes that incur practical errors and constitute one of the primary hurdles limiting the scalable implementation of molecular computing schemes [13, 43]. In contrast, void rules only have the ability to delete species. Such a simple subset of reactions could plausibly permit drastically simpler and scalable molecular implementations based solely on the pair-wise bonding strength of single-stranded DNA. The only drawback is the inability of void rule systems to compute even simple functions. We show that void rules become computationally powerful in the step model with just tri-molecular or bi-molecular interactions. Specifically, we show how Threshold Circuits (TC), a powerful class of circuits with applications in deep learning, are simulated with void rules using a number of steps linear in the circuit's depth. Our utilization of steps under this void rule restriction is necessary, as we further show that even simple circuits require the use of steps when restricted to pure void rules.

1.1 Previous Work

Computation in Chemical Reaction Networks. Stochastic Chemical Reaction Networks are only Turing-complete with the possibility for error [36] while error-free stochastic Chemical Reaction Networks can compute precisely the set of semilinear functions [4, 15]. CRNs have also recently been shown to be experimentally viable through DNA Strand Displacement (DSD) systems [37], and several CRN to DSD compilers have been created [9, 27, 40, 46].

Boolean Circuits. Using molecules for information storage and Boolean logic is a deep field of study. Here, we show a few highlights, starting with one of the first discussions in 1988 [8] and an initial presentation of circuits with CRNs in 1991 [24]. Since then, the area has been extensively studied in CRNs and related models [7, 10, 12, 17, 20, 25, 28, 33, 34]. Numerous gates have been built experimentally and proposed theoretically such as the AND [12, 18, 29, 34, 39, 45], OR [12, 18, 34, 39], NOT [12], XOR [12, 45], NAND [12, 17, 20, 44], NOR [12], Parity [21, 22, 23], and Majority [3, 11, 30]. Symmetric boolean functions of

Function Computation					
Rules	**Species**	**Steps**	**Simulation**	**Family**	**Ref**
$(3,0)$	$O(\min(W^2, GF_{out}))$	$O(D \log F_{out})$	Strict	**TC** Circuits	Theorem 7
$(2,0)(2,1)$	$O(G)$	$O(D)$	Strict	**TC** Circuits	Theorem 8
$(2,1)$	$O(G)$	$O(D \log F_{maj})$	Strict	**TC** Circuits	Corollary 9
$(c,0)$	any	$\Omega(\log k)$	Strict	k-CNOT	Theorem 11

Strict Function Verification			
Rules	**Steps**	**Complexity**	**Ref**
$(3,0)$	$O(1)$	coNP-complete	Theorem 14

Table 1 Summary of n-bit circuit simulation results. D is the depth of the circuit, W is the width, G is the number of gates in a circuit or number of operators in a formula, F_{out} is the max fan-out, and F_{maj} is the max fan-in of majority gates. **TC** stands for Threshold Circuits. The k-CNOT is a k fan-in generalization of a Controlled NOT gate. Rule size $(c,0)$ means the row holds for all integer constants $c > 0$.

n variables such as Majority have been found to have a circuit depth of $O(\log n)$ when implemented by AND, OR and NOT gates [35].

Void Rules. The reachability problem, with systems of only void rules in proper CRNs, was studied in [1]. Previous studies had included void rules as a part of their systems but were never studied exclusively. The CRN++ programming language [42] allows these reactions to be programmed using the *sub* module. However, if any product(s) remain, they can differ from the reactants. They can also be considered a subcategory of the broader concept of the extinction of rules and species in a system as referred to in [44].

Mixing Systems. Another generalization of CRNs that is closely related to the step model is I/O CRNs [20], where additional inputs can be added at timed intervals. Still, those inputs are read-only in the system (used exclusively as catalysts). Step CRNs generalize I/O CRNs as the inputs are not read-only and are rate-independent, unlike I/O CRNs. Staged systems have been explored in many self-assembly models[14, 16, 19, 31].

1.2 Our Contributions

Table 1 has an overview of the main results of this paper beyond the introduction of the model and simulation definitions. The most important results being the ability to simulate the class of Threshold Circuits (TC) by simulating AND, OR, NOT, and MAJORITY gates through a restrictive definition of simulation while only using small void rules.

In Section 2, we define Step Chemical Reaction Networks and what it means to compute a function. Following, in Section 3, we show how to simulate the class TC of Threshold Circuits with void rules of size $(3,0)$ (rules where 3 species react to delete each other) using $O(D \log f)$ steps, where D is the depth of the circuit and f denotes the maximum fan-out of the circuit. In Section 4, we achieve the same result using both $(2,0)$ and $(2,1)$ rules and a slightly more efficient step complexity of $O(D)$. We then use exclusively $(2,1)$ rules to achieve this same result by adding a factor of $\log F_{maj}$ to the steps, where F_{maj} is the maximum fan-in of majority gates. In Section 5, we show there exist functions that require a logarithmic number of steps when restricted to constant reaction size, as well as the existence of $O(1)$-depth threshold circuits of fan-out f that require $\Omega(\log f)$ steps, which matches the $O(D \log f)$ upper bound for $(3,0)$ circuits. Finally, we show that it is coNP-complete to know whether a function can be strictly simulated by a step CRN system.

2 Preliminaries

2.1 Chemical Reaction Networks

Basics. Let $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{|\Lambda|}\}$ denote some ordered alphabet of *species*. A configuration over Λ is a length-$|\Lambda|$ vector of non-negative integers that denotes the number of copies of each present species. A *rule* or *reaction* is represented as an ordered pair of configuration vectors $R = (R_r, R_p)$. R_r contains the minimum counts of each *reactant* species necessary for reaction R to occur, where reactant species are either *consumed* by the rule in some count or leveraged as *catalysts* (not consumed); in some cases a combination of the two. The *product* vector R_p has the count of each species *produced* by the *application* of rule R, effectively replacing vector R_r. The species corresponding to the non-zero elements of R_r and R_p are termed *reactants* and *products* of R, respectively.

The *application* vector of R is $R_a = R_p - R_r$, which shows the net change in species counts after applying rule R once. For a configuration C and rule R, we say R is applicable to C if $C[i] \geq R_r[i]$ for all $1 \leq i \leq |\Lambda|$, and we define the *application* of R to C as the configuration $C' = C + R_a$. For a set of rules Γ, a configuration C, and rule $R \in \Gamma$ applicable to C that produces $C' = C + R_a$, we say $C \rightarrow^1_\Gamma C'$, a relation denoting that C can transition to C' by way of a single rule application from Γ. We further use the notation $C \rightarrow^*_\Gamma C'$ to signify the transitive closure of $\rightarrow^1_\Gamma$ and say C' is *reachable* from C under Γ, i.e., C' can be reached by applying a sequence of applicable rules from Γ to initial configuration C. Here, we use the following notation to depict a rule $R = (R_r, R_p)$: $\sum_{i=1}^{|\Lambda|} R_r[i]s_i \rightarrow \sum_{i=1}^{|\Lambda|} R_p[i]s_i$.

For example, a rule turning two copies of species H and one copy of species O into one copy of species W would be written as $2H + O \rightarrow W$.

▶ **Definition 1** (Discrete Chemical Reaction Network). *A discrete chemical reaction network (CRN) is an ordered pair* (Λ, Γ) *where* Λ *is an ordered alphabet of species, and* Γ *is a set of rules over* Λ.

An initial configuration and CRN (Λ, Γ) is said to be *bounded* if a terminal configuration is guaranteed to be reached within some finite number of rule applications starting from configuration I. We denote the set of reachable configurations of a CRN as $REACH_{I,\Lambda,\Gamma}$. A configuration is called *terminal* with respect to a CRN (Λ, Γ) if no rule R can be applied to it. We define the subset of reachable configurations that are terminal as $TERM_{I,\Lambda,\Gamma}$.

2.2 Void Rules

▶ **Definition 2** (Void and Autogenesis rules). *A rule* $R = (R_r, R_p)$ *is a* void *rule if* $R_a = R_p - R_r$ *has no positive entries and at least one negative entry. A rule is an* autogenesis *rule if* R_a *has no negative values and at least one positive value. If the reactants and products of a rule are each multisets, a void rule is a rule whose product multiset is a strict submultiset of the reactants, and an autogenesis rule one where the reactants are a strict submultiset of the products. There are two classes of void rules, catalytic and true void. In catalytic void rules, such as* $(2, 1)$ *rules, one or more reactants remain, and one or more is deleted after the rule is applied. In true void rules, such as* $(2, 0)$ *and* $(3, 0)$ *rules, there are no products remaining.*

▶ **Definition 3.** *The* size/volume *of a configuration vector* C *is* $\texttt{volume}(C) = \sum C[i]$.

▶ **Definition 4** (size-(i, j) rules). *A rule* $R = (R_r, R_p)$ *is said to be a size-*(i, j) *rule if* $(i, j) = (\texttt{volume}(R_r), \texttt{volume}(R_p))$. *A reaction is trimolecular if* $i = 3$, *bimolecular if* $i = 2$, *and unimolecular if* $i = 1$.

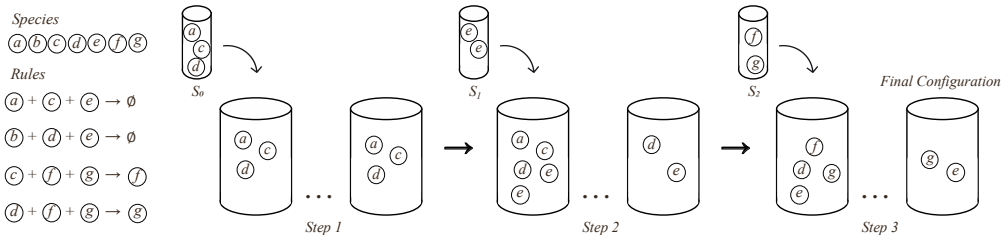


Figure 1 An example step CRN system. The test tubes show the species added at each step and the system with those elements added. The CRN species and void rule-set are shown on the left.

2.3 Step CRNs

A step CRN is an augmentation of a basic CRN in which a sequence of additional copies of some system species are added after a terminal configuration is reached. Formally, a step CRN of k steps is an ordered pair $((\Lambda, \Gamma), (s_0, s_1, s_2, \ldots, s_{k-1}))$, where the first element of the pair is a normal CRN (Λ, Γ), and the second is a sequence of length-$|\Lambda|$ vectors of non-negative integers denoting how many copies of each species type to add after each step. Figure 1 illustrates a simple step CRN system.

Given a step CRN, we define the set of reachable configurations after each sequential step. Let $REACH_1$ be the set of reachable configurations of (Λ, Γ) with initial configuration c_0 at step s_0, and let $TERM_1$ be the subset of reachable configurations that are terminal. Define $REACH_2$ to be the union of all reachable configurations from each possible starting configuration attained by adding s_1 to a configuration in $TERM_1$. Let $TERM_2$ be the subset of these reachable configurations that are terminal. Similarly, define $REACH_i$ to be the union of all reachable sets attained by using initial configuration c_{i-1} at step s_{i-1} plus any element of $TERM_{i-1}$, and let $TERM_i$ denote the subset of these configurations that are terminal.

The set of reachable configurations for a k-step CRN is the set $REACH_k$, and the set of terminal configurations is $TERM_k$. A classical CRN can be represented as a step CRN with $k = 1$ steps and an initial configuration of $I = s_0$.

Note that our definitions assume only the terminal configurations of a given step are passed on to seed the subsequent step. This makes sense if we assume we are dealing with *bounded* systems, as this represents simply waiting long enough for all configurations to reach a terminal state before proceeding to the next step. In this paper we only consider bounded void rule systems; we leave more general definitions to be discussed in future work.

2.4 Computing Functions in Step CRNs

Here, we define what it means for a step CRN to compute a function $f(x_1, \ldots_n) = (y_1, \ldots y_m)$ that maps n-bit strings to m-bit strings. For each input bit, we denote two separate species types, one representing bit 0, and the other bit 1. We add one copy for each bit to encode an input n-bit strig. Similarly, each output bit has two representatives(for 0 and 1), and we say the step CRN computes function f if for any given n-bit input $x_1, \ldots x_n$, the system results in a final configuration whose output species encode the string $f(x_1, \ldots x_n)$. For a fixed function f, the values denoted at each step s_i are fixed to disallow outside computation.

Input-Strict Step CRN Computing. Given a Boolean function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ that maps a string of n bits to a string of m bits, we define the computation of f with a step CRN. An input-strict step CRN computer is a tuple $C_s = (S, X, Y)$ where $S = ((\Lambda, \Gamma), (s_0, s_1, \ldots, s_{k-1}))$ is a step CRN, and $X = ((x_1^0, x_1^1), \ldots, (x_n^0, x_n^1))$ and $Y =$

$((y_1^0, y_1^1), \ldots, (y_m^0, y_m^1))$ are sequences of ordered-pairs with each $x_i^0, x_i^1, y_j^0, y_j^1 \in \Lambda$. Given an n-input bit string $b = b_1, \ldots, b_n$, configuration $X(b)$ is defined as the configuration over Λ obtained by including one copy of x_i^0 only if $b_i = 0$ and one copy of x_i^1 only if $b_i = 1$ for each bit b_i. We consider this representation of the input to be strict, as opposed to allowing multiple copies of each input bit species. The corresponding step CRN $(\Lambda, \Gamma, (s_0 + X(b), \ldots, s_{k-1}))$ is obtained by adding $X(b)$ to s_0 in the first step, which conceptually represents the system programmed with specific input b.

An input-strict step CRN computer *computes* function f if, for all n-bit strings b and for all terminal configurations of $(\Lambda, \Gamma, (s_0 + X(b), \ldots, s_{k-1}))$, the terminal configuration contains at least 1 copy of y_j^0 and 0 copies of y_j^1 if the j^{th} bit of $f(b)$ is 0, and at least 1 copy of y_j^1 and 0 copies of y_j^0 if the j^{th} bit of $f(b)$ is 1, for all j from 1 to m.

We use the term *strict* to denote requiring exactly one copy of each bit species and we leave it for future work to consider a more general form of input allowance or strict output. Here, we only consider input-strict computation, so we use input-strict and strict interchangeably.

Relation to CRN Computers. Previous models of CRN computers considered functions over large domains such as the positive integers. Due to the infinite domain, the input to such systems cannot be bounded. As such, the CRN computers shown in [15] define the input in terms of the volume of some input species. In these scenarios, CRN computers are limited to computing semi-linear functions. Here, we instead focus on computing n-bit functions, and instead encode the input per bit with potentially unique species. This is a model more similar to the PSPACE computer shown in [38].

2.5 Boolean and Threshold Circuits

A Boolean circuit on n variables $x_1, x_2, \ldots, x_n$ is a directed, acyclic multi-graph. The vertices of the graph are generally referred to as *gates*. The in-degree and out-degree of a gate are called the *fan-in* and *fan-out* of the gate respectively. The fan-in 0 gates (*source* gates) are labeled from $x_1, x_2, \ldots, x_n$, or labeled by constants starting at 0 or 1. Each non-source gate is labeled with a function name, such as AND, OR, or NOT. Given an assignment of boolean values to variables $x_1, x_2, \ldots, x_n$, each gate in the circuit can be assigned a value by first assigning all source vertices the value matching the labeled constant or labeled variable value and subsequently assigning each gate the value computed by its labeled function on the values of its children. Given a fixed ordering on the output gates, the sequence of bits assigned to the output gates denotes the value computed by the circuit on the given input.

The *depth* of a circuit is the longest path from a source vertex to an output vertex. Here, we focus on circuits that consist of AND, OR, NOT, and MAJORITY gates with arbitrary fan in. We refer to circuits that use these gates as *threshold circuits* (TC).

2.6 Notation

When discussing a boolean circuit, we use the following variables to denote the properties of the circuit: Let D denote the circuit's depth, G the number of gates in the circuit, W the circuit's width, F_{out} the maximum fan-out of any gate in the circuit, F_{in} the maximum fan-in, and F_{maj} the maximum fan-in of any majority gate within the circuit.

3 Computation of Threshold Circuits with (3, 0) Rules

Here, we introduce a step CRN system construction with only true void rules that can compute Threshold Circuits. In other words, given any TC and some truth assignment to

the input variables, we can construct a step CRN with only true void rules that computes the same output as the circuit.

This section specifically focuses on step CRNs consisting of $(3, 0)$ rules. Subsection 3.1 shows how the system can compute individual logic gates, and we show an example construction of a circuit in Subsection 3.2. We then present the general construction of computing TC circuits by two different methods, differing in the number of species needed based on the fan-out and width of the circuit. This results in Theorem 7, which states that TCs can be strictly computed, even with unbounded fan-out, with $O(\min(W^2, GF_{out}))$ species, $O(D \log F_{out})$ steps, and $O(W)$ volume.

3.1 Computing Logic Gates

Indexing. The number of steps to compute an individual depth level of a circuit varies between 2-8 steps depending on the gates and wiring of the specified circuit. To convert a circuit into a $(3, 0)$ step CRN system, we need to *index* the wires (input and output) at each level of the circuit in order to ensure the species is input to the correct gate. An example circuit with bit/wire indexing is shown in Figure 3c. At each level, we call the indices of the inputs of gates the *input indices*, and the indices of the output of each gate the *gate indices*. Note that the index numbers may need to change along the wire, or change due to fan-out/fan-in (see Figure 3c). This is accomplished by rules of the form $t_{j\to i}$ that map an input index of j to a gate index of i.

Bit Representation. The input bits of a binary gate are represented in a step CRN with $(3, 0)$ rules by the species x_n^b, where $n \in \mathbb{N}$ and $b \in \{T, F\}$. Here, n represents the bit's index (based on the ordering of all bits into the gates) and b represents its truth value. Let f_i^{in} be the set of all the indices of input bits fanning into a gate at index i (gate indices). Let f_i^{out} be the set of all indices of the output bits fanning out of a gate at index i.

The output bits of a gate are represented by the species $y_{n,g}^b$, where n is instead the output bit's index (input index of the next level) and g denotes the gate type $g \in \{B, A, O, N, M\}$ (BUFFER, AND, OR, NOT, and MAJORITY). The BUFFER (B) represents a signal wire that changes depth without passing through a gate. For example, the outputs of an AND gate, an OR gate, and a NOT gate at index n are represented by the species $y_{n,A}^b$, $y_{n,O}^b$, and $y_{n,N}^b$, respectively.

AND/OR Gate. The general process to compute an AND gate (an OR gate is similar) is given in Table 2. First, all input species are converted into a new species $a_{i,g}^b$ (step 1). The species retains truth value b as the original input, and includes the gate index i and type of the gate g. The species $b_{i,g}^b$ is then introduced (step 2), which computes the operation of gate g across all existing species. Any species that do not share the same truth value as the gate's intended output are deleted (step 3-4). The species remaining after the operation is then converted into the correct output species (step 5). The species u_i, v_i, w_i, and $t_{j\to i}$, where j is the input index and i is the gate index, are used to assist in removing excess species in certain steps.

AND Example. Consider an AND gate whose gate index is 1 with input bits 1 and 0 as shown in Figure 2. In this case, $|f_i^{in}| = 2$ and the initial configuration consists of the species x_1^T and x_2^F. Following Table 2, this gate can be computed in five steps.

1. Two $a_{1,A}^T$, two $a_{1,A}^F$, one $t_{1\to 1}$, and one $t_{2\to 1}$ species are added to the system. This converts the two input species of the gate into $a_{1,A}^T$ and $a_{1,A}^F$ (causes all species except $a_{1,A}^T$ and $a_{1,A}^F$ to be deleted).
2. One $b_{1,A}^T$, two $b_{1,A}^F$, and two u_1 species are added. All species except a single $b_{1,A}^F$ are

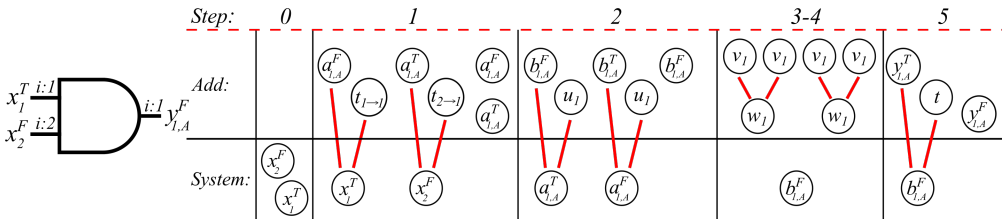


Figure 2 Example AND gate and steps to compute using $(3, 0)$ rules. Note the gate indexing of the wires ($i:1$ and $i:2$) and the input indexing for the next level ($i:1$ since there is only one gate). The process of computing the gate is shown on the right in steps. The new species added at each step are above and the remaining ones are in the system. The lines show the rules that would be executed during each step. To see the added species and rules in detail, see Table 2.

