University of Arkansas, Fayetteville
ScholarWorks@UARK

Computer Science and Computer Engineering Undergraduate Honors Theses | Computer Science and Computer Engineering

5-2018

The 3D abstract Tile Assembly Model is Intrinsically Universal

Aaron Koch

Daniel Hader

Matthew J. Patitz

Follow this and additional works at: http://scholarworks.uark.edu/csceuht

Part of the Computational Engineering Commons, and the Other Computer Engineering Commons

Recommended Citation

Koch, Aaron; Hader, Daniel; and Patitz, Matthew J., "The 3D abstract Tile Assembly Model is Intrinsically Universal" (2018). *Computer Science and Computer Engineering Undergraduate Honors Theses*. 59.
http://scholarworks.uark.edu/csceuht/59

This Thesis is brought to you for free and open access by the Computer Science and Computer Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Computer Science and Computer Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu, ccmiddle@uark.edu.

The 3D abstract Tile Assembly Model is Intrinsically Universal

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR
April, 2018

by

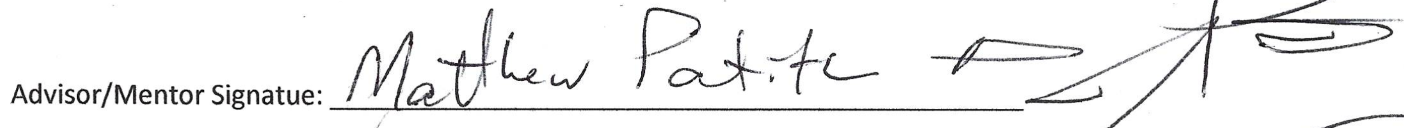
Aaron Koch

Title: The 3D aTAM is IU

Author: Aaron Koch (with contribution from Daniel Hader)

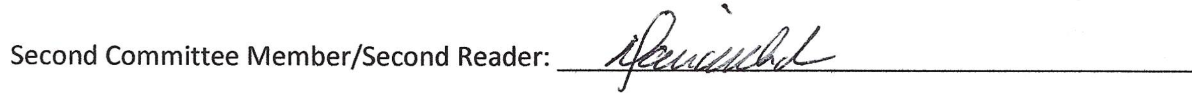
Graduation Month and Year: May 2018

Advisor/Mentor Signatue: ______________________________



First Committee Member/First Reader: ______________________________

Second Committee Member/Second Reader: ______________________________



Third Committee Member: ______________________________

This is for your additional files only.

The 3D abstract Tile Assembly Model is Intrinsically Universal

Daniel Hader * Aaron Koch † Matthew J. Patitz ‡

Abstract

In this paper, we prove that the three-dimensional abstract Tile Assembly Model (3DaTAM) is intrinsically universal. This means that there is a universal tile set in the 3DaTAM which can be used to simulate any 3DaTAM system. This result adds to a body of work on the intrinsic universality of models of self-assembly, and is specifically motivated by a result in FOCS 2016 showing that any intrinsically universal tile set for the 2DaTAM requires nondeterminism (i.e. undirectedness) even when simulating directed systems. To prove our result we have not only designed, but also fully implemented what we believe to be the first intrinsically universal tile set which has been implemented and simulated in any tile assembly model, and have made it and a simulator which can display it freely available.

Table of Contents

1. Introduction
   (a) Motivation
   (b) Background
   (c) Explicit Tileset
   (d) Software
2. Definitions
   (a) aTAM
   (b) Simulation
   (c) Intrinsic Universality
3. Construction
   (a) Orientation
   (b) Movement Patterns
   (c) Datapath
   (d) Adder Array
   (e) Bracket
   (f) Layout
   (g) Correctness

*Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA dhader@email.uark.edu. This author's research was supported in part by National Science Foundation Grant CCF-1422152.

†Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA aekoch@email.uark.edu. This author's research was supported in part by National Science Foundation Grant CCF-1422152.

‡Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA patitz@uark.edu This author's research was supported in part by National Science Foundation Grant CCF-1422152 and CAREER-1553166.

1 Introduction

Self-Assembly Self-Assembly is a process which generates complexity out of randomness. From the self-assembly of a snowflake via interactions between water molecules, to the assembly of a spiral galaxy from the interactions between stars, self-assembly processes are responsible for much of the order we observe in the universe. There are currently both practical and theoretical explorations of self-assembling systems. This work is part of the theoretical exploration, but is motivated by the increasingly precise control possible in the lab. This work moves forward a line of research exploring the theoretical capabilities and limitations of different models of self-assembly. Comparing the capabilities and limitations of different models can help guide and motivate the practical study of self-assembling systems.