Steps			Relevant Rules	Description
1	*Add*	$\|f_i^{in}\| \cdot a_{i,g}^T$ $\|f_i^{in}\| \cdot a_{i,g}^F$ $\forall j \in f_i^{in} : t_{j\to i}$	$\forall j \in f_i^{in}$: $x_j^T + a_{i,g}^F + t_{j\to i} \to \emptyset$ $x_j^F + a_{i,g}^T + t_{j\to i} \to \emptyset$	$\forall j \in f_i^{in}$, convert x_j^b input species into $a_{i,g}^b$ species.
2	*Add*	$b_{i,g}^T$ $\|f_i^{in}\| \cdot b_{i,g}^F$ $\|f_i^{in}\| \cdot u_i$	$u_i + a_{i,g}^T + b_{i,g}^F \to \emptyset$ $u_i + a_{i,g}^F + b_{i,g}^T \to \emptyset$	Keep at least one of the correct output species and delete all incorrect species. This step computes the AND gate.*
3	*Add*	$2\|f_i^{in}\| \cdot v_i$	$u_i + v_i + v_i \to \emptyset$	Delete extra/unwanted species.
4	*Add*	$\|f_i^{in}\| \cdot w_i$	$w_i + v_i + v_i \to \emptyset$ $w_i + a_{i,g}^F + b_{i,g}^F \to \emptyset$	Delete extra/unwanted species.
5	*Add*	$y_{i,g}^T$, $y_{i,g}^F$, t	$b_{i,g}^T + y_{i,g}^F + t \to \emptyset$ $b_{i,g}^F + y_{i,g}^T + t \to \emptyset$	Convert $b_{i,g}^b$ into the proper output species $y_{i,g}^b$.

Table 2 *(3, 0) rules and steps for an AND gate. To compute an OR gate, add* $|f_i^{in}| \cdot b_{i,g}^T$ *and one* $b_{i,g}^F$ *in step 2 instead, and replace* $w_i + a_{i,g}^F + b_{i,g}^F \to \emptyset$ *with* $w_i + a_{i,g}^T + b_{i,g}^T \to \emptyset$ *in step 4.*

deleted by reactions.

3. Four v_1 species are added to remove excess species. There are none, so no reactions occur.
4. Two w_1 are added to delete excess species. Now, only a $b_{1,A}^F$ species remains.
5. One $y_{1,A}^T$, one $y_{1,A}^F$ and one t species are added. The $b_{1,A}^F$ species cause the $y_{1,A}^T$ and t species to be deleted. The $y_{1,A}^F$ species is the only species remaining, which represents the intended "false" output of the AND gate.

NOT Gate. Table 3 shows the general process to computing NOT gates. To compute a NOT gate, only the output species and species t are added in. In NOT gates specifically, the input species and the output species that share the same truth value b remove each other, leaving the complement of the input species as the remaining and correct output species.

Majority Gate. The majority gate outputs 1 if and only if more than half of its inputs are 1. Otherwise, it returns 0. The general step process is overviewed in Table 4 To compute a majority gate, all input species are converted into a new species $a_{i,M}^b$ (step 1). The species retains the same index i and truth value b as the original input. If the number of species fanning into the majority gate is even, an extra *false* input species is added in. The species $b_{i,M}^b$ is then introduced, which computes the majority operation across all existing species. Any species that represent the minority inputs are deleted (step 2). The species remaining after the operation are converted into the correct output (gate index) species (step 5). The species u_i, v_i, w_i, and $t_{j\to i}$, where j is the input index and i is the gate index, are used to assist in removing excess species in certain steps.

Steps			Relevant Rules	Description
1	*Add*	$y^T_{i,N}$ $y^F_{i,N}$ $t_{j\to i}$	$y^T_{i,N} + x^T_j + t_{j\to i} \to \emptyset$ $y^F_{i,N} + x^F_j + t_{j\to i} \to \emptyset$	The output species ($y^b_{i,N}$) that is the complement of the input species (x^b_j) will be the only species remaining.

Table 3 *(3, 0) rules and steps for a NOT gate.*

Steps			Relevant Rules	Description
1	*Add*	$\lvert f^{in}_i\rvert \cdot a^T_{i,M}$ $\lvert f^{in}_i\rvert \cdot a^F_{i,M}$ $\forall j \in f^{in}_i : t_{j\to i}$	$\forall j \in f^{in}_i :$ $x^T_j + a^F_{i,M} + t_{j\to i} \to \emptyset$ $x^F_j + a^T_{i,M} + t_{j\to i} \to \emptyset$	$\forall j \in f^{in}_i$, convert x^b_j input species into $a^b_{i,M}$ species.
2	*Add*	$\lfloor\lvert f^{in}_i\rvert/2\rfloor \cdot b^T_{i,M}$ $\lfloor\lvert f^{in}_i\rvert/2\rfloor \cdot b^F_{i,M}$ $(\lvert f^{in}_i\rvert - 1) \cdot u_i$	$u_i + a^T_{i,M} + b^F_{i,M} \to \emptyset$ $u_i + a^F_{i,M} + b^T_{i,M} \to \emptyset$	Adding $\lfloor\lvert f^{in}_i\rvert/2\rfloor$ amounts of $b^T_{i,M}$ and $b^F_{i,M}$ species will delete all of the minority species, leaving some amount of the majority species remaining.
3	*Add*	$2(\lvert f^{in}_i\rvert - 1) \cdot v_i$	$u_i + 2v_i \to \emptyset$	Delete extra/unwanted species.
4	*Add*	$(\lvert f^{in}_i\rvert - 1) \cdot w_i$	$w_i + 2v_i \to \emptyset$ $w_i + a^T_{i,M} + b^T_{i,M} \to \emptyset$ $w_i + a^F_{i,M} + b^F_{i,M} \to \emptyset$	Delete extra/unwanted species.
5	*Add*	$y^T_{i,M}$ $y^F_{i,M}$ t	$a^T_{i,M} + y^F_{i,M} + t \to \emptyset$ $a^F_{i,M} + y^T_{i,M} + t \to \emptyset$	Convert $a^b_{i,M}$ into the proper output species ($y^b_{i,M}$).

Table 4 *(3, 0) rules and steps for a majority gate.*

3.2 (3,0) Circuit Example

With the computation of individual gates demonstrated in our system, we now expand these features to computing entire circuits. We begin with a simple example (Figure 3c) to show the concepts before giving the general construction. The circuit has four inputs: x_1, x_2, x_3, and x_4. At the first depth layer, x_1 fans into a NOT gate and x_2 and x_3 are both fanned into an OR gate. At the next depth level, the output of the OR gate is fanned out twice. One of these outputs, along with the output of the NOT gate, is fanned into an AND gate, while the other and x_4 fans into another AND gate. At the last depth level, both AND gate outputs fan into an OR gate, which computes the final output of the circuit.

Table 5 shows how to compute the circuit in Figure 3c. The primary inputs of the circuit in Figure 3c are represented by the species in the initial configuration. Step 1 converts the primary inputs into input species. If there was any fan out of the primary inputs, it would be done in this step. Steps 2-6 compute the gates at the first depth level. Steps 7-8 compute the fan out between the first and second depth level. Step 9 converts the outputs of the gates at the first depth level into input species. Steps 10-14 use those inputs to compute the gates at the second depth level. Step 15 converts the outputs of these gates into inputs. Steps 16-20 compute the final gate. Step 21 converts the output of that gate into an input species that represents the solution to the circuit (x^F_1).

3.3 Computing Circuits

▶ **Lemma 5.** *Threshold circuits (TC) with a max fan-out of 2 can be strictly computed by a step CRN with only (3,0) rules, $O(W^2)$ species, $O(D)$ steps, and $O(W)$ volume.*

Proof. The initial configuration of the step CRN should consist of one $y^b_{n,B}$ species for each primary input with the appropriate indices and truth values. Section 3.1 explains how to compute TC gates. In order to run a circuit, we need to convert the outputs at an index i into the inputs for the next gate at index j. To simulate circuits with $O(W^2)$ species, we also need to be able to reuse these input, output, and helper species. This can be accomplished

Initial Configuration: $y^T_{1,B}$ $y^T_{2,B}$ $y^T_{3,B}$ $y^F_{4,B}$

Steps		Relevant Rules
1	x^T_1, x^T_2, x^T_3, x^T_4 $t_{1\to1}$, $t_{3\to3}$ x^F_1, x^F_2, x^F_3, x^F_4 $t_{2\to2}$, $t_{4\to4}$	$y^T_{1,B} + x^F_1 + t_{1\to1} \to \emptyset$ $y^T_{2,B} + x^F_2 + t_{2\to2} \to \emptyset$ $y^T_{3,B} + x^F_3 + t_{3\to3} \to \emptyset$ $y^F_{4,B} + x^T_4 + t_{4\to4} \to \emptyset$
2	$y^T_{1,N}$, $2a^T_{2,O}$, $y^T_{3,B}$ $t_{1\to1}$, $t_{3\to2}$ $y^F_{1,N}$, $2a^F_{2,O}$, $y^F_{3,B}$ $t_{2\to2}$, $t_{4\to3}$	$x^T_1 + y^T_{1,N} + t_{1\to1} \to \emptyset$ $x^T_2 + a^F_{2,O} + t_{2\to2} \to \emptyset$ $x^T_3 + a^F_{2,O} + t_{3\to2} \to \emptyset$ $x^T_4 + y^T_{3,B} + t_{4\to3} \to \emptyset$
3	$2b^T_{2,O}$, $2u_2$, $b^F_{2,O}$	$a^T_{2,O} + b^F_{2,O} + u_2 \to \emptyset$
4	$4v_2$	$u_2 + v_2 + v_2 \to \emptyset$
5	$2w_2$	$w_2 + v_2 + v_2 \to \emptyset$ $w_2 + a^T_{2,O} + b^T_{2,O} \to \emptyset$
6	$y^T_{2,O}$, t, $y^F_{2,O}$	$b^T_{2,O} + y^F_{2,O} + t \to \emptyset$
7	$y^T_{2,O}$, r, $y^F_{2,O}$	$y^T_{2,O} + y^T_{2,O} + r \to \emptyset$
8	$2y^T_{2,O}$, $2y^F_{2,O}$	$y^F_{2,O} + y^F_{2,O} + y^F_{2,O} \to \emptyset$
9	x^T_1, x^T_2, x^T_3, x^T_4 $t_{1\to1}$, $t_{2\to3}$ x^F_1, x^F_2, x^F_3, x^F_4 $t_{2\to2}$, $t_{3\to4}$	$y^F_{1,N} + x^T_1 + t_{1\to1} \to \emptyset$ $y^T_{2,O} + x^F_2 + t_{2\to2} \to \emptyset$ $y^T_{2,O} + x^F_3 + t_{2\to3} \to \emptyset$ $y^F_{3,B} + x^T_4 + t_{3\to4} \to \emptyset$
10	$2a^T_{1,A}$, $2a^T_{2,A}$ $2a^F_{1,A}$, $2a^F_{2,A}$ $t_{2\to1}$, $t_{4\to2}$ $t_{1\to1}$, $t_{3\to2}$	$x^F_1 + a^T_{1,A} + t_{1\to1} \to \emptyset$ $x^T_2 + a^F_{1,A} + t_{2\to1} \to \emptyset$ $x^T_3 + a^F_{2,A} + t_{3\to2} \to \emptyset$ $x^F_4 + a^T_{2,A} + t_{4\to2} \to \emptyset$
11	$b^T_{1,A}$, $b^T_{2,A}$ $2u_1$, $2b^F_{1,A}$ $2b^F_{2,A}$, $2u_2$	$a^T_{1,A} + b^F_{1,A} + u_1 \to \emptyset$ $a^F_{1,A} + b^T_{1,A} + u_1 \to \emptyset$ $a^T_{2,A} + b^F_{2,A} + u_2 \to \emptyset$ $a^F_{2,A} + b^T_{2,A} + u_2 \to \emptyset$
12	$4v_1$, $4v_2$	*No Rules Apply*
13	$2w_1$, $2w_2$	$w_1 + v_1 + v_1 \to \emptyset$ $w_2 + v_2 + v_2 \to \emptyset$
14	$y^T_{1,A}$, $y^T_{2,A}$, $2t$ $y^F_{1,A}$, $y^F_{2,A}$	$b^F_{1,A} + y^T_{1,A} + t \to \emptyset$ $b^F_{2,A} + y^T_{2,A} + t \to \emptyset$
15	x^T_1, x^T_2, $t_{1\to1}$ x^F_1, x^F_2, $t_{2\to2}$	$y^F_{1,A} + x^T_1 + t_{1\to1} \to \emptyset$ $y^F_{2,A} + x^T_2 + t_{2\to2} \to \emptyset$
16	$2a^T_{1,O}$, $t_{1\to1}$ $2a^F_{1,O}$, $t_{2\to1}$	$x^F_1 + a^T_{1,O} + t_{1\to1} \to \emptyset$ $x^F_2 + a^T_{1,O} + t_{2\to1} \to \emptyset$
17	$2b^T_{1,O}$, $2u_1$, $b^F_{1,O}$	$a^F_{1,O} + b^T_{1,O} + u_1 \to \emptyset$
18	$4v_1$	*No Rules Apply*
19	$2w_1$	$w_1 + v_1 + v_1 \to \emptyset$
20	$y^T_{1,O}$, t, $y^F_{1,O}$	$b^F_{1,O} + y^T_{1,O} + t \to \emptyset$
21	x^T_1, $t_{1\to1}$ x^F_1	$y^F_{1,O} + x^T_1 + t_{1\to1} \to \emptyset$

Table 5 $(3, 0)$ rules and steps to compute the circuit in Figure 3c based on the indexing shown in Figure 3a. Note that as in other tables, the 'Steps' column shows the number and types of species being added at the beginning of that step.

Steps		Relevant Rules	Description
1	*Add* $\forall j \in f_i^{out} : x^T_j, x^F_j, t_{i\to j}$	$\forall j \in f_i^{out}$: $y^T_{i,g} + x^F_j + t_{i\to j} \to \emptyset$ $y^F_{i,g} + x^T_j + t_{i\to j} \to \emptyset$	$\forall j \in f_i^{out}$, convert $y^b_{i,g}$ output species into x^b_j input species.

Table 6 *(3, 0) rules for converting outputs into inputs per circuit level.*

by having unique species for each gate at a given depth level. Figure 3a shows a pattern the gates can be indexed in.

When reusing species, we incorporate a unique $t_{i\to j}$ species (different from the $t_{j\to i}$ species used in computing gates) for each gate at index i that converts the output species into an input species with the index j. Converting outputs into inputs is done for all gates at the same depth level. Table 6 shows the steps and rules needed to complete this process.

Fan Out. In order to perform a 2-fan out, we create a second copy of the output species that is fanning out. Table 7 shows the steps and rules needed to perform this duplication. After duplicating the output, the simulation continues as usual. All outputs at the same depth level can be fanned out at the same time using these two steps.

Complexity. The $t_{i\to j}$ approach results in, at most, W^2 unique species since $1 \leq i, j \leq W$. All other types of species either have $O(1)$ or $O(W)$ unique species. Therefore, a simulation of a circuit with a max fan-out of 2 requires $O(W^2)$ species.

All gates at a given depth level are evaluated at the same time, so a simulation of a circuit with a max fan-out of 2 requires $O(D)$ steps. Additionally, circuits are evaluated one depth level at a time. Thus, at most, a max width amount of input, output, and helper species are added at the same time. All of the input, output, and helper species from previous depth

Steps			Relevant Rules	Description
1	*Add*	$y^T_{i,g}, y^F_{i,g}, r$	$y^T_{i,g} + y^T_{i,g} + r \to \emptyset$ $y^F_{i,g} + y^F_{i,g} + r \to \emptyset$	Flip output's bit (e.g. if species $y^T_{i,g}$ is present, then delete it and preserve $y^F_{i,g}$)
2	*Add*	$2y^T_{i,g}, 2y^F_{i,g}$	$y^T_{i,g} + y^T_{i,g} + y^T_{i,g} \to \emptyset$ $y^F_{i,g} + y^F_{i,g} + y^F_{i,g} \to \emptyset$	Delete all copies of the negation of the initial input, and preserve the two copies of the input that were just added.

Table 7 *(3,0) rules and steps for 2-fan out.*

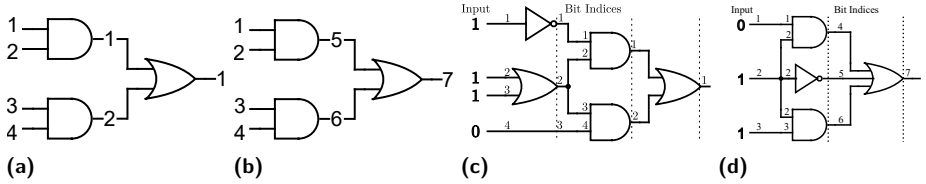


Figure 3 (a) Example indexing pattern of wires for the step CRN method using $O(W^2)$ species. (b) Example indexing pattern of wires for the step CRN method using $O(G)$ species. (c) Example circuit (with indexing) for Table 5. (d) Example circuit (with indexing) for Table 10.

levels get deleted when progressing to the next depth level, so the simulation requires $O(W)$ volume. A constant number of species, steps, and volume are needed to perform a 2-fan out, so a 2-fan out operation does not affect the complexity. ◀

▶ **Lemma 6.** *Threshold circuits (TC) with a max fan-out of 2 can be strictly computed by a step CRN with only* $(3, 0)$ *rules,* $O(G)$ *species,* $O(D)$ *steps, and* $O(W)$ *volume.*

Proof. Most of the rules, species, and steps used to compute a circuit with $O(W^2)$ species (Lemma 5) should also be used for this step CRN. The main difference is that there is an index for every gate in the circuit instead of limiting the indexes of these species by the max width. Figure 3b shows a pattern the gates can be indexed in. Also, we don't need the $t_{i\to j}$ species. This is because rules could overlap when reusing species, so the $t_{i\to j}$ species was used to make certain rules distinct and prevent the wrong reactions from occurring. However, each gate being represented by unique species eliminates the possibility of this error as every rule used to compute a gate will also be unique. So, in this step CRN, all instances where $t_{i\to j}$ species is used (including those in Section 3.1) are replaced by the generic t species.

Complexity. The $t_{i\to j}$ species that was the bottleneck for species in the step CRN discussed in Lemma 5 has been replaced by the t species. Therefore, the number of species is no longer upper bounded by $O(W^2)$. Instead, there are unique species for each gate, thus requiring $O(G)$ species. The differences discussed in this lemma do not affect the step or volume complexity determined in Lemma 5 ($O(D)$ and $O(W)$, respectively). ◀

▶ **Theorem 7.** *Threshold circuits (TC) can be strictly computed by a step CRN with only* $(3, 0)$ *rules,* $O(\min(W^2, G \cdot F_{out}))$ *species,* $O(D \log F_{out})$ *steps, and* $O(W)$ *volume.*

Proof. Since the methods used in Lemmas 5 and 6 can only achieve a max fan-out of 2, a circuit with a larger fan-out (F_{out}) must be turned into a circuit with a max fan-out of 2. The max width does not change in the transformation process, but the total number of gates does. The transformation process creates a binary tree-like structure within the circuit, where the gate that is fanning out can be likened to the root, and the gates that the output is being inputted into can be likened to the leaves. Therefore, because a binary tree can have, at most, $2\ell - 1$ vertices, and an arbitrary fan out already has the "root" gate and

"leaves" gates, the transformation process requires, at most, $\ell - 2$ more gates to construct the binary tree-like structure. Thus, the method requiring $O(G)$ species would require $O(GF_{out})$ species to simulate a circuit with arbitrary fan-out. The most efficient method can be used for a given circuit, resulting in $O(\min(W^2, GF_{out}))$ species required to simulate a circuit.

Due to the binary tree-like structure made for gates with large fan out, the transformation process would increase the size of the depth from D to $D \log F_{out}$. Since the steps are dependent on the size of the depth, a circuit simulation would require $O(D \log F_{out})$ steps.

The volume of a circuit simulation does not differ between the two methods ($O(W)$ volume) nor is it affected by the transformation process. ◀

4 Threshold Circuits with (2, 0) and (2, 1) Catalyst Rules

Having established computation results with step CRNs using only true void rules, we now examine step CRNs whose rule-sets contain catalytic rules. These rulesets can either consist of only $(2, 1)$ rules or both $(2, 1)$ and $(2, 0)$ rules. Subsection 4.1 shows how the computation of logic gates is possible in step CRNs with just $(2, 0)$ or $(2, 1)$ rules. We then demonstrate with Theorem 4.3 how the system can compute TCs with $O(G)$ species, $O(D)$ steps, and $O(W)$ volume. Subsection 4.4 then shows that TCs can also be calculated (with more steps) with only the $(2, 1)$ catalyst rules.

4.1 Computing Logic Gates

Bit Representation and Indexing. The inputs of a binary gate are constructed the same as in Section 3.1. However, with catalysts, we slightly modify our indexing scheme. When fanning out, we do not split the output of the gate into input species with different indices because the catalyst rules remove the need to differentiate the input species. Let f_i^{in} be the set of all the indices of the inputs fanning into a gate at index i. Let f_i^{out} be the set of all the indices of the inputs fanning out of a gate at index i.

The output of a gate is represented by the species y_i^b or $y_{j\to i}^b$, where j is the index of the input bit and i is the index of the gate.

AND/OR/NOT Gate. Table 8 shows the general process to computing AND, OR, and NOT gates. To compute an AND gate, we add a single copy of the species representing a true output (y_i^T) and a species representing a false output for each input ($\forall j \in f_i^{in} : y_{j\to i}^F$). To compute an OR gate instead, we add a species representing a true output ($y_{j\to i}^T$) for each input and a single y_i^F species. To compute NOT gates, we add one copy of each output species (y_i^b). For every input into an AND/OR/NOT gate, a corresponding rule should be created to remove the output species of the gate with the opposite truth value to the input. If the output species has a unique $j \to i$ index, then only the input with the corresponding i can delete that output species.

These gates can also be computed with (2,1) catalyst rules by making the x_j^b species a catalyst. For example, the (2,0) rule $x_j^T + y_i^T \to \emptyset$ would be replaced by the (2,1) rule $x_j^T + y_i^T \to x_j^T$.

OR Example. Consider OR gate whose gate index is 1 with input bits 0 and 1. Here, $|f_i^{in}| = 2$, and the initial configuration consists of the species x_1^F and a x_2^T.

This gate can be computed in one step, following Table 8, by adding one $y_{1\to1}^T$, one $y_{2\to1}^T$, and one y_1^F species to the system. The species x_2^T and y_1^F delete each other. x_1^F and $y_{1\to1}^T$ are also removed together. Only the species $y_{2\to1}^T$ remains, which represents the intended "true" output of the OR gate.

Majority Gate. The general process of computing a majority gate is shown at Table 9. To compute a majority gate, all input species are converted into a new species a_i^b (step 1).

Gate Type	Step	Relevant Rules	Description
AND	*Add* y_i^T $\forall j \in f_i^{in} : y_{j\to i}^F$	$x_j^T + y_{j\to i}^F \to \emptyset$ $x_j^F + y_i^T \to \emptyset$	An input species with a certain truth value deletes the complement output species.
OR	*Add* y_i^F $\forall j \in f_i^{in} : y_{j\to i}^T$	$x_j^T + y_i^F \to \emptyset$ $x_j^F + y_{j\to i}^T \to \emptyset$	An input species with a certain truth value deletes the complement output species.
NOT	*Add* y_i^T y_i^F	$x_j^T + y_i^T \to \emptyset$ $x_j^F + y_i^F \to \emptyset$	The input and output species that share the same truth value delete each other.

Table 8 *(2, 0) rules for AND, OR, and NOT gates.*

Steps			Relevant Rules	Description
1	*Add*	$\lvert f_i^{in}\rvert \cdot a_i^T$ $\lvert f_i^{in}\rvert \cdot a_i^F$	$\forall j \in f_i^{in}$: $x_j^T + a_i^F \to \emptyset$ $x_j^F + a_i^T \to \emptyset$	$\forall j \in f_i^{in}$, convert x_j^b input species into a_i^b species.
2	*Add*	$\lfloor \lvert f_i^{in}\rvert /2 \rfloor \cdot b_i^T$ $\lfloor \lvert f_i^{in}\rvert /2 \rfloor \cdot b_i^F$	$a_i^T + b_i^F \to \emptyset$ $a_i^F + b_i^T \to \emptyset$	Adding $\lfloor \lvert f_i^{in}\rvert /2 \rfloor$ amounts of b_i^T and b_i^F species will delete all of the minority species, leaving some amount of the majority species remaining.
3	*Add*	y_i^T y_i^F	$a_i^T + y_i^F \to \emptyset$ $a_i^F + y_i^T \to \emptyset$	Convert a_i^b into the proper output species (y_i^b).

Table 9 *(2, 0) rules for majority gates.*

The species retains the same truth value b as the original input and has the gate index i. If the number of species fanning into the majority gate is even, an extra *false* input species is added in. The species b_i^b is then introduced, which computes the majority operation across all existing species. Any species that represent the minority inputs are deleted (step 2). The species remaining afterwards are then converted into the correct output species (step 3).

4.2 Examples

With the computation of individual gates demonstrated in our system, we now expand these features to computing entire circuits. We begin with a simple example in Figure 3d to show the concepts before giving the general construction.

Our example circuit has three inputs: x_1, x_2, and x_3. In the first layer, x_2 is fanned out three times. One is fanned into an AND gate with x_1, another fanned into a NOT gate, and the other fanned into an AND gate with x_3. Finally, at the next depth level, the output of all three gates are fanned into an OR gate, whose output is the final circuit output.

Table 10 shows how to compute the circuit in Figure 3d. The primary inputs of the circuit in Figure 3d are represented by the species in the initial configuration. Steps 1-5 fan out the second primary input, convert the output species (y_n^b) into input species (x_n^b), and delete excess species. Step 6 computes the gates at the first depth level. Steps 7-11 convert the output species into input species and deletes excess species. Step 12 computes the final gate. Steps 13-17 delete excess species and converts the output of the final gate into an input species that represents the solution to the circuit (x_7^T).

4.3 Computing Circuits with (2,0) Void and (2,1) Catalyst Rules

▶ **Theorem 8.** *Threshold circuits (TC) can be strictly computed with* $(2, 0)$ *void rules and* $(2, 1)$ *catalyst rules,* $O(G)$ *species,* $O(D)$ *steps, and* $O(W)$ *volume.*

Proof. With only $(2, 0)$ rules, there is no known way to perform fan-outs without introducing, at most, an exponential count of species at certain steps. This makes it impossible to strictly compute circuits with only $(2, 0)$ rules and results in a large volume. The use of $(2, 1)$

Initial Configuration:							
	y_1^F	y_2^T	y_3^T				
Steps			**Relevant Rules**		**Steps**		**Relevant Rules**
1	*Add*	d_x	*No Rules Apply*	**8**	*Add*	d_x	$d_x + d_x \to \emptyset$
2	*Add*	d_x	$d_x + d_x \to \emptyset$	**9**	*Add*	x_4^T, x_4^F x_5^T, x_5^F x_6^T, x_6^F	$y_{1\to4}^F + x_4^T \to y_{1\to4}^F$ $y_5^F + x_5^T \to y_5^F$ $y_6^T + x_6^F \to y_6^T$
3	*Add*	x_1^T, x_1^F $3x_2^T$, $3x_2^F$ x_3^T, x_3^F	$y_1^F + x_1^T \to y_1^F$ $y_2^T + x_2^F \to y_2^T$ $y_3^T + x_3^F \to y_3^T$	**10**	*Add*	d_y	$y_{1\to4}^F + d_y \to d_y$ $y_5^F + d_y \to d_y$ $y_6^T + d_y \to d_y$
4	*Add*	d_y	$y_1^F + d_y \to d_y$ $y_2^T + d_y \to d_y$ $y_3^T + d_y \to d_y$	**11**	*Add*	d_y	$d_y + d_y \to \emptyset$
5	*Add*	d_y	$d_y + d_y \to \emptyset$	**12**	*Add*	$y_{4\to7}^T$, $y_{5\to7}^T$ $y_{6\to7}^T$, y_7^F	$x_4^F + y_{4\to7}^T \to \emptyset$ $x_5^F + y_{5\to7}^T \to \emptyset$ $x_6^T + y_7^F \to \emptyset$
6	*Add*	y_4^T, $y_{1\to4}^F$ y_5^T, $y_{2\to4}^F$ y_5^F, $y_{2\to6}^F$ y_6^T, $y_{3\to6}^F$	$x_1^F + y_4^T \to \emptyset$ $x_2^T + y_{2\to4}^F \to \emptyset$ $x_2^T + y_5^T \to \emptyset$ $x_2^T + y_{2\to6}^F \to \emptyset$ $x_3^T + y_{3\to6}^F \to \emptyset$	**13**	*Add*	d_x	*No Rules Apply*
				14	*Add*	d_x	$d_x + d_x \to \emptyset$
				15	*Add*	x_7^T, x_7^F	$y_{6\to7}^T + x_7^F \to y_{6\to7}^T$
				16	*Add*	d_y	$y_{6\to7}^T + d_y \to d_y$
7	*Add*	d_x	*No Rules Apply*	**17**	*Add*	d_y	$d_y + d_y \to \emptyset$

Table 10 *(2, 0) and (2, 1) rules and steps to compute the circuit in Figure 3d with Figure 3b's indexing.*

catalyst rules enables the step CRN to compute with arbitrary fan-out without an increase in species count, as well as deleting all species that are no longer needed. This allows for strict computation and decreases the volume of the step CRN to a polynomial size.

The initial configuration of this step CRN should consist of a y_n^b species with the appropriate indexing and truth values for each primary input. Section 4.1 explains how to compute TC gates. To run the circuit, we must convert the output species into input species. In addition, this step CRN uses d_x and d_y as *deleting* species. Table 11 shows the steps and rules needed to perform arbitrary fan-out for a gate. All outputs at the same depth level can be fanned out at the same time.

Complexity: Having a constant amount of unique species represent each gate in a circuit and a constant number of helper species results in $O(G)$ species. All gates at a given depth level are computed at the same time in a constant number of steps. Thus, the circuit simulation requires $O(D)$ steps. This step CRN only needs to introduce a constant number of species to compute each gate, and it deletes all species no longer needed after computing all gates at a given depth level. Thus, the step CRN requires $O(W)$ volume. ◀

4.4 Computing Circuits with (2,1) Catalyst Rules

It's worth noting that (2,1) catalyst rules are able to compute TCs on their own. The main drawback is that there is no known way to directly compute majority gates without $(k \geq 2, 0)$ void rules. Thus, any majority gate must be computed using AND, OR, and NOT gates when using only catalyst rules. Furthermore, deleting species that are no longer needed is slightly more convoluted with (2,1) rules compared to pure void rules.

▶ **Corollary 9.** *Threshold circuits (TC) can be strictly computed with only* $(2, 1)$ *catalyst rules,* $O(G)$ *species,* $O(D \log F_{maj})$ *steps, and* $O(W)$ *volume.*

Proof. Section 4.1 explains how to compute AND/OR/NOT gates using (2,0) rules, and they can easily be changed to (2,1) rules by making the x_j^b species a catalyst. The method used to perform arbitrary fan out in Theorem 8 can be slightly modified to function with only (2,1) rules. Table 12 demonstrates how this can be done. A special property of using

Steps			Relevant Rules	Description
1	*Add*	d_x	$\forall n \in \{1, \cdots, G\}$: $\forall b \in \{T, F\}$ $d_x + x_n^b \to d_x$ $d_x + a_n^b \to d_x$ $d_x + b_n^b \to d_x$	Delete all input species (x_n^b) and helper species that are no longer needed.
2	*Add*	d_x	$d_x + d_x \to \emptyset$	Remove deleting species d_x.
3	*Add*	$\lvert f_i^{out}\rvert \cdot x_i^T$ $\lvert f_i^{out}\rvert \cdot x_i^F$	$y_i^T + x_i^F \to y_i^T$ $y_i^F + x_i^T \to y_i^F$ $\forall j \in f_i^{in}$: $y_{j\to i}^T + x_i^F \to y_{j\to i}^T$ $y_{j\to i}^F + x_i^T \to y_{j\to i}^F$	Add species representing true and false inputs and delete the species that are the complement of the output. A single output species can assign the truth value for as many input species as needed.
4	*Add*	d_y	$\forall n \in \{1, \cdots, G\}$: $d_y + y_n^T \to d_y$ $d_y + y_n^F \to d_y$ $\forall j \in f_i^{in}$: $d_y + y_{j\to i}^T \to d_y$ $d_y + y_{j\to i}^F \to d_y$	Delete all output species (y_n^b) that no longer needed.
5	*Add*	d_y	$d_y + d_y \to \emptyset$	Remove deleting species d_y.

Table 11 *(2, 0) and (2, 1) rules and steps for a gate with arbitrary fan out.*

(2,1) rules to compute gates is that the counts of the species being added are flexible. This is not the case when gates are computed with pure void rules, as it is necessary for the counts of certain species to be precise. For example, while Step 3 in Table 11 and Step 5 in Table 12 are functionally equivalent steps, they have different computing requirements. When computing with (2,0) rules, we need exactly $|f_i^{out}|$ amount of x_i^b species by the end of that step. On the other hand, when computing with (2,1) rules, we only need one copy of x_i^b, and even if multiple copies of that species were added, it would not have a significant impact on the computation of the circuit.

Complexity: The techniques used to compute circuits are functionally equivalent to the ones used in Theorem 8, so the upper bound of species and volume remain the same, that is, $O(G)$ and $O(W)$, respectively.

Since majority gates must be computed using AND and OR gates when using only (2,1) rules, the depth of the circuit must increase. The conversion of a majority gate to AND/OR/NOT gates can be achieved with $O(n)$ gates and $O(\log n)$ depth where n is the number of input bits of the majority gate [35] (any symmetric boolean function has a circuit of depth $O(\log n)$ and size $O(n)$ for n bits). Thus, the maximum number of steps needed to compute a circuit would be $O(D \log F_{maj})$, where F_{maj} is the maximum fan-in of any majority gate in the circuit. ◀

5 Lower Bounds and Hardness

In this section, we prove negative results for computing with step CRNs. First, we show there exists a family of functions that require a logarithmic number of steps to compute. Then, we show hardness of verifying whether a step CRN properly computes a given function.

5.1 Step Lower Bound for Controlled NOT

CNOT. The Controlled NOT gate is a 2-bit input and 2-bit output gate taking inputs X and Y, and outputting X and $X \oplus Y$. In other words, the gate flips Y if X is true.

k-CNOT. We generalize this to a Controlled k-NOT gate. This is a $(k+1)$-bit gate with

Steps			Relevant Rules	Description
1	*Add*	d'_x	$d'_x + d''_x \to d'_x$	Deleting species d''_x makes it possible for species d_x to exist in the next step without complications.
2	*Add*	d_x	$d_x + d'_x \to d_x$ $\forall n \in \{1, \cdots, G\}$: $\forall b \in \{T, F\}$ $d_x + x_n^b \to d_x$ $d_x + a_n^b \to d_x$ $d_x + b_n^b \to d_x$	Deleting species d'_x makes it possible for species d''_x to exist in the next step without complications. Delete all input species (x_n^b) and helper species that are no longer needed.
3	*Add*	d''_x	$d_x + d''_x \to d''_x$	Removes deleting species d_x.
4	*Add*	d'_y	$d'_y + d''_y \to d'_y$	Deleting species d''_y makes it possible for species d_y to exist in the next step without complications.
5	*Add*	x_i^T x_i^F	$y_i^T + x_i^F \to y_i^T$ $y_i^F + x_i^T \to y_i^F$ $\forall j \in f_i^{in}$: $y_{j\to i}^T + x_i^F \to y_{j\to i}^T$ $y_{j\to i}^F + x_i^T \to y_{j\to i}^F$	Add species representing true and false inputs and delete the species that are the complement of the output. A single output species can assign the truth value for as many input species as needed.
6	*Add*	d_y	$d_y + d'_y \to d_y$ $\forall n \in \{1, \cdots, G\}$: $d_y + y_n^T \to d_y$ $d_y + y_n^F \to d_y$ $\forall j \in f_i^{in}$: $d_y + y_{j\to i}^T \to d_y$ $d_y + y_{j\to i}^F \to d_y$	Deleting species d'_y makes it possible for species d''_y to exist in the next step without complications. Delete all output species (y_n^b) that are no longer needed.
7	*Add*	d''_y	$d_y + d''_y \to d''_y$	Remove deleting species d_y.

Table 12 *(2, 1) rules and steps for a gate with arbitrary fan out.*

inputs $X, Y_1, \ldots, Y_k$. The Y bits all flip if X is true. We choose this function since it has the property that changing 1 bit of the input changes a large number of output bits.

Configuration Distance. Recall configurations are defined as vectors. For two configurations c_0, c_1, we say the distance between them is $||c_0 - c_1||_1$, i.e., the sum of the absolute value of each entry in $c_0 - c_1$.

▶ **Lemma 10.** *Let r be a positive integer parameter. For all step CRNs Γ with void rules of size $(r_1, 0)$ with $r_1 \leq r$ and pairs of initial configurations c_T and c_F with distance 2 and equal volume, for any configuration c_{Ts} terminal in the step s from c_T, there exists a configuration c_{Fs} terminal in step s from c_F such that the distance between c_{Ts} and c_{Fs} is $O(r^s)$.*

Proof. Let T be the species that is in configuration c_T and not in c_F. Similarly, define the species F to be the species in configuration c_F and not in c_T. Consider the reaction sequence R_T starting from c_T and ending in c_{T1}. All but one reaction in R_T can be applied to c_F. Consider the reaction sequence R_F that differs from R_T by two reactions. The first is the reaction in R_T that consumes T and $r - 1$ other species, and the second is a reaction that consumes F and $r - 1$ other species. Applying R_F results in a configuration c_{F1} that differs from c_{T1} by at most $2r$ species.

Now, assume there are two initial configurations in step s with distance $2r^{s-1}$ away from each other. In the base case, each different species between the configurations can be used in

at most one rule, thus the species can only propagate $r-1$ additional changes in the next terminal configuration. This results in a difference of $2rr^{s-1} = 2r^s$ in the s stage. ◀

The configuration distance between two output configurations is related to the Hamming distance of the output strings they represent. Lemma 10 can be used to get a get a logarithmic lower bound for the number of steps required when we fix our rule size to be a constant.

▶ **Theorem 11.** *For all constants r, any CRN that strictly computes a k-CNOT gate with rules of size $(r_1, 0)$ satisfying $r_1 \leq r$ requires $\Omega(\log k)$ steps.*

Proof. Flipping only the X bit of the the input changes all $k+1$ output bits. It follows that in order to compute a k-CNOT, we must have at least k distance between the two final configurations even when starting from configurations with distance 1. We can also assume these have the same volume since by changing a bit value we are only changing which species we add. With Lemma 10, we get the following inequality and must compute s.

$$k \leq 2r^s \quad \rightarrow \quad \log k \leq 2s \log r \quad \rightarrow \quad \frac{\log k}{2 \log r} \leq s$$

Since r is a constant we get an asymptotic bound of $s = \Omega(\log k)$. ◀

We also note the k-CNOT can be computed by k XOR gates in parallel. This implies this lower bound does not hold with catalytic reactions either as Theorem 8 shows this can be computed in $O(1)$ steps or without the input-strict requirement. This is because increasing the fan-out of the X bit does not incur a cost in the number of steps in both of these generalizations. Plugging this XOR circuit into Theorem 7 gives a bound of $\Theta(\log k)$ steps showing the construction is optimal for some circuits.