Previous IU Results This work builds on previous explorations of intrinsic universality (IU) in other models of self-assembly, including the 2-Handed Assembly Model (2HAM) and the 2D version of the abstract Tile Assembly Model (aTAM). There are a mix of positive and negative IU results under different models and parameters. The 2HAM has been shown not to be IU across temperatures (where temperature is a parameter specifying the minimum binding threshold required for components to stably combine), but systems at each specific temperature are IU [1]. The aTAM, both 2D and 3D, has been shown to not be IU at temperature 1 [4]. This result directly effects our construction, requiring that we make our IU tileset at least temperature 2.

Two results in the 2D aTAM are direct precursors to this work. One early exploration into IU in self-assembly models found that the entire 2D aTAM is IU [2]. A later result was able to show that the subset of 2D aTAM systems that are directed are not IU [3]. A primary motivation for this work was to explore whether the 3D aTAM exhibited similar or different properties. If the 3D aTAM is IU both in totality as well as in just the subset of directed systems, that provides evidence that the planarity of the 2D aTAM is involved in the limitations of the power of the 2D aTAM.

Explicit IU Tileset Due to the complexity of other IU results, as far as we know no IU tileset has ever been fully defined to the individual tile level. Our tileset is the first to be explicitly generated and tested in full. We used a series of Python scripts to design, generate, and test each component of our construction. We combined the tilesets for each component into a master tileset with approximately 100,000 tile types. We explicitly defined the algorithm required to correctly seed our IU tileset for an arbitrary 3D aTAM system. Therefore for very simple assemblies we can, given enough time and computational resources, simulate them using our IU tileset to fully grow from seed to termination. This is a milestone in the exploration of self-assembly models.

Simulation Software This construction put significant pressure on the standard software used to simulate the aTAM. The previous software, called ISU TAS [5], could barely handle 1 million tiles in an assembly. Some of our single components require more than 1 million tiles, and to fully simulate a single supercube we need software that can handle tens of millions of tiles. PyTAS is an updated version of the ISU TAS software that was developed concurrently with this construction to help handle these requirements. This construction pushes the frontier of the scale of self-assembly simulation.

Simulation We simulated two example tilesets. The first is Grow, possibly the simplest tileset possible containing a single tile with a single strength 2 glue on all of its faces. Grow is an infinite directed system that fills space with a single tile type. The second example that we use is a very simple 4 tile system we call CO-OP. CO-OP uses a single seed that uses strength 2 glues to in two different orthogonal directions to grow two tiles which cooperate to place a single tile. Even this four tile assembly requires more than 1 billion tiles from our IU tileset to fully simulate, due to the 16 extra frontier locations that are generated as direct neighbors of the differentiated locations.

2 Preliminaries

In this section, we present definitions for the models and concepts used throughout the paper.

2.1 Formal description of the 3D abstract Tile Assembly Model

This section gives a formal definition of the 3-dimensional version of the abstract Tile Assembly Model (3DaTAM), which is the natural extension of the (2-dimensional) abstract Tile Assembly Model (aTAM) [7]. For readers unfamiliar with the aTAM, [6] gives an excellent introduction to the model, here we provide a brief introduction.

Fix an alphabet Σ. Σ^* is the set of finite strings over Σ. $\mathbb{Z}$, $\mathbb{Z}^+$, and $\mathbb{N}$ denote the set of integers, positive integers, and nonnegative integers, respectively. Given $V \subseteq \mathbb{Z}^3$, the *full grid graph* of V is the undirected graph $G^{\mathrm{f}}_V = (V, E)$, and for all $\vec{v_1} = (x_1, y_1, z_1), \vec{v_2} = (x_2, y_2, z_2) \in V$, $\{\vec{v_1}, \vec{v_2}\} \in E \iff \|\vec{v_1} - \vec{v_2}\| = 1$; i.e., if and only if $\vec{v_1}$ and $\vec{v_2}$ are adjacent in the 3-dimensional integer Cartesian space.

A *tile type* is a tuple $t \in (\Sigma^* \times \mathbb{N})^6$; e.g., a unit cube with six sides listed in some standardized order, each side having a *glue* $g \in \Sigma^* \times \mathbb{N}$ consisting of a finite string *label* and nonnegative integer *strength*. We assume a finite set of tile types, but an infinite number of copies of each tile type, each copy referred to as a *tile*. A tile set T is a set of tile types.

A *configuration* is a (possibly empty) arrangement of tiles on the integer lattice $\mathbb{Z}^3$, i.e., a partial function $\alpha : \mathbb{Z}^3 \dashrightarrow T$. Two adjacent tiles in a configuration *interact*, or are *attached*, if the glues on their abutting sides are equal (in both label and strength) and have positive strength. Each configuration α induces a *binding graph* G^{b}_α, a grid graph whose vertices are positions occupied by tiles, according to α, with an edge between two vertices if the tiles at those vertices interact. An *assembly* is a connected non-empty configuration, i.e., a partial function $\alpha : \mathbb{Z}^3 \dashrightarrow T$ such that $G^{\mathrm{f}}_{\mathrm{dom}\ \alpha}$ is connected and $\mathrm{dom}\ \alpha \neq \varnothing$. The *shape* $S_\alpha \subseteq \mathbb{Z}^3$ of α is $\mathrm{dom}\ \alpha$.