5.2 Function Verification Hardness

We have established that void step CRNs can simulate Boolean circuits. We now discuss the complexity of the computational problem of determining if a given (void) step CRN does compute a given function. Specifically, we consider the following decision problem:

(Strict Function Verification): Given a step CRN $C_S = (S, X, Y)$ and a Boolean function $f(\cdot)$[1] where $f(x_1, \cdots, x_n) = y_1 : \{0,1\}^n \rightarrow \{0,1\}$, decide if C_S computes Boolean function $f(\cdot)$. In particular, let $f_0(x_1, \cdots, x_n) = false$, which is false for all inputs.

Theorem 12 shows that strict function verification in a step CRN system with void rules is coNP-hard, and coNP membership for this problem is shown in Theorem 13.

▶ **Theorem 12.** *It is coNP-hard to determine if a given a $\mathcal{O}(1)$-step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules computes the boolean function $f_0(x_1, \cdots, x_n)$.*

Proof. The decision problem 3SAT is a classical NP-complete problem. Its complementary problem, which is coNP-complete, is to decide if the conjunction of several clauses with three literals cannot be satisfied. We also reduce from the special case of 3SAT where each variable appears at most 4 times, shown to be NP-complete in [41]. Let $F(x_1, \cdots, x_n) = C_1 \wedge C_2 \wedge \cdots C_m$ be an instance for a 3SAT problem, where each C_i is a clause that has at most three literals, for example, $C_1 = (x_1 \vee \overline{x_2} \vee x_3)$. The function $F(.)$ can be computed by a boolean circuit of constant depth. Checking if $F(x_1, \cdots, x_n) = f_0(x_1, \cdots, x_n)$ for all $(x_1, \cdots, x_n)$ is the complementary problem for 3SAT.

[1] We assume that $f(\cdot)$ is given in the form of a circuit c_f. We leave as future work the complexity of other representations such as a truth table.

It follows from Theorem 7, which shows $F(.)$ can be simulated by a step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules. The number of steps of the CRN is $\mathcal{O}(D \log F_out)$. The depth of the circuit is constant since all clauses can be computed in parallel and the gates handle arbitary fan-in. The fanout is also constant because each variable only appears 4 times. ◀

▶ **Theorem 13.** *Determining if a given a s-step CRN $C_S = (S, X, Y)$ with $(r, 0)$ rules computes the boolean function $f_0(x_1, \cdots, x_n)$ is in coNP.*

Proof. This problem can be solved by a polynomial time non-deterministic algorithm which does the following. Pick a string b of length n and compute $f_0(b) = Y$. Then convert b to the initial configuration $X(b)$. Guess a sequence of s terminal configurations $c_1, \ldots, c_s$ where c_i is a terminal configuration in the ith step. To verify this, call the NP algorithm for reachability in Volume Decreasing CRNs from [1] to verify each configuration is reachable in the correct step. If the final configuration c_s does not represent Y then reject. ◀

▶ **Theorem 14.** *It is coNP-Complete to determine if a given a $O(1)$-step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules computes the boolean function $f_0(x_1, \cdots, x_n)$.*

Proof. Follows from Theorems 12 and 13. ◀

6 Conclusions and Future Work

We have proposed the step CRN model, a natural augmentation to the CRN model, and shown that void rule CRNs, a simple but computationally weak class of CRNs, become capable of efficiently simulating Threshold Circuits under this extension. We have shown this holds even when limited to $(3, 0)$ reactions, and further shown that bi-molecular reactions are equally powerful if permitted access to catalytic reactions. We also show that the step augmentation is fundamentally needed: without access to a super-constant number of steps, such computation is impossible. Finally, we utilize our positive results to show coNP-completeness for the problem of deciding if a given step CRN computes a given function.

The step CRN model presented in this work, along with our results, naturally lead to a number of additional promising research directions. A small sample of these are:

- Lower Bounds and Catalysts: We conjecture $(3, 0)$ void rules are the smallest size rules for strictly simulating TC without the use of a catalyst. Further, we show that more than a constant number of steps are required for circuit computation for non-catalytic void rules, but is it true with a catalyst?
- Robustness: While void rules potentially offer a simpler path to experimental feasibility and scalability, some of our techniques require precise counts of species to be added at different steps of the computational process. Such precision is a hurdle to experimental implementation. Thus, it is interesting to consider to what extent these results can be made robust to approximate counts (or have a lower-bound).
- Reachability: The *reachability* problem of determining if a given configuration is reachable from an initial configuration is well-studied in CRNs and other computational models. We showed that steps allow for greater computational power with void rules. How does the addition of steps affect the reachability problem, and specifically, what is the complexity when relating void rules, catalysts, rule size, and number of steps?
- General Staged CRNs: We explored a simple scheme for including separate stages into the CRN model by simply adding new species at each step. A more general modelling could include multiple separate bins that may be mixed or split together over a sequence of stages. Formalizing such a model and exploring the added power of this generalization is an interesting direction for future work.

References

1. Robert M. Alaniz, Bin Fu, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Reachability in restricted chemical reaction networks, 2022. arXiv:2211.12603.
2. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distribed Computing*, 18(4):235–253, mar 2006. doi:10.1007/s00446-005-0138-3.
3. Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21:87–102, 2008.
4. Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 2007.
5. Rutherford Aris. Prolegomena to the rational analysis of systems of chemical reactions. *Archive for Rational Mechanics and Analysis*, 19(2):81–99, jan 1965. doi:10.1007/BF00282276.
6. Rutherford Aris. Prolegomena to the rational analysis of systems of chemical reactions ii. some addenda. *Archive for Rational Mechanics and Analysis*, 27(5):356–364, jan 1968. doi:10.1007/BF00251438.
7. Adam Arkin and John Ross. Computational functions in biochemical reaction networks. *Biophysical journal*, 67(2):560–578, 1994.
8. Ari. Aviram. Molecules for memory, logic, and amplification. *Journal of the American Chemical Society*, 110(17):5687–5692, Aug 1988. doi:10.1021/ja00225a017.
9. 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 *International conference on DNA-based computers*, pages 232–248. Springer, 2017.
10. Z Beiki, Z Zare Dorabi, and Ali Jahanian. Real parallel and constant delay logic circuit design methodology based on the dna model-of-computation. *Microprocessors and Microsystems*, 61:217–226, 2018.
11. Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific reports*, 2(1):656, 2012.
12. Luca Cardelli, Marta Kwiatkowska, and Max Whitby. Chemical reaction network designs for asynchronous logic circuits. *Natural computing*, 17:109–130, 2018.
13. Luca Cardelli, Mirco Tribastone, and Max Tschaikowski. From electric circuits to chemical networks. *Natural Computing*, 19:237–248, 2020.
14. Cameron Chalk, Eric Martinez, Robert Schweller, Luis Vega, Andrew Winslow, and Tim Wylie. Optimal staged self-assembly of general shapes. *Algorithmica*, 80:1383–1409, 2018.
15. Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13(4):517–534, 2014.
16. Sonya C Cirlos, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Simulation of multiple stages in single bin active tile self-assembly. In *International Conference on Unconventional Computation and Natural Computation*, pages 155–170. Springer, 2023.
17. Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. *Programmability of Chemical Reaction Networks*, pages 543–584. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-540-88869-7_27.
18. Neil Dalchau, Harish Chandran, Nikhil Gopalkrishnan, Andrew Phillips, and John Reif. Probabilistic analysis of localized dna hybridization circuits. *ACS synthetic biology*, 4(8):898–913, 2015.
19. Erik D Demaine, Sarah Eisenstat, Mashhood Ishaque, and Andrew Winslow. One-dimensional staged self-assembly. *Natural Computing*, 12(2):247–258, 2013.
20. Samuel J Ellis, Titus H Klinge, and James I Lathrop. Robust chemical circuits. *Biosystems*, 186:103983, 2019.

21 Abeer Eshra and Ayman El-Sayed. An odd parity checker prototype using dnazyme finite state machine. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):316–324, 2013.

22 Daoqing Fan, Yongchao Fan, Erkang Wang, and Shaojun Dong. A simple, label-free, electrochemical dna parity generator/checker for error detection during data transmission based on "aptamer-nanoclaw"-modulated protein steric hindrance. *Chemical Science*, 9(34):6981–6987, 2018. doi:10.1039/C8SC02482K.

23 Daoqing Fan, Jun Wang, Jiawen Han, Erkang Wang, and Shaojun Dong. Engineering dna logic systems with non-canonical dna-nanostructures: Basic principles, recent developments and bio-applications. *Science China Chemistry*, 65(2):284–297, 2022.

24 Allen Hjelmfelt, Edward D Weinberger, and John Ross. Chemical implementation of neural networks and turing machines. *Proceedings of the National Academy of Sciences*, 88(24):10983–10987, 1991.

25 Hua Jiang, Marc D Riedel, and Keshab K Parhi. Digital logic with molecular reactions. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 721–727. IEEE, 2013.

26 Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi:https://doi.org/10.1016/S0022-0000(69)80011-5.

27 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.

28 Yu-Chou Lin and Jie-Hong R Jiang. Mining biochemical circuits from enzyme databases via boolean reasoning. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

29 David C Magri. A fluorescent and logic gate driven by electrons and protons. *New Journal of Chemistry*, 33(3):457–461, 2009.

30 Shay Mailloux, Nataliia Guz, Andrey Zakharchenko, Sergiy Minko, and Evgeny Katz. Majority and minority gates realized in enzyme-biocatalyzed systems integrated with logic networks and interfaced with bioelectronic systems. *The Journal of Physical Chemistry B*, 118(24):6775–6784, Jun 2014. doi:10.1021/jp504057u.

31 Dandan Mo and Darko Stefanovic. Iterative self-assembly with dynamic strength transformation and temperature control. In *DNA Computing and Molecular Programming: 19th International Conference, DNA 19, Tempe, AZ, USA, September 22-27, 2013. Proceedings 19*, pages 147–159. Springer, 2013.

32 Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, 1962.

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

34 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, Feb 2011. doi:10.1098/rsif.2010.0729.

35 Igor Sergeevich Sergeev. Upper bounds for the formula size of symmetric boolean functions. *Russian Mathematics*, 58:30–42, 2014.

36 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7(4):615–633, 2008.

37 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.

38 Chris Thachuk and Anne Condon. Space and energy efficient computation with dna strand displacement systems. In *International Workshop on DNA-Based Computers*, 2012.

39 Chris Thachuk, Erik Winfree, and David Soloveichik. Leakless dna strand displacement systems. In *DNA Computing and Molecular Programming: 21st International Conference,*

DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings 21, 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(1):1–12, 2017.

41 Craig A Tovey. A simplified np-complete satisfiability problem. *Discrete applied mathematics*, 8(1):85–89, 1984.

42 Marko Vasić, David Soloveichik, and Sarfraz Khurshid. Crn++: Molecular programming language. *Natural Computing*, 19:391–407, 2020.

43 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.

44 Erik Winfree. Chemical reaction networks and stochastic local search. In *DNA Computing and Molecular Programming: 25th International Conference, DNA 25, Seattle, WA, USA, August 5–9, 2019, Proceedings 25*, pages 1–20. Springer, 2019.

45 Wei Xiao, Xinjian Zhang, Zheng Zhang, Congzhou Chen, and Xiaolong Shi. Molecular full adder based on dna strand displacement. *IEEE Access*, 8:189796–189801, 2020. doi:10.1109/ACCESS.2020.3031221.

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

APPENDIX F

APPENDIX F

COMPLEXITY OF RECONFIGURATION IN SURFACE CHEMICAL REACTION NETWORKS

Complexity of Reconfiguration in Surface Chemical Reaction Networks

Robert M. Alaniz
University of Texas Rio Grande Valley, USA

Josh Brunner
Massachusetts Institute of Technology, USA

Michael Coulombe
Massachusetts Institute of Technology, USA

Erik D. Demaine
Massachusetts Institute of Technology, USA

Jenny Diomidova
Massachusetts Institute of Technology, USA

Timothy Gomez
Massachusetts Institute of Technology, USA

Elise Grizzell
University of Texas Rio Grande Valley, USA

Ryan Knobel
University of Texas Rio Grande Valley, USA

Jayson Lynch
Massachusetts Institute of Technology, USA

Andrew Rodriguez
University of Texas Rio Grande Valley, USA

Robert Schweller
University of Texas Rio Grande Valley, USA

Tim Wylie
University of Texas Rio Grande Valley, USA

Abstract

We analyze the computational complexity of basic reconfiguration problems for the recently introduced surface Chemical Reaction Networks (sCRNs), where ordered pairs of adjacent species nondeterministically transform into a different ordered pair of species according to a predefined set of allowed transition rules (chemical reactions). In particular, two questions that are fundamental to the simulation of sCRNs are whether a given configuration of molecules can ever transform into another given configuration, and whether a given cell can ever contain a given species, given a set of transition rules. We show that these problems can be solved in polynomial time, are NP-complete, or are PSPACE-complete in a variety of different settings, including when adjacent species just swap instead of arbitrary transformation (swap sCRNs), and when cells can change species a limited number of times (k-burnout). Most problems turn out to be at least NP-hard except with very few distinct species (2 or 3).

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Problems, reductions and completeness

Keywords and phrases Chemical Reaction Networks, reconfiguration, hardness

Digital Object Identifier 10.4230/LIPIcs...

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

1 Introduction

The ability to engineer molecules to perform complex tasks is an essential goal of molecular programming. A popular theoretical model for investigating molecular systems and distributed systems is Chemical Reaction Networks (CRNs) [6, 26]. The model abstracts chemical reactions to independent rule-based interactions that creates a mathematical framework equivalent [8] to other well-studied models such as Vector Addition Systems [18] and Petri nets [24]. CRNs are also interesting for experimental molecular programmers, as examples have been built using DNA strand displacement (DSD) [27].

Abstract Surface Chemical Reaction Networks (sCRNs) were introduced in [25] as a way to model chemical reactions that take place on a surface, where the geometry of the surface is used to assist with computation. In this work, the authors gave a possible implementation of the model similar to ideas of spatially organized DNA circuits [21]. This strategy involves DNA strands being anchored to a DNA origami surface. These strands allow for "species" to be attached. Fuel complexes are pumped into the system, which perform the reactions. While these reactions are more complex than what has been implemented in current lab work, it shows a route to building these types of networks.

1.1 Motivation

Feed-Forward circuits using DNA hairpins anchored to a DNA origami surface were implemented in [5]. This experiment used a single type of fuel strand. The copies of the fuel strand attached to the hairpins and were able to drive forward the computation.

A similar model was proposed in [9], which modeled DNA walkers moving along tracks. These tracks have guards that can be opened or closed at the start of computation by including or omitting specific DNA species at the start. DNA walkers have provided interesting implementations such as robots that sort cargo on a surface [29].

A new variant of surface CRNs we introduce is the k-burnout model in which cells can switch states at most k time before being stuck in their final state. This models the practical scenario in which state changes expend some form of limited fuel to induce the state change. Specific experimental examples of this type of limitation can be seen when species encode "fire-once" DNA strand replacement reactions on the surface of DNA origami, as is done within the Signal Passing Tile Model [22].

1.2 Previous Work

The initial paper on sCRNs [25] gave a 1D reversible Turing machine as an example of the computational power of the model. They also provided other interesting constructions such as building dynamic patterns, simulating continuously active Boolean logic circuits, and cellular automata. Later work in [7] gave a simulator of the model, improved some results of [25], and gave many open problems- some of which we answer here.

In [2], the authors introduce the concept of swap reactions. These are reversible reactions that only "swap" the positions of the two species. The authors of [2] gave a way to build feed-forward circuits using only a constant number of species and reactions. These swap reactions may have a simpler implementation and also have the advantage of the reverse reaction being the same as the forward reaction, which makes it possible to reuse fuel species.

A similar idea for swap reactions on a surface that has been studied theoretically are friends-and-strangers graphs [10]. This model was originally introduced to generalize problems such as the 15 Puzzle and Token Swapping. In the model, there is a location graph containing

uniquely labeled tokens and a friends graph with a vertex for every token, and an edge if they are allowed to swap locations when adjacent in the location graph. The token swapping problem can be represented with a complete friends graph, and the 15 puzzle has a grid graph as the location graph and a star as the friends graph (the 'empty square' can swap with any other square). Swap sCRNs can be described as multiplicities friends-and-strangers graph [19], which relax the unique restriction, with the surface grid (in our case the square grid) as the location graph and the allowed reactions forming the edges of the friends graph.

1.3 Our Contributions

In this work, we focus on two main problems related to sCRNs. The first is the reconfiguration problem, which asks given two configurations and a set of reactions, can the first configuration be transformed to the second using the set of reactions. The second is the 1-reconfiguration problem, which asks whether a given cell can ever contain a given species. Our results are summarized in Table 1. The first row of the table comes from the Turing machine simulation in [25] although it is not explicitly stated. The size comes from the smallest known universal reversible Turing machine [20] (see [30] for a survey on small universal Turing machines.)

We first investigate swap reactions in Section 3. We prove both problems are PSPACE-complete using only four species and three swap reactions. For reconfiguration, we show this complexity is tight by showing with three or less species and only swap reactions the problem is in P.

In Section 4, we study a restriction on surface CRNs called k-burnout where each species is guaranteed to only transition k times. This is similar to the freezing restriction from Cellular Automata [14, 15, 28] and Tile Automata [4]. We start with a simple reduction showing reconfiguration is NP-complete in 2-burnout. This is also of interest since the reduction only uses three species types and a reaction set of size one. For 1-reconfiguration, we show the problem is also NP-complete in 1-burnout sCRNs. This reduction uses a constant number of species.

In Section 5, we analyze reconfiguration for all sCRNs that have a reaction set of size one. For the case of only two species, we show for every possible reaction, the problem is solvable in polynomial time. With three species or greater, we show that reconfiguration is NP-complete. The hardness comes from the reduction in burnout sCRNs.

Finally, in Section 6, we conclude the paper by discussing the results as well as many open questions and other possible directions for future research related to surface CRNs.

2 Surface CRN model

Chemical Reaction Network. A *chemical reaction network (CRN)* is a pair $\Gamma = (S, R)$ where S is a set of species and R is a set of reactions, each of the form $A_1 + \cdots + A_j \to B_1 + \cdots + B_k$ where $A_i, B_i \in S$. (We do not define the dynamics of general CRNs, as we do not need them here.)

Surface, Cell, and Species. A *surface* for a CRN Γ is an (infinite) undirected graph G. The vertices of the surface are called *cells*. A *configuration* is a mapping from each cell to a species from the set S. While our algorithmic results apply to general surfaces, our hardness constructions assume the practical case where G is a grid graph, i.e., an induced subgraph of the infinite square grid (where omitted vertices naturally correspond to cells without any species). When G is an infinite graph, we assume there is some periodic pattern of cells that is repeated on the edges of the surface. Figure 1 shows an example set of species and reactions and a configuration of a surface.

Problem	Type	Graph	Species	Rules	Result	Ref
Reconfiguration	sCRN	1D	17	67	PSPACE-complete	[25]
1-Reconfiguration	Swap sCRN	Grid	4	3	PSPACE-complete	Thm. 3
1-Reconfiguration	Swap sCRN	Any	≤ 3	Any	P	Thm. 6
1-Reconfiguration	Swap sCRN	Any	Any	≤ 2	P	Thm. 6
Reconfiguration	Swap sCRN	Grid	4	3	PSPACE-complete	Thm. 4
Reconfiguration	Swap sCRN	Any	≤ 3	Any	P	Thm. 5
Reconfiguration	Swap sCRN	Any	Any	≤ 2	P	Thm. 5
Reconfiguration	2-burnout	Grid	3	1	NP-complete	Thm. 7
1-Reconfiguration	1-burnout	Grid	17	40	NP-complete	Thm. 8
Reconfiguration	sCRN	Grid	≥ 3	1	NP-complete	Cor. 15
Reconfiguration	sCRN	Any	≤ 2	1	P	Thm. 11

Table 1 Summary of our and known complexity results for sCRN reconfiguration problems, depending on the type of sCRN, number of species, and number of rules. All problems are contained in PSPACE, while all k-burnout problems are in NP.

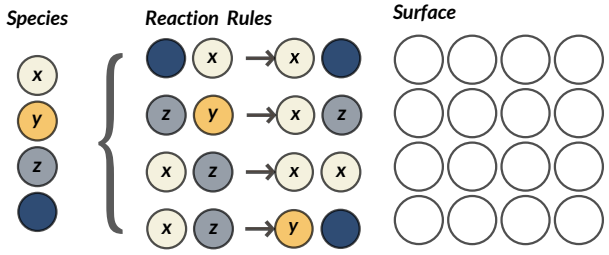


Figure 1 Example sCRN system.

Reaction. A *surface Chemical Reaction Network (sCRN)* consists of a surface and a CRN, where every *reaction* is of the form $A + B \to C + D$ denoting that, when A and B are in neighboring cells, they can be replaced with C and D. A is replaced with C and B with D.

Reachable Configurations. For two configurations I, T, we write $I \to_\Gamma^1 T$ if there exists a $r \in R$ such that performing reaction r on a pair of species in I yields the configuration T. Let $I \to_\Gamma T$ be the transitive closure of $I \to_\Gamma^1 T$, including loops from each configuration to itself. Let $\Pi(\Gamma, I)$ be the set of all configurations T for which $I \to_\Gamma T$ is true. A sequence of reachable states is shown in Figure 2

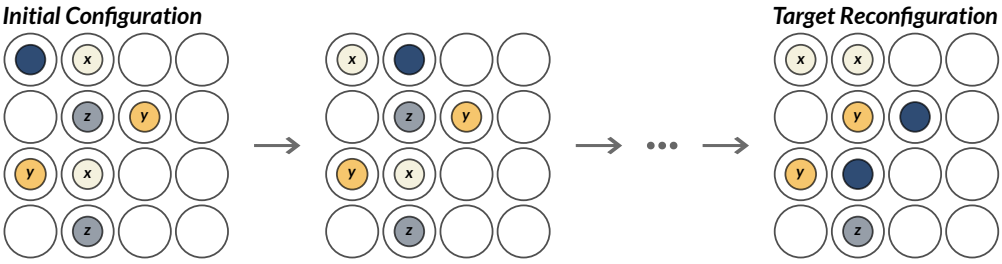


Figure 2 An initial, single step, and target configurations

2.1 Restrictions

Reversible Reactions. A set of reactions R is *reversible* if, for every rule $A + B \to C + D$ in R, the reaction $C + D \to A + B$ is also in R. We may also denote this as a single reversible reaction $A + B \rightleftharpoons C + D$.

Swap Reactions. A reaction of the form $A + B \rightleftharpoons B + A$ is called a *swap reaction*.

***k*-Burnout.** In the k-burnout variant of the model, each vertex of the system's graph can only switch states at most k times (before "burning out" and being stuck in its final state).

2.2 Problems

Reconfiguration Problem. Given a sCRN Γ and two configurations I and T, is $T \in \Pi(\Gamma, S)$?

1-Reconfiguration Problem. Given a sCRN Γ, a configuration I, a vertex v, and a species s, does there exist a $T \in \Pi(\Gamma, S)$ such that T has species s at vertex v?

3 Swap Reactions

In this section, we will show 1-reconfiguration and reconfiguration with swap reactions is PSPACE-complete with only 4 species and 3 swaps in Theorems 3 and 4. We continue by showing that this complexity is tight, that is, reconfiguration with 3 species and swap reactions is tractable in Theorems 5 and 6.

3.1 Reconfiguration is PSPACE-complete

We prove PSPACE-completeness by reducing from the motion planning through gadgets framework introduced in [11]. This is a one player game where the goal is to navigate a robot through a system of gadgets to reach a goal location. The problem of changing the state of the entire system to a desired state has been shown to be PSPACE-complete [1]. This reduction treats the model as a game where the player must perform reactions moving a robot species through the surface.

The Gadgets Framework

Framework. A gadget is a finite set of locations and a finite set of states. Each state is a directed graph on the locations of the gadgets, describing the *traversals* of the gadget. An example can be seen in Figure 3. Each edge (traversal) describes a move the robot can take in the gadget and what state the gadget ends up in if the robot takes that traversal. A robot enters from the start of the edge and leaves at the exit.

In a *system* of gadgets there are multiple gadgets connected by their locations. The *configuration* of a system of gadgets is the state of all gadgets in the system. There is a single robot that starts at a specified location. The robot is allowed to move between connected locations and allowed to move along traversals within gadgets. The system of gadgets can also be restricted to be planar, in which case the cyclic order of the locations on the gadgets is fixed, and the gadgets along with their connections must be embeddable in the plane without crossings.

The *1-player motion planning reachability problem* asks whether there exists a sequence of moves within a system of gadgets which takes the robot from its initial location to a target location. The *1-player motion planning reconfiguration problem* asks whether there exists

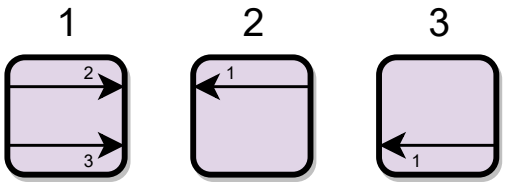


Figure 3 The Locking 2-Toggle (L2T) gadget and its states from the motion planning framework. The numbers above indicate the state and when a traversal happens across the arrows, the gadget changes to the indicated state.

a sequence of moves which brings the configuration of a system of gadgets to some target configuration.

There are many sets of motion planning models and gadgets to build our reduction. We select 1-player over 0-player since in the sCRN model there are many reactions that may occur and we are asking whether there exists a sequence of reactions which reaches some target configuration; in the same way 1-player motion planning asks if there exists a sequence of moves which takes the robot to the target location. The existential query of possible moves/swaps remains the same regardless of whether a player is making decisions vs them occurring by natural processes. The complexity of the gadgets used here are considered in the 0-player setting in [12].

Locking 2-Toggle. The Locking 2-toggle (L2T) is a 4 location, 3 state gadget. The states of the gadget are shown in Figure 3. The L2T has advantages because it universal for reversible deterministic gadgets. Reversibility was important to picking a gadget since swap reactions are naturally reversible.

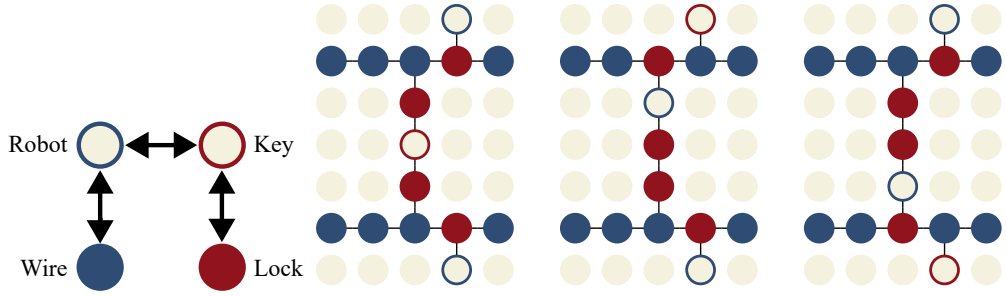
Constructing the L2T

We will show how to simulate the L2T in a swap sCRN system. Planar 1-player motion planning with the L2T was shown to be PSPACE-complete [11]. We now describe this construction.

Species. We utilize 4 species types in this reduction and we name each of them according to their role. First we have the *wire*. The wire is used to create the connection graph between gadgets and can only swap with the robot species. The *robot* species is what moves between gadgets by swapping with the wire and represents the robot in the framework. Each gadget initially contains 2 robot species, and there is one species that starts at the initial location of the robot in the system. The robot can also swap with the key species. Each gadget has exactly 1 *key* species. The key species is what performs the traversal of the gadget by swapping with the lock species. The *lock* species can only swap with the key. There are 4 locks in each gadget. The locks ensure that only legal traversals are possible by the robot species.

These species are arranged into gadgets consisting of two length-5 horizontal tunnels. The two tunnels are connected by a length-3 central vertical tunnel at their 3rd cell. At the 4th cell of both tunnels there is an additional degree 1 cell connected we will call the holding cell.

States and Traversals. The states of the gadget we build are represented by the location of the key species in each gadget. If the key is in the central tunnel of the gadget then we are in state 1 as shown in Figure 4b. Note that in this state the key may swap with the adjacent locks, however we consider these configurations to also be in state 1 and take



(a) Swap rules/species **(b)** State 1 **(c)** State 2 **(d)** State 3

Figure 4 Locking 2-toggle implemented by swap rules. (a) The swap rules and species names. (b-d) The three states of the locking 2-toggle.

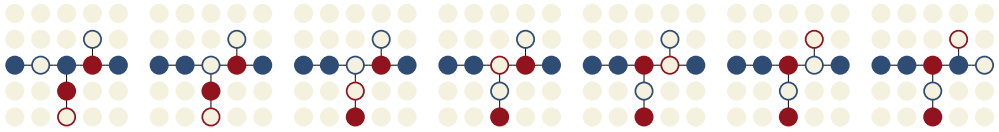


Figure 5 Traversal of the robot species.

advantage of this later. The horizontal tunnels of the gadget in this state contain a single lock with an adjacent robot species.

States 2 and 3 are reflections of each other (Figures 4c and 4d). This state has a robot in the central tunnel and the key in the respective holding cell. The gadget in this state can only be traversed from right to left in one of the tunnels.

Figure 5 shows the process of a robot species traversing through the gadget. Notice when a robot species "traverse" a gadget, it actually traps itself to free another robot at the exit. We prove two lemmas to help verify the correctness of our construction. The lemmas prove the gadgets we design correctly implement the allowed traversals of a locking 2-toggle.

▶ **Lemma 1.** *A robot may perform a rightward traversal of a gadget through the north/south tunnel if and only if the key is moved from the central tunnel to the north/south holding cell.*

Proof. The horizontal tunnels in state 1 allow for a rightward traversal. The robot swaps with wires until it reaches the third cell where it is adjacent to two locks. However the key in the central tunnel may swap with the locks to reach the robot. The key and robot then swap. The key is then in the horizontal tunnel and can swap to the right with the lock there. It may then swap with the robot in the holding cell. This robot then may continue forward to the right and the key is stuck in the holding cell.

Notice when entering from the left the robot will always reach a cell adjacent to lock species. The robot may not swap with locks so it cannot traverse unless the key is in the central tunnel. ◀

▶ **Lemma 2.** *A robot may perform a leftward traversal of a gadget through the north/south tunnel if and only if the key is moved from the north/south holding cell to the central tunnel.*

Proof. In state 2 the upper tunnel can be traversed and in state 3 the lower tunnel can be traversed. The swap sequence for a leftward traversal is the reverse of the rightward traversal, meaning we are undoing the swaps to return to state 1. The robot enters the gadget and

swaps with the key, which swaps with the locks to move adjacent to the central tunnel. The key then returns to the central tunnel by swapping with the robot. The robot species can then leave the gadget to the left.

A robot entering from the right will not be able to swap to the position adjacent to the holding cell if it contains a lock. This is true in both tunnels in state 1 and in the non-traversable tunnels in states 2 and 3. ◀

We use these lemmas to first prove PSPACE-completeness of 1-reconfiguration. We reduce from the planar 1-player motion planning reachability problem.

▶ **Theorem 3.** 1-*reconfiguration is PSPACE-complete with* 4 *species and* 3 *swap reactions or greater even when the surface is a subset of the grid graph.*

Proof. Given a system of gadgets create a surface encoding the connection graph between the locations. Each gadget is built as described above in a state representing the initial state of the system. Ports are connected using multiple cells containing wire species. When more than two ports are connected we use degree-3 cells with wire species. The target cell for 1-reconfiguration is a cell containing a wire located at the target location in the system of gadgets.

If there exists a solution to the robot reachability problem then we can convert the sequence of gadget traversals to a sequence of swaps. The swaps relocate a robot species to the location as in the system of gadgets.

If there exists a swap sequence to place a robot species in the target cell there exists a solution to the robot reachability problem. Any swap sequence either moves an robot along a wire, or traverses it through a gadget. From Lemmas 1 and 2 we know the only way to traverse a gadget is to change its state (the location of its key) and a gadget can only be traversed in the correct state. ◀

Now we show Reconfiguration in sCRNs is hard with the same set of swaps is PSPACE-complete as well. We do so by reducing from the Targeted Reconfiguration problem which asks, given an initial and target configuration of a system of gadgets, does there exist sequence of gadget traversals to change the state of the system from the initial to the target and has the robot reach a target location. Note prior work only shows reconfiguration (without specifying the robot location) is PSPACE-complete[1] however a quick inspection of the proof of Theorem 4.1 shows the robot ends up at the initial location so requiring a target location does not change the computational complexity for the locking 2-toggle. One may also find it useful to note that the technique used in [1] for gadgets and in [17] for Nondeterministic Constraint Logic can be applied to reversible deterministic systems more generally. This means the method described in those could be used to give an alternate reduction directly from 1-reconfiguration of swap sCRNs to reconfiguration of swap sCRNs.

▶ **Theorem 4.** *Reconfiguration is PSPACE-complete with* 4 *species and* 3 *swap reactions or greater.*

Proof. Our initial and target configurations of the surface are built with the robot species at the robots location in the system of gadget, and each key is placed according to the starting configuration of the gadget.

Again as in the previous theorem we know from Lemmas 1 and 2 the robot species traversal corresponds to the traversals of the robot in the system of gadgets. The target surface can be reached if and only the target configuration in the system of gadgets is reachable. ◀

3.2 Polynomial-Time Algorithm

Here we show that the previous two hardness results are tight: when restricting to a smaller cases, both problems become solvable in polynomial time. We prove this by utilizing previously known algorithms for *pebble games*, where labeled pebbles are placed on a subset of nodes of a graph (with at most one pebble per node). A *move* consists of moving a pebble from its current node to an adjacent empty node. These pebble games are again a type of multiplicity friends-and-strangers graph.

▶ **Theorem 5.** *Reconfiguration is in P with* 3 *or fewer species and only swap reactions. Reconfiguration is also in P with* 2 *or fewer swap reactions and any number of species.*

Proof. First we will cover the case of only two swap reactions. There are two possibilities: the two reactions share a common species or they do not. If they do not, we can partition the problem into two disjoint problems, one with only the species involved in the first reaction and the other with only the species from the second reaction. Each of these subproblems has only one reaction, and is solvable if and only if each connected component of the surface has the same number of each species in the initial and target configurations.

The only other case is where we have three species, A, B, and C, where A and C can swap, B and C can swap, but A and B cannot swap. In this case, we can model it as a pebble motion problem on a graph. Consider the graph of the surface where we put a white pebble on each A species vertex, a black pebble on each B species vertex, and leave each C species vertex empty. A legal swap in the surface CRN corresponds to sliding a pebble to an adjacent empty vertex. Goraly et al. [16] gives a linear-time algorithm for determining whether there is a feasible solution to this pebble motion problem. Since the pebble motion problem is exactly equivalent to the surface CRN reconfiguration problem, the solution given by their algorithm directly says whether our surface CRN problem is feasible. ◀

▶ **Theorem 6.** 1*-reconfiguration is in P with* 3 *or fewer species and only swap reactions.* 1*-reconfiguration is also in P with* 2 *or fewer swap reactions.*

Proof. If there are only two swap reactions, we again have two cases depending on whether they share a common species. If they do not share a common species, then we only need to consider the rule involving the target species. The problem is solvable if and only if the connected component of the surface of species involved in this reaction containing the target cell also has at least one copy of the target species. Equivalently, if the target species is A, and A and B can swap, then there must either be A at the target location or a path of B species from the target location to the initial location of an A species.

The remaining case is when we again have three species, A, B, and C, where A and C can swap, B and C can swap, but A and B cannot swap. If C is the target species, then the problem is always solvable as long as there is any C in the initial configuration. Otherwise, suppose without loss of generality that the target species is A. Some initial A must reach the target location. For each initial A, consider the modified problem which has only that single A and replaces all of the other copies of A with B. A sequence of swaps is legal in this modified problem if and only if it was legal in the original problem. The original problem has a solution if and only if any of the modified ones do. We then convert each of these problems to a robot motion planning problem on a graph: place the robot at the vertex with a single copy of A, and place a moveable obstacle at each vertex with a B. A legal move is either sliding the robot to an adjacent empty vertex or sliding an obstacle to an adjacent empty vertex. Papadimitriou et al. [23] give a simple polynomial time algorithm for determining whether it is possible to get the robot to a given target location. By applying their algorithm

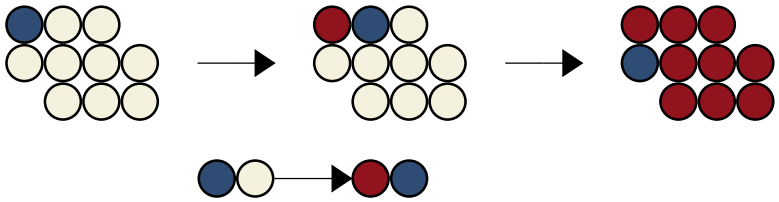


Figure 6 An example reduction from Hamiltonian Path. We are considering graphs on a grid, so any two adjacent locations are connected in the graph. Left: an initial board with the starting location in blue. Middle: One step of the reaction. Right: The target configuration with the ending location in blue. Bottom: the single reaction rule.

to each of these modified problems (one for each cell that has an initial A), we can determine whether any of them have a solution in polynomial time (since there are only linearly many such problems), and thus determine whether the original 1-reconfiguration problem has a solution in polynomial time.

◀

4 Burnout

In this section, we show reconfiguration in 2-burnout with species (A, B, C) and reaction $A + B \to C + A$ is NP-complete in Theorem 7. Next, we show 1-reconfiguration in 1-burnout with 17 species and 40 reactions is NP-complete in Theorem 8.

Reconfiguration and 1-Reconfiguration for burnout sCRNs are in NP since there is the length of any reconfiguration is bounded. For space we do not include this proof but note this has been proved in other system such as Resource Bounded Cellular Automata [13], Freezing Cellular Automata [14] and Freezing Tile Automata [3].

4.1 2-Burnout Reconfiguration

This is a simple reduction from Hamiltonian Path, specifically when we have a stated start and end vertex.

▶ **Theorem 7.** *Reconfiguration in 2-burnout sCRNs with species (A, B, C) and reaction $A + B \to C + A$ is NP-complete even when the surface is a subset of the grid graph. It is also NP-complete with the same species and reactions without the 2-burnout restriction.*

Proof. Let $\Gamma = \{(A, B, C), (A + B \to C + A)\}$. Given an instance of the Hamiltonian path problem on a grid graph H with a specified start and target vertex v_s and v_t, respectively, create a surface G where each cell in G is a node from H. Each cell contains the species B except for the cell representing v_s which contains species A. The target surface has species C in every cell except for the final node containing A, v_t. An example can be seen in Figure 6.

The species A can be thought of as an agent moving through the graph. The species B represents a vertex that hasn't been visited yet, while the species C represents one that has been. Each reaction moves the agent along the graph, marking the previous vertex as visited.

($\Rightarrow$) If there exists a Hamiltonian path, then the target configuration is reachable. The sequence of edges in the path can be used as a reaction sequence moving the agent through the graph, changing each cell to species C finishing at the cell representing v_t.

($\Leftarrow$) If the target configuration is reachable, there exists a Hamiltonian path. The sequence of reactions can be used to construct the path that visits each of the vertices exactly once,

ending at v_t.

Note that we have not discussed the effect of Burnout on the reduction. However since each cell transitions through species in the following order: B, A, C this reaction always results in a 2-burnout sCRN so the reduction holds with and without the restriction.

This means the CRN is 2-burnout which bounds the max sequence length for reaching any reachable surface, putting the reconfiguration problem in NP. ◀

4.2 1-Burnout 1-Reconfiguration

For 1-burnout 1-reconfiguration, we show NP-completeness by reducing from 3SAT and utilizing the fact that once a cell has reacted it is burned out and can no longer participate in later reactions.

Figure 7 All the possible configurations of two variable gadgets.

▶ **Theorem 8.** *1-reconfiguration in 1-burnout sCRNs with* 17 *species and* 40 *reactions is NP-complete even when the surface is a subset of the grid graph. It is also NP-complete with the same species and reactions without the 1-burnout restriction.*

Proof. We reduce from 3SAT. The idea is to have an 'agent' species traverse the surface to assign variables and check that the clauses are satisfied by 'walking' through each clause. If the agent can traverse the whole surface and mark the final vertex as 'satisfied', there is a variable assignment that satisfies the original 3SAT instance.

Variable Gadget. The variable gadget is constructed to allow for a nondeterministic assignment of the variable via the agent walk. At each intersection, the agent 'chooses' a path depending on the reaction that occurs. If the agent chooses 'true' for a given variable, it will walk up then walk down to the center species. If the agent chooses 'false', the agent will walk down then walk up to the center species. From the center species, the agent can only continue following the path it chose until it reaches the next variable gadget. Examples of the agent assigning variables can be seen in Figure 7.

Each variable assignment is 'locked' by way of geometric blocking. When the agent encounters a variable gadget whose variable has already been assigned, the agent must follow that same assignment or it will get 'stuck' trying to react with a burnt out vertex. This can be seen in Figure 8.

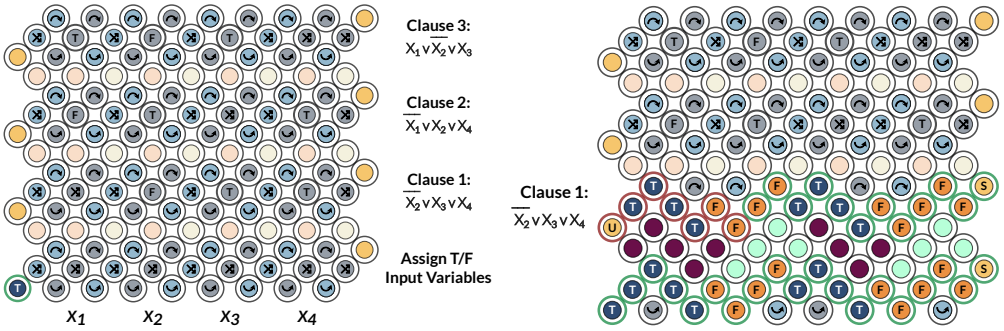
Initial Configuration. First, the configuration is constructed with variable gadgets connected in a row, one for each variable in the 3SAT instance. This row of variable gadgets is where the agent will nondeterministically assign values to the variables. Next, a row of variable gadgets, one row for each clause, is placed on top of the assignment row, connected with helper species to fill in the gaps.

For each clause, if a certain variable is present, the center species of the variable gadget reflects its literal value from the clause. For example, if the variable $x1$ in clause $c1$ should be true to satisfy the clause, the variable gadget representing $x1$ in $c1$'s row will contain a T species in the center cell. Lastly, the agent species is placed in the bottom left of the configuration. An example configuration can be seen in Figure 9.

The agent begins walking and nondeterministically assigns a value to each variable. After assigning every variable, the agent walks right to left. If at an intersection, the agent chooses

(a) Successful navigation of an intersection. **(b)** Agent stuck due to not following the assignment.

Figure 8 The assignment 'locking' process.



(a) Example starting configuration. **(b)** The surface after evaluating the first clause.

Figure 9 Reduction from 3SAT to 1-burnout 1-reconfiguration. (a) The starting configuration of the surface for the example formula $\varphi = (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3)$. (b) The configuration after evaluating the first clause. A red outline represents the unsatisfied state, and a green outline represents the satisfied state.

a different assignment than it did its first pass, the agent becomes 'stuck' only being able to react with a burnt out vertex.

After walking all the way to the left, the first clause can be checked. The agent starts in the unsatisfied state, walking through each variable in the row, left to right. If the current variable assignment at a variable gadget satisfies this clause, the agent changes to the satisfied state and continues walking. If the agent walks through all the variables without becoming satisfied, the computation ends. If the clause was satisfied, the agent continues by walking back, right to left, to begin evaluation of the next clause. If the agent walks all the way to the final vertex with a satisfied state, then the initial variable assignment satisfies all the clauses.

($\Rightarrow$) If there exists a variable assignment that satisfies the 3SAT instance, then the final vertex can be marked with the satisfied state s. The agent can only mark the final cell with the satisfied state s if all clauses can be satisfied.

($\Leftarrow$) If the final vertex can be marked with satisfied state s, there exists a variable assignment that satisfies the 3SAT instance. The variable assignment that the agent non-deterministically chose can be read and used to satisfy the 3SAT instance. ◀

5 Single Reaction

When limited to a single reaction, we show a complete characterization of the reconfiguration problem. There exists a reaction using 3 species for which the problem is NP-complete. For all other cases of 1 reaction, the problem is solvable in polynomial time.

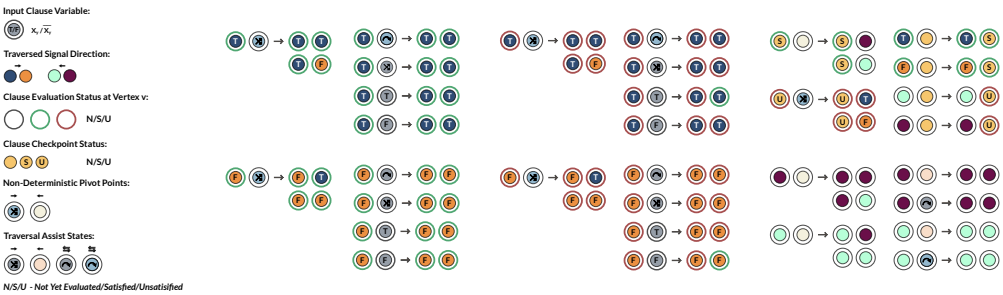


Figure 10 Species identification and transition rules for 1-burnout 1-reconfiguration.

5.1 2 Species

We start with proving reconfiguration is in P when we only have 2 species and a single reaction.

▶ **Lemma 9.** *Reconfiguration with species* $\{A, B\}$ *and reaction* $A + A \to A + B$ *OR* $A + B \to A + A$ *is solvable in polynomial time on any surface.*

Proof. The reaction $A + B \to A + A$ is the reverse of the first case. By flipping the target and initial configurations, we can reduce from reconfiguration with $A + B \to A + A$ to reconfiguration $A + A \to A + B$.

We now solve the case where we have the reaction $A + A \to A + B$.

All cells that start and end with species B can be ignored as they do not need to be changed, and can not participate in any reactions. If there is a cell that contains B in the initial configuration but A in the target, the instance is 'no' as B may never become A.

Let any cell that starts in species A but ends in species B be called a *flip* cell, and any species that starts in A and stays in A a *catalyst* cell.

An instance of reconfiguration with these reactions is solvable if and only if there exists a set of spanning trees, each rooted at a catalyst cell, that contain all the flip cells. Using these trees, we can construct a reaction sequence from post-order traversals of each spanning tree, where we have each non-root node react with its parent to change itself to a B. In the other direction, given a reaction sequence, we can construct the spanning trees by pointing each flip cell to the neighbor it reacts with. ◀

▶ **Lemma 10.** *Reconfiguration with species* $\{A, B\}$ *and reaction* $A + A \to B + B$ *is solvable in polynomial time on any surface.*

Proof. Reconfiguration in this case can be reduced to perfect matching. Create a graph M including a node for each cell in S containing the A species initially and containing B in the target, with edges between nodes of neighboring cells. If M has a perfect matching, then each edge in the matching corresponds to a reaction that changes A to B. If the target configuration is reachable, then the reactions form a perfect matching since they include each cell exactly once. ◀

▶ **Theorem 11.** *Reconfiguration with* 2 *species and* 1 *reaction is in P on any surface.*

Proof. As we only have two species and a single reaction, we can analyze each of the four cases to show membership in P. We divide into two cases:

$A + A$: When a species reacts with itself, it can either change both species, which is shown to be in P by Lemma 10; or it changes only one of the species, which is in P by Lemma 9.

$A + B$: When two different species react, they can either change to the same species, which is in P by Lemma 9; or they can both change, which is a swap and thus is in P by Theorem 5. ◀

5.2 3 or more Species

Moving up to 3 species and 1 reaction, we showed earlier that there exists a reaction for which reconfiguration is NP-complete in Theorem 7. Here, we give reactions for which reconfiguration between 3 species is in P, and in Corollary 15 we prove that all remaining reactions are isomorphic to one of the reactions we've analyzed.

▶ **Lemma 12.** *Reconfiguration with species (A, B, C) and reaction $A+B \to C+C$ is solvable in polynomial time on any surface.*

Proof. At a high level, we create a new graph of all the cells that must change to species C, and add an edge when the two cells can react with each other. Since a reaction changes both cells to C we can think of the reaction as "covering" the two reacting cells. Finding a perfect matching in this new graph will give a set of edges along which to perform the reactions to reach the target configuration.

Consider a surface G and a subgraph $G' \subseteq G$ where we include a vertex v' in G' for each cell that contain A or B in the initial configuration and C in the target configuration. We include an edge (u', v') between any vertices in G' that contain different initial species, i.e. any pair of cell which one initially contains A and the other initially B.

Reconfiguration is possible if and only if there is a perfect matching in G'. If there is a perfect matching then there exists a set of edges which cover each cell once. Since G' represents the cells that must change states, and the edges between them are reactions, the covering can be used as a sequence of pairs of cells to react. If there is a sequence of reactions then there exists a perfect matching in G': each cell only reacts once so the matching must be perfect, and the cells that react have edges between them in G'. ◀

▶ **Lemma 13.** *Reconfiguration with species (A, B, C) and reaction $A+B \to A+C$ is solvable in polynomial time on any surface.*

Proof. The instance of reconfiguration is solvable if and only if any cell that ends with species C either contained C in the initial configuration, or started with species B and have an A adjacent to perform the reaction. Additionally, since a reaction cannot cause a cell to change to A or B, each cell with an A or B in the target configuration must contain the same species in the initial configuration. ◀

The final case we study is 4 species 1 reaction. Any sCRN with 5 or more species and 1 reaction has a species which is not included in the reaction.

▶ **Lemma 14.** *Reconfiguration with species (A, B, C, D) and the reaction $A + B \to C + D$ is in P on any surface.*

Proof. We can reduce Reconfiguration with $A + B \to C + D$ to perfect matching similar to Lemma 12. Create a new graph with each vertex representing a cell in the surface that must change species. Add an edge between each pair of neighboring cells that can react (between one containing A and the other B). A perfect matching then corresponds to a sequence of reactions that changes each of the species in each cell to C or D. ◀

▶ **Corollary 15.** *Reconfiguration with* 3 *or greater species and* 1 *reaction is NP-complete on any surface.*

Proof. First, from Theorem 7 we see that there exists a case of reconfiguration with 3 species that is NP-hard with or without the burnout restriction.

For membership in NP, we analyze each possible reaction. We note that we only need to consider two cases for the left hand side of the rule, $A + A$ and $A + B$. Any other reaction is isomorphic to one of this form as we can relabel the species. For example, rule $B+C \to A+A$ can be relabeled as $A + B \to C + C$. Also, we know that C must appear somewhere in the right hand side of the rule. If it does not then the reaction only takes place between two species, which is always polynomial time as shown above, or it involves a species we can relabel as C.

Here are the cases for $A + B$ and our analysis results:

$A + B \to A + C$	P in Lemma 13
$A + B \to C + B$	P in Lemma 13 under isomorphism
$A + B \to C + A$	NP in Theorem 7
$A + B \to B + C$	NP in Theorem 7 under isomorphism
$A + B \to C + C$	P in Lemma 12
$A + B \to C + D$	P in Lemma 14

When we have $A + A$ on the left side of the rule, the only case we must consider is $A + A \to B + C$ (since all 3 species must be included in the rule). We have already solved this reaction: first swap the labels of A and C giving rule $C + C \to B + A$, then reverse the rule to $B + A \to C + C$ and swap the initial and target configuration. Finally since rules do not care about orientation this is equivalent to the rule $A + B \to C + C$ in Lemma 12.

Finally, for 4 species and greater, the only new case is $A + B \to C + D$, which is proven to be in P in Lemma 14. Any other case would have species that are not used since a rule can only have 4 different species in it.

Thus, all cases are either in NP, or in P which is a subset of NP, therefore, the problem is in NP. Also, since our results for each case apply for any surface, the same is true in general. ◀

6 Conclusion

In this paper, we explored the complexity of the configuration problem within natural variations of the surface CRN model. While general reconfiguration is known to be PSPACE-complete, we showed that it is still PSPACE-complete even with several extreme constraints. We first considered the case where only swap reactions are allowed, and showed reconfiguration is PSPACE-complete with only four species and three distinct reaction types. We further showed that this is the smallest possible number of species for which the problem is hard by providing a polynomial-time solution for three or fewer species when only using swap reactions.

We next considered surface CRNs with rules other than just swap reactions. First, we considered the burnout version of the reconfiguration problem, and then followed by the normal version with small species counts. In the case of 2-burnout, we showed reconfiguration is NP-complete for three species and one reaction type, and 1-burnout is NP-complete for 17 species with 40 distinct reaction types. Without burnout, we achieved, as a corollary,

that three species, one reaction type is NP-complete while showing that dropping the species count down to two yields a polynomial-time solution.

6.1 Computing Polynomial Space Functions

An interpretation of Theorem 3 is that surface Chemical Reactions are capable of computing any function that can be computed in polynomial space. Perhaps the most important PSPACE-Complete is the acceptance problem for polynomial space Turing machines. While there may be a few reduction between these problems, we can may turn any polynomial space Turing machine into a surface CRN such that the robot species swaps with a wire species at a target location. In experiments one can imagine the target location as having a special type of wire species that acts as a reporting, emitting a signal when it reacts with the robot species. The size of the surface is polynomial in the space of the Turing machine since these are all polynomial time reductions. While we do not claim this experiment can be done with such a small number of species, but rather that theoretically more sequence efficient reaction systems which can compute should exists by taking advantage of the surface.

Our polynomial time algorithms describe experiments with 1, 2, or 3 reactions on surfaces where well studied algorithms for problems such as matching and motion planning may be of use.

6.2 Open Problems

This work introduced new concepts that leaves open a number of directions for future work. While we have fully characterized the complexity of reconfiguration for the swap-only version of the model, the complexity of reconfiguration with general rule types for three species systems remains open if the system uses more than one rule. All of hardness results also use a square grid graph, while our algorithms work on general surfaces. We would like to know if the threshold for hardness can be lowered on more general graphs. In the 1-burnout variant of the model, we have shown 1-reconfiguration to be NP-complete, but the question of general reconfiguration remains a "burning" open question.

References

1 Joshua Ani, Erik D. Demaine, Yevhenii Diomidov, Dylan Hendrickson, and Jayson Lynch. Traversability, reconfiguration, and reachability in the gadget framework. In *WALCOM: Algorithms and Computation: 16th International Conference and Workshops, WALCOM 2022, Jember, Indonesia, March 24–26, 2022, Proceedings*, pages 47–58. Springer, 2022.

2 Tatiana Brailovskaya, Gokul Gowri, Sean Yu, and Erik Winfree. Reversible computation using swap reactions on a surface. In *International Conference on DNA Computing and Molecular Programming*, pages 174–196. Springer, 2019.

3 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Verification and computation in restricted tile automata. *Natural Computing*, pages 1–19, 2020.

4 Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In *DNA Computing and Molecular Programming: 24th International Conference, DNA 24, Jinan, China, October 8–12, 2018, Proceedings 24*, pages 155–172. Springer, 2018.

5 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, 2017.

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

7 Samuel Clamons, Lulu Qian, and Erik Winfree. Programming and simulating chemical reaction networks on a surface. *Journal of the Royal Society Interface*, 17(166):20190790, 2020.

8 Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In *Algorithmic bioprocesses*, pages 543–584. Springer, 2009.

9 Frits Dannenberg, Marta Kwiatkowska, Chris Thachuk, and Andrew J Turberfield. DNA walker circuits: computational potential, design, and verification. *Natural Computing*, 14(2):195–211, 2015.

10 Colin Defant and Noah Kravitz. Friends and strangers walking on graphs. *Combinatorial Theory*, 1, 2021.

11 Erik D. Demaine, Isaac Grosof, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In *9th International Conference on Fun with Algorithms (FUN 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

12 Erik D Demaine, Robert A Hearn, Dylan Hendrickson, and Jayson Lynch. Pspace-completeness of reversible deterministic systems. In *Machines, Computations, and Universality: 9th International Conference, MCU 2022, Debrecen, Hungary, August 31–September 2, 2022, Proceedings*, pages 91–108. Springer, 2022.

13 Alberto Dennunzio, Enrico Formenti, Luca Manzoni, Giancarlo Mauri, and Antonio E Porreca. Computational complexity of finite asynchronous cellular automata. *Theoretical Computer Science*, 664:131–143, 2017.

14 Eric Goles, Diego Maldonado, Pedro Montealegre, and Martín Ríos-Wilson. On the complexity of asynchronous freezing cellular automata. *Information and Computation*, 281:104764, 2021.

15 Eric Goles, Nicolas Ollinger, and Guillaume Theyssier. Introducing freezing cellular automata. In *Cellular Automata and Discrete Complex Systems, 21st International Workshop (AUTOMATA 2015)*, volume 24, pages 65–73, 2015.

16 Gilad Goraly and Refael Hassin. Multi-color pebble motion on graphs. *Algorithmica*, 58:610–636, 2010.

17 Robert A Hearn and Erik D Demaine. *Games, puzzles, and computation*. CRC Press, 2009.

18 Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi:10.1016/S0022-0000(69)80011-5.

19 Aleksa Milojevic. Connectivity of old and new models of friends-and-strangers graphs. *arXiv preprint arXiv:2210.03864*, 2022.

20 Kenichi Morita and Yoshikazu Yamaguchi. A universal reversible turing machine. In *Machines, Computations, and Universality: 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007. Proceedings 5*, pages 90–98. Springer, 2007.

21 Richard A. Muscat, Karin Strauss, Luis Ceze, and Georg Seelig. DNA-based molecular architecture with spatially localized components. *ACM SIGARCH Computer Architecture News*, 41(3):177–188, 2013.

22 Jennifer Padilla, Wenyan Liu, and Nadrian Seeman. Hierarchical self assembly of patterns from the robinson tilings: Dna tile design in an enhanced tile assembly model. *Natural computing*, 11:323–338, 06 2012. doi:10.1007/s11047-011-9268-7.

23 Christos H Papadimitriou, Prabhakar Raghavan, Madhu Sudan, and Hisao Tamaki. Motion planning on a graph. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 511–520. IEEE, 1994.

24 Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, 1962.

25 Lulu Qian and Erik Winfree. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In *DNA Computing and Molecular Programming: 20th International Conference, DNA 20, Kyoto, Japan, September 22-26, 2014. Proceedings*, volume 8727, page 114. Springer, 2014.

26 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7:615–633, 2008.

27 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.

28 Guillaume Theyssier and Nicolas Ollinger. Freezing, bounded-change and convergent cellular automata. *Discrete Mathematics & Theoretical Computer Science*, 24, 2022.

29 Anupama J. Thubagere, Wei Li, Robert F. Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjan Srinivas, Damien Woods, et al. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017.

30 Damien Woods and Turlough Neary. The complexity of small universal turing machines: A survey. *Theoretical Computer Science*, 410(4-5):443–450, 2009.

APPENDIX G

APPENDIX G

RECONFIGURATION OF LINEAR SURFACE CHEMICAL REACTION NETWORKS WITH BOUNDED STATE CHANGE

Reconfiguration of Linear Surface Chemical Reaction Networks with Bounded State Change

Robert M. Alaniz* Michael Coulombe† Erik D. Demaine† Bin Fu* Timothy Gomez†
Elise Grizzell* Ryan Knobel* Andrew Rodriguez* Robert Schweller* Tim Wylie*

Abstract

We present results on the complexity of reconfiguration of surface Chemical Reaction Networks (sCRNs) in a model where surface vertices can change state a bounded number of times based on a given burnout parameter k. We primarily focus on linear $1 \times n$ surfaces. Without a burnout bound, or even with an exponentially high bound on burnout, reconfiguration on linear surfaces is known to be PSPACE-complete. In contrast, we show that the problem becomes NP-complete when the burnout k is polynomially bounded in n. For smaller $k = O(1)$, we show the problem is polynomial-time solvable, and in the special case of $k = 1$ burnout, reconfiguration can be solved in linear $O(n + |R|)$ time, where $|R|$ denotes the number of system rules. We additionally explore some extensions of this problem to more general graphs, including a fixed-parameter tractable algorithm in the height m of an $m \times n$ rectangle in 1-burnout, a polynomial-time solution for 1-burnout in general graphs if reactions are non-catalytic, and an NP-complete result for 1-burnout in general graphs.

1 Introduction

A prominent area of research in molecular computation, Chemical Reaction Networks (CRNs), study well-mixed solutions of molecules. Limited by the inherent lack of geometry, the model has important restrictions on its computational power, including no proven capability of error-free computation of logarithm [6] or Turing universality [16]. Specifically, CRNs are capable of computing all semilinear functions [5]. The introduction of a surface and, by extension, geometry, with abstract Surface Chemical Reaction Networks (sCRNs) removes these limitations, and thus has increased computational power. Molecular computing on a surface is an increasingly popular direction in both experimental [4, 18] and theoretical [10, 13] research.

In this paper, we explore a restricted version of the powerful surface CRN model, where each molecule in the system can only change in a reaction a set number of times. We refer to this constraint as *burnout*. Bounding the number of state changes leads to polynomial-time and XP algorithms for many reconfiguration problems that are otherwise PSPACE-complete.

Motivations for the study of problems with burnout include examples such as optimizing limited lifetime biomolecules or modeling redox reactions in which the electron transfer from one chemical species to another increases the cost of further reaction beyond what any other current or future neighbors could afford.

1.1 Previous Work

Surface Chemical Reaction Networks (sCRNs) were introduced in [15] with a simulator provided in [7]. These papers show various constructions such as Boolean circuits and a Cellular Automata simulation.

Another restricted version of sCRNs uses only swap reactions, in which the two species only change position, Example: $A + B \rightarrow B + A$. In [2], the authors show swap reactions are capable of feed-forward computation and provide an analysis of thermodynamic properties of the circuit. Recently, [1] showed that reconfiguration is PSPACE-complete for swap surface CRNs with only 4 species and 3 reactions, and in P with any system of fewer species or reactions. This work also introduces k-burnout surface CRNs and show two important results: that 1-reconfiguration (whether a single cell can change) is NP-complete with 1-burnout and that general reconfiguration is NP-complete with 2-burnout. Burnout is similar to the freezing concept from Cellular Automata [11, 12, 17] and Tile Automata [3], but while freezing is defined as having an ordering on states or a tile never revisiting a state, burnout is a constraint where a cell never reacts more than a fixed number of times. Thus, returning to a previous state is possible, unlike the freezing restrictions.

1D Cellular Automata are capable of Turing computation from [8]. P-completeness of prediction, is this cell in state at time step less than t, for Cellular Automata Rule 110 was shown in [14], implying it is also capable of efficient computation. This problem is also P-complete for a number of Freezing CAs in 2D, while it is always in NL for Freezing 1D CAs [12]. This work also gives a 1D freezing CA, which is Turing universal.

*Department of Computer Science, University of Texas Rio Grande Valley
†Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology

Shape	Burnout	Result	Theorem
$1 \times n$	1	$O(n + \|R\|)$	Thm. 1
$1 \times n$	2	$O(n \cdot \|S\|^2 \cdot \|R\|^4)$	Thm. 2
$1 \times n$	$O(1)$	P for $O(1)$ degree	Thm. 3
$1 \times n$	k (unary)	NP-complete	Thm. 4
$1 \times n$	Unbounded	PSPACE-complete	[15]
Planar	1	$O(\|V\|^{1.5} + \|R\|)$	Thm. 5**
General	1	NP-complete	Thm. 7
$m \times n$	1	NP-complete	[1]‡
$m \times n$	1	FPT in m	Thm. 8‡

Table 1: Comparison of reconfiguration results. For a CRN system, R is the set of rules and S is the set of species. V is the set of vertices for the graph defining the shape. **Non-catalytic rules only. ‡These results are for the problem of 1-Reconfiguration.

1.2 Our Contributions

This work investigates the reconfiguration problem for linear surface CRNs with k-burnout. Our results are outlined in Table 1. We begin in Section 3, where we present a polynomial-time algorithm for 1D 1-burnout. We then increase the burnout number to investigate 1D 2-burnout systems and prove that this is still in P. Following this, we show that for the case of any fixed $k = \mathcal{O}(1)$, there exists an algorithm that has a polynomial runtime. In the terms of parameterized complexity classes, this is the class XP, also known as slice-wise polynomial [9]. We then present an NP-completeness proof for when the burnout is a unary input. This result contrasts PSPACE-completeness known when the burnout is unbounded or exponentially high [15].

After $1 \times n$ lines, we begin investigating 1-burnout in 2D systems in Section 5. We start with the problem of reconfiguration, where we only have non-catalytic rules. We then show that on an arbitrary graph and with all types of rules, the reconfiguration problem is NP-complete. Finally, we study the problem of 1-reconfiguration for bounded-height surfaces, presenting an XP algorithm parameterized by height.

2 Preliminaries

A brief overview of the model and relevant problems.

Surface, Cells and Species. A *surface* for a CRN Γ is an undirected graph G of large size n. The vertices of the surface are also referred to as *cells*. Many of our results deal with $1 \times n$ grid graphs, or *linear* surfaces.

The state of a vertex is representative of a molecular *species* in the system. Chemical *reactions* considered here are bimolecular, as in they occur between two species in neighboring vertices. A rule denoting that neighboring species A and B may react to become C and D is written as $A + B \to C + D$. This is a **non-catalytic** rule, as both species change. In a **catalytic** reaction, only one of the two species will change, e.g. $C + D \to C + B$, the other used as a catalyst.

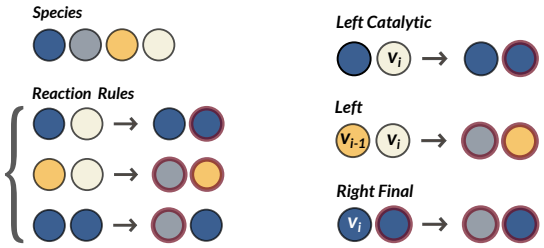


Figure 1: (a) An example sCRN system with 4 species, three rules, and 1 burnout. (b) Rule types used in Figure 2 example. *Note: The red ring outline shows whether the vertex has been "burned out." There is no effect on the reaction rule itself.*

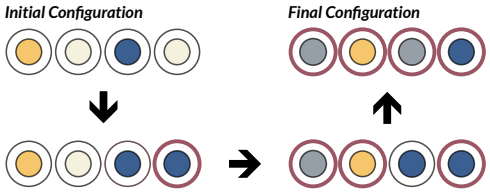


Figure 2: A possible sequence of reactions for the system described in Figure 1

A **surface Chemical Reaction Network (sCRN)** consists of a surface, a set of molecular species S, and a set of reaction rules R. A *configuration* is a mapping from each vertex to a species from the set S.

Reachable Configurations. For two configurations I, T, we write $I \to^1_\Gamma T$ if there exists a $r \in R$ such that performing reaction r on a pair of species in I yields the configuration T. Let $I \to_\Gamma T$ be the transitive closure of $I \to^1_\Gamma T$, including loops from each configuration to itself. Let $\Pi(\Gamma, I)$ be the set of all configurations T where $I \to_\Gamma T$ is true.

Burnout. A limit on the number of changes that can occur in any vertex v_i. In systems that allow catalytic reactions, after this limit has been reached, while v_i will not change again, neighboring species may still use the species in that cell as a catalyst.

Reconfiguration Problem. Given an sCRN Γ and two configurations I and T, is $T \in \Pi(\Gamma, I)$?

1-Reconfiguration Problem. Given an sCRN Γ, configuration I, vertex v, and species s, does there exist a $T \in \Pi(\Gamma, I)$ such that T has species s at vertex v?

3 Algorithms for Constant Burnout

We show that reconfiguration of a linear surface is solvable in polynomial-time when the burnout is one or two.

3.1 1-Burnout Linear Surfaces

In the case of 1-burnout with a $1 \times n$ line, the problem of reconfiguration is solvable in linear time with respect to n and the size of the rule set. As an observation, there are at most six reactions for any vertex, v_i, on a

linear surface since a vertex has at most two neighbors. These reactions include the following:

- A left reaction, where vertex v_i reacts with vertex v_{i-1} and both vertices reach their final states.
- A left catalytic reaction, where vertex v_i reacts with vertex v_{i-1} in its initial state to transition vertex v_i to its final state without changing v_{i-1}.
- A left final-catalytic reaction (or left final), where vertex v_i reacts with vertex v_{i-1} in its final state to transition vertex v_i to its final state without changing v_{i-1}.
- A right reaction, where vertex v_i reacts with vertex v_{i+1} and both vertices reach their final states.
- A right catalytic reaction, where vertex v_i reacts with vertex v_{i+1} in its initial state to transition vertex v_i to its final state without changing v_{i+1}.
- A right final-catalytic reaction (or right final), where vertex v_i reacts with vertex v_{i+1} in its final state to transition vertex v_i to its final state without changing v_{i+1}.

Additionally, we also consider when a vertex is in its final state. An example system and sequence of reactions can be found in Figures 1 and 2.

We construct a $7 \times n$ table (Example in Table 2), where each row represents one of the possible reactions, including no reaction, and each column represents the starting configuration's vertices from left to right. For each entry in the table, we see if the reaction exists for that vertex and if the vertex reaches its final state. If both cases are satisfied, place a 1 in the corresponding row, otherwise, place a 0. After all cells are evaluated, we construct a directed graph with edges being directed from column i to column $i+1$ with the following properties for each row entry in column i:

- In final state: edge to every row in column $i+1$ with a 1 except left reaction.
- Left final: edge to every row in column $i+1$ with a 1 except left reaction.
- Left catalytic: edge to every row in column $i+1$ with a 1 except left reaction.
- Left reaction: edge to every row in column $i+1$ with a 1 except left reaction.
- Right final: edge to every row in column $i+1$ with a 1 except left reaction and left final.
- Right catalytic: edge to every row in column $i+1$ with a 1 except left reaction and left catalytic.
- Right reaction: edge only to the row corresponding to left reaction in column $i+1$ if there is a 1.

These edges ensure that no matter which reaction is chosen for a vertex represented by column i, the reaction chosen for the column $i+1$ vertex will be able to perform its reaction either before or after the previous reaction.

Once these edges are defined for every column, the problem is then finding a path from s to t, where s is a vertex that has directed edges to each entry in column 1 and t is a vertex that can be reached from each entry in column n (see Table 2 and Figure 3 for reference). Any path represents a set of rules that can be assigned an ordering to reconfigure all vertices to their final states.

Reaction Type				
In Final State	-	-	-	-
Left	-	1	-	-
Left Catalytic	-	-	-	1
Left Final	-	-	-	-
Right	1	-	-	-
Right Catalytic	-	-	-	-
Right Final	-	-	1	-

Table 2: Turning the example system from Figure 1 into a table of reactions.

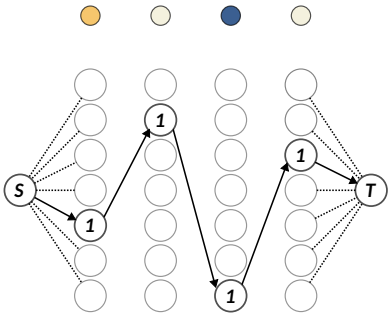


Figure 3: Table 2 as a graph.

Theorem 1 *Reconfiguration in 1-burnout for $1 \times n$ lines is solvable in $O(n + |R|)$ time.*

Proof. We provide proof by induction for the previously described algorithm that solves reconfiguration in $1 \times n$ surfaces. This proof guarantees that any solution from this algorithm constitutes a set of reactions that can be reordered to successfully reconfigure a given initial configuration to its final configuration.

Base case: $n = 2$. Let v_i be the leftmost vertex. Since this vertex does not have a neighbor to its left, there are only 4 reactions we need to consider for this vertex:

1. In final state: vertex v_{i+1} must be in its final state or a left catalytic or left final reaction.
2. Right catalytic: vertex v_{i+1} must be in its final state or a left final reaction.
3. Right final: vertex v_{i+1} must be in its final state or a left catalytic reaction.
4. Right reaction: vertex v_{i+1} must be a left reaction.

If two such reactions exist for each vertex, then a path exists from s to t visiting the vertices in the table that correspond to each reaction. Otherwise, no such path would exist.

Inductive step: let $n = k$. Assume that there is a set of k reactions for vertices $v_1, \ldots, v_k$ that can be reordered to transition all k vertices to their final states.

In order for the reaction chosen for vertex v_{k+1} to be valid, it must not interfere with the k^{th} reaction corresponding to vertex v_k. Consider two cases:

1. Vertex v_k is currently in its final state or reacts with its left neighbor v_{k-1}. Vertex v_{k+1} is never used, so as long as v_{k+1} does not perform a left reaction with v_k, it will not interfere with the k^{th} reaction.
2. Vertex v_k reacts with vertex v_{k+1}. Consider 3 possible reactions for v_k: (1) Right reaction: the only valid reaction for v_{k+1} is a left reaction, (2) Right catalytic: except left or left catalytic, all reactions are valid for v_{k+1}, and (3) Right final: except left or left final, all reactions are valid for v_{k+1}.

If we think of v_k as being column i and v_{k+1} as being column $i+1$, edges are defined from i to $i+1$ in a way that avoid these conflicting reactions. Any other reaction that is chosen for v_{i+1} can always perform its reaction before or after v_i performs its reaction. As a result, any path up to column $i+1$ would represent a set of reactions that can be reordered to transition these $k+1$ vertices to their final states.

Given the initial and final configurations, it takes $O(n)$ time to compare the states. Constructing the table takes $O(|R|)$ time. The path finding algorithm runs in $O(V+E) = O(n+E)$ time. However, the number of edges is a constant factor of the number of vertices, whereas $|R|$ might be exponential in n. Thus, the final runtime for the algorithm is $O(n+|R|)$. □

3.2 2-Burnout Linear Surfaces

Theorem 2 *Reconfiguration of a $1 \times n$ line for surface CRNs with 2-burnout is solvable in $O(n \cdot |S|^2 \cdot |R|^4)$ time.*

Proof. Since we are considering 2-burnout, every cell can only change species twice. This is a cell starting with the initial species, possibly changing to an intermediate species, then finally changing to the target species. It is then possible to track all the possible transitions of a cell in a polynomial sized table. We define the table D with each entry $D(x, s, r_1, r_2)$ being a Boolean indicating if the cells at indices $0, 1, \ldots, x$ can reach their target species using reactions r_1 and r_2 on x, and using s as intermediate species for cell x. (Note, r_1 and s may be null if the cell only reacts once to reach the target species.) The reactions are specific with which neighbor the cell reacts with, left or right. This results in $\mathcal{O}(n \cdot |S| \cdot |R|^2)$ cells of the table.

To compute each entry $D(x, s, r_1, r_2)$, we check if r_1 and r_2 are consistent with cell $x-1$. Meaning, if r_1 reacts with the left neighbor, some entry $D(x-1, s', r_1, r_3)$ or $D(x-1, s', r_3, r_1)$ for any s, r_3 must be true. If r_1 is a catalytic reaction, then the species in cell $x-1$ does not change and must be the initial species, intermediate species, or the target species. We must also be careful with the ordering of the reactions. If r_1 or r_2 reacts with the intermediate species s' of the $(x-1)$th cell, then r_1 must be the second reaction for $x-1$. The run time to compute each cell of the table is $\mathcal{O}(|S| \cdot |R|^2)$.

If any $D(n-1, s, r_1, r_2)$ is true, then the answer to reconfiguration is true. □

3.3 Constant Burnout

In this section, we consider the problem of reconfiguration for a surface CRN with n cells with at most k-burnouts on a $1 \times n$ board.

Theorem 3 *There is an $n^{1+k \log h}$-time algorithm for k-burnout degree-h 1D surface CRN reconfiguration, where each species is in at most h rules.*

Proof. We have a divide-and-conquer approach in our algorithm. A brute force method is used to enumerate all the possible transitions for the median position. The problem is split into two independent problems that can be solved independently.

Let p be the position of the median in a 1D surface CRN. We enumerate all the possible ways to burn out the position p at most k times. Since each species is in at most h rules, we have at most h^k combinations about the list of transitions involved by position p. Let $T(n)$ be the running time to solve the reconfiguration problem. We have the recursion $T(n) = h^k(2T(\frac{n}{2}))$. It brings a solution with $T(n) = h^{k \log n} \cdot n = n^{1+k \log h}$. □

4 Non-constant Burnout on a Line

Here, we show that reconfiguration with k-burnout, where k is part of the input, is NP-hard. Without burnout (no bound on state changes), reconfiguration of a $1 \times n$ line is PSPACE-complete [15], but even with a burnout k given in binary, the problem may not be in the class NP since $O(kn)$ possible reactions could occur, which is exponential in $\log k$. This motivates looking at bounds on state changes that are polynomial in n and further motivates the other algorithms in the paper.

Reduction. We reduce from Vertex Cover (VC) by enumerating all vertices and using them as states on a $1 \times n$ line. A state "walks" back and forth choosing a vertex to add to the cover and crossing off instances it finds. Given a graph $G = (V, E)$ where $V = \{1, 2, \ldots, n\}$ and an edge $e \in E$ is defined as $e = \{v_i, v_j\}$ for $v_i, v_j \in V$ and $i \neq j$. An edge is listed as two states: 34 meaning an edge between vertices v_3 and v_4. Between any two edges we include a spacing state $-$.

Create the line representing the graph with edges in any order: $BS_0 - e_1 - e_2 - \cdots - e_m - E$, where the B state indicates the beginning of the line, E is the end of the line, and S_0 is a special state indicating no vertices are in the vertex cover. Example: $BS_0 - 34 - 13 - 21 - 14 - E$.

Basically, each edge independently and nondeterministically picks the vertex to cover it with both possible

rules. Create rules for all $v_i, v_j \in V$ as $v_i + v_j \to v_i' + x$ and $v_i + v_j \to x + v_j'$ where x is an ignored state and the prime state is the chosen vertex for that edge. The spacing states ensure edges do not affect each other. Example: $3 + 4 \to 3' + x$ and $3 + 4 \to x + 4'$.

The S counting state sweeps back and forth k times to choose a vertex to add to the cover and ignores the other states. The S state takes the first picked vertex and removes all duplicates of it while remembering the count. There is a state S_{count}^{vertex} that exists for each vertex and count up to k. Thus, the rules $S_i + v_j' \to S_{i+1}^j + x$ are added for each vertex and count up to k. Example: $S_0 + 3' \to S_1^3 + x$ is used if v_3 is the first vertex added.

Once a vertex transitions to an S^i state, it ignores everything but v_i' states. Meaning it only swaps states, or "walks" in that direction. Thus, all rules $S^i + A \to A + S^i$ is added for any state X that is not v_i, B, or E. For v_i, $S_c^i + v_i' \to S_c^i + x$.

When a S_c^j vertex is next to the B or E states, it can transition to S_c. The rules $B + S_c^j \to B + S_c$ and $E + S_c^j \to E + S_c$ exist for all vertices v_j. This means we have removed all instances of the chosen vertex and can pick a new vertex for the cover.

This requires $O(kn)$ states to handle counting for each vertex. If k is odd, the final configuration, given a k VC exists, is $B - xx - xx - xx - \cdots - S_k E$. If k is even, then the final configuration is $BS_k - xx - xx - xx - \cdots - E$. S_k can not interact with anything. This requires $k + 1$ burnout.

Theorem 4 *Reconfiguration of a $1 \times n$ configuration in sCRNs with k-burnout is NP-hard, even when $k < n$, and NP-complete as long as k is polynomial in n.*

Proof. Given a VC with graph $G = (V, E)$ and $k \in \mathbb{N}$, we create a surface CRN system with configuration C and rules R as described above. We define the output configuration D based on the number of edges and parity of k as described. G has a VC of size k if and only if C can reach configuration D with burnout $k + 1$. Note that $k \leq n$ as input from VC, so the number of states and rules in the reduction is polynomial.

Given that the graph G has a k vertex cover, in the sCRN system, the only transitions possible at first are for each edge to pick a vertex to cover it. Then the counting state walks across, increases the count and selects the vertex from the first edge, and that state continues walking and removes any other instance of that vertex. In the best case, all locations but the first and last have changed twice. If this continues, and it always adds the correct vertices, then after k passes only x's are left. S_k does not interact with anything, so nothing else transitions. The k passes and the initial choice requires $k + 1$ burnout.

If the sCRN system ends in the output configuration with x's on every edge state, which can only occur if the k passes chose vertices that appeared in the other edges and were crossed out. Thus, every edge correctly chose the right vertex to cover it so that only k different vertices were used. □

5 Extension to 2D Graphs

As an extension to the 1D case, we now consider reconfiguration and 1-reconfiguration for 2D surfaces. In the case of reconfiguration, we study a restricted version of the problem where all reactions are non-catalytic.

Theorem 5 *Reconfiguration in 1-burnout for a planar graph $G = (V, E)$ is solvable in $O(|V|^{1.5} + |R|)$ time if every reaction is non-catalytic.*

Proof. Given a planar graph $G = (V, E)$, construct a subgraph G' from G such that there is an edge between pairs of vertices if there exists a non-catalytic reaction that transitions both vertices to their final states. Run maximum matching on G'. If all vertices are either matched or in their final state, then reconfiguration is possible. Otherwise, reconfiguration is not possible.

Since non-catalytic reactions transition both vertices to their final states, a vertex must be involved in at most one reaction. Edges represent these non-catalytic reactions between two vertices. As a result, limiting a vertex to one reaction is the equivalent of matching each vertex in G' to at most one other vertex it shares an edge with, which is a perfect matching problem. For planar graphs, this can be solved using a maximum matching algorithm. If any unmatched vertex is not in its final state, then reconfiguration is not possible because this vertex is unable to react.

Constructing G' takes $O(V + |R|)$ time. Running the maximum matching algorithm takes $O(V^{1.5})$ time. A last check of G' for any unmatched vertices that are not in their final state takes $O(V)$ time. Therefore, the runtime is $O(V^{1.5} + |R|)$. □

Corollary 6 *Reconfiguration in 1-burnout for general graphs is solvable in $O(V^4 + |R|)$ time if every reaction is non-catalytic, where V is the number of vertices.*

Proof. Proof follows from Theorem 5. Maximum matching on general graphs runs in $O(V^4)$ time. □

5.1 Arbitrary Graphs with 1-Burnout

We now consider surface CRNs that allow catalytic as well as non-catalytic rules. With this additional rule type, we prove the problem of reconfiguration is NP-complete on an arbitrary graph with 1-burnout.

Theorem 7 *Reconfiguration with 1-burnout of an arbitrary surface in surface CRNs is NP-complete.*

Proof. We reduce from the dominating set problem to sCRN reconfiguration with 1-burnout. Let $G = (V, E)$ be an arbitrary graph and k be an integer parameter.

We need to decide if graph G has a dominating set of size k. Note that a subset $U \subseteq V$ is a dominating set of G if each vertex $v \in V - U$ has $(u, v) \in E$ for some $u \in U$ (vertex u dominates v).

Let $v_1, \cdots, v_n$ be the n vertices of G. We design a surface CRN system. For each edge (v_i, v_j) in E, create two rules $v_i + v_j \to v_i + v_j'$ and $v_i + v_j \to v_i' + v_j$. We introduce k additional species $u_1, \cdots, u_k$. The target configuration is to let each v_i enter v_i' for $i = 1, \cdots, n$ and each u_t enter u_t'. We set up the rules $u_t + v_j \to u_t' + v_j'$ for all $t \leq k$ and all $j \leq n$.

If graph G has a dominating set of size k, the target configuration is reachable. Assume that $v_{i_1}, \cdots, v_{i_k}$ dominate all the vertices in the graph G. For each v_j with $j \in \{1, \cdots, n\} - \{i_1, \cdots, i_k\}$, it can be transformed into v_j' by a rule $v_{i_s} + v_j \to v_{i_s} + v_j'$. Each v_{i_s} can enter v_{i_s}' by a rule $u_s + v_{i_s} \to u_s' + v_{i_s}'$. Here, the burnout is 1. Similarly, if the target configuration is reachable, there is a dominating set of size k. If the target configuration is reachable, we have at most $v_{i_1}, \cdots, v_{i_h}$ with $(h \leq k)$ such that each v_{i_r} enters v_{i_r}' via the type of rule $u_t + v_{i_r} \to u_t' + v_{i_r}'$ as there is only one burnout for each v_i and u_j. Clearly, $v_{i_1}, \cdots, v_{i_h}$ dominate all the other vertices in the graph G.

This is a polynomial-time reduction and membership is known from [1]. □

5.2 1-Burnout 1-Reconfiguration

Theorem 8 *1-Reconfiguration in 1-burnout of a $w \times n$ rectangle for surface CRNs is solvable in $O\left(n \cdot (|S||R|)^{2w} \cdot f(w)\right)$ time.*

Proof. We use a dynamic programming approach similar to that in Theorem 2, defining a table D with Boolean entries $D(x, \vec{s}, \vec{r}, \pi)$, where x is a column index, $\vec{s} = [s_1, s_2, \ldots, s_w]$, $\vec{r} = [r_1, r_2, \ldots, r_w]$, and π a permutation of $[1, w]$. Each $s_y \in S$ is a potential final species of cell (x, y), which changes from its initial species into (x, y) due to reaction $r_y \in R$, and π gives the order in which the reactions occur. As before, r_y specifies which of its up-to-four neighboring cells participated in the reaction, and s_y and r_y may be null if the cell never changes species.

Since only one cell (x_t, y_t) of the target configuration is fixed, the top-level of the dynamic program will be column x_t, and it will symmetrically recurse outwards in both directions, with base-cases at both ends. So, for $x < x_t$ $D(x, \vec{s}, \vec{r}, \pi)$ is true if the cells in columns $0, 1, \ldots, x$ can reach a target configuration in which column x reaches species $\vec{s}$ using reactions $\vec{r}$ occurring in order π, for $x > x_t$ we consider columns $x, x+1, \ldots, n-1$ instead, and for $x = x_t$ we consider the entire surface.

To compute $D(x, \vec{s}, \vec{r}, \pi)$, say when $x < x_t$, we search for a smaller subproblem $D(x - 1, \vec{s}', \vec{r}', \pi')$ which has value true and $(\vec{r}', \vec{r})$ together are a chain of reactions that actually transform columns $x-1$ and x into species $(\vec{s}', \vec{s})$ from their initial species, given that they must occur in relative orders π' and π. Specifically, we can search each possible interleaving of $\pi(\vec{r})$ and $\pi'(\vec{r}')$, simulate the reactions in that order, and verify that the reactions within these two columns can actually be performed and do result in $(\vec{s}', \vec{s})$. Notably, for reactions between columns $x - 2$ and $x - 1$, we do not need to validate the species in column $x-2$ because the smaller subproblem already performed that validation, and for reactions between columns x and $x + 1$, the species in column $x+1$ are assumed to be validated later in a larger subproblem. For $x > x_t$, the recursion is symmetric.

For top-level subproblems $D(x_t, \vec{s}, \vec{r}, \pi)$, we only consider $\vec{s}$ that include the fixed target species s_{y_t}, and we search for both $D(x_t - 1, \vec{s}', \vec{r}', \pi')$ and $D(x_t + 1, \vec{s}'', \vec{r}'', \pi'')$ and validate between all three columns $x_t - 1, x_t, x_t + 1$ in a similar manner. If any $D(x_t, \vec{s}, \vec{r}, \pi)$ is true, then the answer to 1-reconfiguration is true.

The size of D is $O\left(n \cdot |S|^w \cdot (4|R|)^w \cdot w!\right)$. Computing each entry involves checking $O\left(|S|^w \cdot (4|R|)^w \cdot w!\right)$ subproblems, and each check considers $\binom{2w}{w}$ interleavings of orderings and runs an simulation taking $O(w)$ time. Combined, the total time is $O\left(n \cdot (|S||R|)^{2w} \cdot f(w)\right)$ for a function f only depending on w. Therefore, for constant w, this is polynomial time. □

6 Conclusion

In this paper, we have shown that the reconfiguration problem on $1 \times n$ surface CRNs with k-burnout is in P when $k = 1$ or $k = 2$. To show this, we have given algorithms that output a sequence of reactions to achieve the given configuration. Further, we show that for any $k = \mathcal{O}(1)$, there exists an algorithm that has a polynomial runtime in k. To conclude our investigation of 1-Dimensional surface CRNs, we prove that when the burnout number, k, is part of the input (in unary), the problem of reconfiguration is NP-complete.

Following by exploring 2-Dimensional surface CRNs and showing that a restricted case of 1-burnout reconfiguration can be seen as perfect matching, showing this case of the problem to still be in P. Finishing with a proof that the problem of 1-Reconfiguration in 1-burnout can be solved in polynomial time on a $w \times n$ rectangle when w is constant.

Some of the open questions are then:

- What is the lower bound for a given k-burnout?
- In a rectangle/grid graph, what is the lower/upper bound for k-burnout?
- Most of our complexity is in terms of the size of the surface. Are there interesting results looking at the complexity of other aspects of an sCRN such as states, rules, and burnout?
- We have a direct NP-complete reduction but does there exist an L-reduction for some inapproximability result?

References

[1] R. M. Alaniz, J. Brunner, M. Coulombe, E. D. Demaine, Y. Diomidov, T. Gomez, E. Grizzell, R. Knobel, J. Lynch, A. Rodriguez, R. Schweller, and T. Wylie. Complexity of reconfiguration in surface chemical reaction networks. In *Proc. of the 29th International Conference on DNA Computing and Molecular Programming*, DNA'23, 2023. To appear.

[2] T. Brailovskaya, G. Gowri, S. Yu, and E. Winfree. Reversible computation using swap reactions on a surface. In *Proc. of the International Conference on DNA Computing and Molecular Programming*, DNA'19, pages 174–196. Springer, 2019.

[3] C. Chalk, A. Luchsinger, E. Martinez, R. Schweller, A. Winslow, and T. Wylie. Freezing simulates non-freezing tile automata. In *DNA Computing and Molecular Programming: 24th International Conference, DNA 24, Jinan, China, October 8–12, 2018, Proceedings 24*, pages 155–172. Springer, 2018.

[4] G. Chatterjee, N. Dalchau, R. A. Muscat, A. Phillips, and G. Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature nanotechnology*, 12(9):920–927, 2017.

[5] H.-L. Chen, D. Doty, and D. Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13:517–534, 2014.

[6] C. T. Chou. Chemical reaction networks for computing logarithm. *Synthetic Biology*, 2(1):ysx002, Jan. 2017.

[7] S. Clamons, L. Qian, and E. Winfree. Programming and simulating chemical reaction networks on a surface. *Journal of the Royal Society Interface*, 17(166):20190790, 2020.

[8] M. Cook et al. Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.

[9] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

[10] F. Dannenberg, M. Kwiatkowska, C. Thachuk, and A. J. Turberfield. DNA walker circuits: computational potential, design, and verification. *Natural Computing*, 14(2):195–211, 2015.

[11] E. Goles, D. Maldonado, P. Montealegre, and M. Ríos-Wilson. On the complexity of asynchronous freezing cellular automata. *Information and Computation*, 281:104764, 2021.

[12] E. Goles, N. Ollinger, and G. Theyssier. Introducing freezing cellular automata. In *Cellular Automata and Discrete Complex Systems, 21st International Workshop (AUTOMATA 2015)*, volume 24, pages 65–73, 2015.

[13] R. A. Muscat, K. Strauss, L. Ceze, and G. Seelig. DNA-based molecular architecture with spatially localized components. *ACM SIGARCH Computer Architecture News*, 41(3):177–188, 2013.

[14] T. Neary and D. Woods. P-completeness of cellular automaton rule 110. In *Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I 33*, pages 132–143. Springer, 2006.

[15] L. Qian and E. Winfree. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In *DNA Computing and Molecular Programming: 20th International Conference, DNA 20, Kyoto, Japan, September 22-26, 2014. Proceedings*, volume 8727, page 114. Springer, 2014.

[16] D. Soloveichik, M. Cook, E. Winfree, and J. Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7:615–633, 2008.

[17] G. Theyssier and N. Ollinger. Freezing, bounded-change and convergent cellular automata. *Discrete Mathematics & Theoretical Computer Science*, 24, 2022.

[18] A. J. Thubagere, W. Li, R. F. Johnson, Z. Chen, S. Doroudi, Y. L. Lee, G. Izatt, S. Wittman, N. Srinivas, D. Woods, et al. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017.

REFERENCES

[1] R. M. ALANIZ, D. CABALLERO, S. C. CIRLOS, T. GOMEZ, E. GRIZZELL, A. RODRIGUEZ, R. SCHWELLER, A. TENORIO, AND T. WYLIE, *Building squares with optimal state complexity in restricted active self-assembly*, Journal of Computer and System Sciences, 138 (2023), p. 103462.

[2] E. R. BANKS, *Universality in cellular automata*, in 11th Annual Symposium on Switching and Automata Theory (swat), 1970, pp. 194–215.

[3] R. BRICEÑO AND I. RAPAPORT, *Communication complexity meets cellular automata: Necessary conditions for intrinsic universality*, Natural Computing, 20 (2021), pp. 307–320.

[4] A. A. CANTU, A. LUCHSINGER, R. SCHWELLER, AND T. WYLIE, *Signal Passing Self-Assembly Simulates Tile Automata*, in 31st International Symposium on Algorithms and Computation (ISAAC 2020), vol. 181, 2020, pp. 53:1–53:17.

[5] C. CHALK, A. LUCHSINGER, E. MARTINEZ, R. SCHWELLER, A. WINSLOW, AND T. WYLIE, *Freezing Simulates Non-freezing Tile Automata*, in DNA Computing and Molecular Programming, 2018, pp. 155–172.

[6] E. D. DEMAINE, M. L. DEMAINE, S. P. FEKETE, M. J. PATITZ, R. T. SCHWELLER, A. WINSLOW, AND D. WOODS, *One Tile to Rule Them All: Simulating Any Turing Machine, Tile Assembly System, or Tiling System with a Single Puzzle Piece*, ArXiv e-Prints, (2012).

[7] E. D. DEMAINE, M. J. PATITZ, T. A. ROGERS, R. T. SCHWELLER, S. M. SUMMERS, AND D. WOODS, *The Two-Handed Tile Assembly Model is not Intrinsically Universal*, Algorithmica, 74 (2016), pp. 812–850.

[8] D. DOTY, J. H. LUTZ, M. J. PATITZ, R. T. SCHWELLER, S. M. SUMMERS, AND D. WOODS, *The Tile Assembly Model is Intrinsically Universal*, in 53rd Annual Symposium on Foundations of Computer Science, 2012, pp. 302–310.

[9] D. DOTY, J. H. LUTZ, M. J. PATITZ, S. M. SUMMERS, AND D. WOODS, *Intrinsic Universality in Self-Assembly*, in 27th International Symposium on Theoretical Aspects of Computer Science (2010), 2010.

[10] B. DURAND AND Z. RÓKA, *The game of life: universality revisited*, in Cellular Automata: a Parallel Model, 1999, pp. 51–74.

[11] J. O. DURAND-LOSE, *Intrinsic universality of a 1-dimensional reversible cellular automaton*, in 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS), 1997, pp. 439–450.

[12] E. GOLES, P. MEUNIER, I. RAPAPORT, AND G. THEYSSIER, *Communication complexity and intrinsic universality in cellular automata*, Theoretical Computer Science, 412 (2011), pp. 2–21.

[13] D. HADER, A. KOCH, M. J. PATITZ, AND M. SHARP, *The Impacts of Dimensionality, Diffusion, and Directedness on Intrinsic Universality in the abstract Tile Assembly Model*, in Symposium on Discrete Algorithms (SODA), 2019, pp. 2607–2624.

[14] J. HENDRICKS AND M. J. PATITZ, *On the Equivalence of Cellular Automata and the Tile Assembly Model*, Electronic Proceedings in Theoretical Computer Science, 128 (2013), pp. 167–189.

[15] J. IIRGEN ALBERT AND K. CULIK II, *A simple universal cellular automaton and its one-way and totalistic version*, Complex Systems, 1 (1987), pp. 1–16.

[16] G. LAFITTE AND M. WEISS, *Universal tilings*, in 24th Annual Symposium on Theoretical Aspects of Computer Science (STACS), vol. 4393, 2007, pp. 367–380.

[17] ——, *An Almost Totally Universal Tile Set*, in Theory and Applications of Models of Computation, vol. 5532, 2009, pp. 271–280.

[18] ——, *Tilings: simulation and universality*, Mathematical Structures in Computer Science, 20 (2010), pp. 813–850.

[19] M. MARGENSTERN, *An algorithm for building intrinsically universal cellular automata in hyperbolic spaces*, in International Conference on Foundations of Computer Science (FCS), 2006, pp. 3–9.

[20] P.- MEUNIER, M. J. PATITZ, S. M. SUMMERS, G. THEYSSIER, A. WINSLOW, AND D. WOODS, *Intrinsic universality in tile self-assembly requires cooperation*, in 25th Annual ACM-SIAM Symposium on Discrete Algorithms, 2014, pp. 752–771.

[21] J. V. NEUMANN, *Theory of self-reproducing automata*, Urbana, University of Illinois Press, 1966.

[22] N. OLLINGER, *The Quest for Small Universal Cellular Automata*, in Automata, Languages and Programming, vol. 2380, 2002, pp. 318–329.

[23] ——, *The Intrinsic Universality Problem of One-Dimensional Cellular Automata*, in STACS 2003, H. Alt and M. Habib, eds., Lecture Notes in Computer Science, Berlin, Heidelberg, 2003, Springer, pp. 632–641.

[24] ——, *Universalities in cellular automata; a (short) survey*, Journees Automates Cellulaires, (2008).

[25] N. OLLINGER AND G. RICHARD, *Four states are enough!*, Theoretical Computer Science, 412 (2011), pp. 22–32.

[26] P. PETERSEN, G. TIKHOMIROV, AND L. QIAN, *Information-based autonomous reconfiguration in systems of interacting DNA nanostructures*, Nature communications, 9 (2018), p. 5362.

[27] N. SARRAF, K. R. RODRIGUEZ, AND L. QIAN, *Modular reconfiguration of DNA origami assemblies using tile displacement*, Science Robotics, 8 (2023), p. eadf1511.

[28] E. WINFREE, *Algorithmic self-assembly of DNA*, California Institute of Technology, 1998.

[29] D. WOODS, D. DOTY, C. MYHRVOLD, J. HUI, F. ZHOU, P. YIN, AND E. WINFREE, *Diverse and robust molecular algorithms using reprogrammable DNA self-assembly*, Nature, 567 (2019), pp. 366–372.

[30] T. WORSCH, *Towards intrinsically universal asynchronous CA*, Natural Computing, 12 (2013), pp. 539–550.

VITA

Elise Constance Grizzell grew up in Salt Lake City, Utah and moved to the Rio Grande Valley in 2016. While an undergraduate she joined the Algorithmic Self-Assembly research group where she along with other students published many papers working under Dr. Bin Fu, Dr. Robert Schweller and Dr. Tim Wylie. After completing her undergraduate in Computer Science at UTRGV in 2019, Elise continued her education at UTRGV where she received the Graduate Assistance in Areas of National Need Fellowship to support her as a graduate student. She received her Master's degree in Computer Science in 2024. Elise can be reached at elisegrizzell@gmail.com.