Given $\tau \in \mathbb{Z}^+$, α is *τ-stable* if every cut of G^{b}_α has weight at least τ, where the weight of an edge is the strength of the glue it represents. When τ is clear from context, we say α is *stable*. Given two assemblies α, β, we say α is a *subassembly* of β, and we write $\alpha \sqsubseteq \beta$, if $S_\alpha \subseteq S_\beta$ and, for all points $p \in S_\alpha$, $\alpha(p) = \beta(p)$.

A *tile assembly system* (TAS) is a triple $\mathcal{T} = (T, \sigma, \tau)$, where T is a finite set of tile types, $\sigma : \mathbb{Z}^3 \dashrightarrow T$ is the finite, τ-stable, *seed assembly*, and $\tau \in \mathbb{Z}^+$ is the *temperature*. Given two τ-stable assemblies α, β, we write $\alpha \to_1^{\mathcal{T}} \beta$ if $\alpha \sqsubseteq \beta$ and $|S_\beta \setminus S_\alpha| = 1$. In this case we say α *produces β in one step*. If $\alpha \to_1^{\mathcal{T}} \beta$, $S_\beta \setminus S_\alpha = \{p\}$, and $t = \beta(p)$, we write $\beta = \alpha + (p \mapsto t)$. The *frontier* of α is the set $\partial^{\mathcal{T}} \alpha = \bigcup_{\alpha \to_1^{\mathcal{T}} \beta} S_\beta \setminus S_\alpha$, the set of empty locations at which a tile could stably attach to α. The *t-frontier* $\partial^{\mathcal{T}}_t \alpha \subseteq \partial^{\mathcal{T}} \alpha$ of α is the set $\left\{\ p \in \partial^{\mathcal{T}} \alpha \ \middle|\ \alpha \to_1^{\mathcal{T}} \beta \text{ and } \beta(p) = t\ \right\}$.

Let $\mathcal{A}^T$ denote the set of all assemblies of tiles from T, and let $\mathcal{A}^T_{<\infty}$ denote the set of finite assemblies of tiles from T. A sequence of $k \in \mathbb{Z}^+ \cup \{\infty\}$ assemblies $\alpha_0, \alpha_1, \ldots$ over $\mathcal{A}^T$ is an *assembly sequence* if, for all $1 \leq i < k$, $\alpha_{i-1} \to_1^{\mathcal{T}} \alpha_i$. The *result* of an assembly sequence is the unique limiting assembly (for a finite sequence, this is the final assembly in the sequence).

We write $\alpha \to^{\mathcal{T}} \beta$, and we say α *produces* β (in 0 or more steps) if there is an assembly sequence $\alpha_0, \alpha_1, \ldots, \alpha_{k-1}$ of length $k = |S_\beta \setminus S_\alpha| + 1$ such that (1) $\alpha = \alpha_0$, (2) $S_\beta = \bigcup_{0 \leq i < k} S_{\alpha_i}$, and (3) for all $0 \leq i < k$, $\alpha_i \sqsubseteq \beta$. If k is finite then it is routine to verify that $\beta = \alpha_{k-1}$. We say α is *producible* if $\sigma \to^{\mathcal{T}} \alpha$, and we write $\mathcal{A}[\mathcal{T}]$ to denote the set of producible assemblies in $\mathcal{T}$. The relation $\to^{\mathcal{T}}$ is a partial order on $\mathcal{A}[\mathcal{T}]$. An assembly α is *terminal* if α is τ-stable and $\partial^{\mathcal{T}} \alpha = \varnothing$. We write $\mathcal{A}_\Box[\mathcal{T}] \subseteq \mathcal{A}[\mathcal{T}]$ to denote the set of producible, terminal assemblies. If $|\mathcal{A}_\Box[\mathcal{T}]| = 1$ then $\mathcal{T}$ is said to be *directed*. When $\mathcal{T}$ is clear from context, we may omit $\mathcal{T}$ from the notation above and instead write $\to_1$, $\to$, $\partial \alpha$, etc.

2.2 Simulation definition

To state our main result, we must formally define what it means for one TAS to "simulate" another. Our definitions come from [4]. Intuitively, simulation of a system $\mathcal{T}$ by another system $\mathcal{S}$ is done by utilizing some scale factor $m \in \mathbb{Z}^+$ such that $m \times m$ squares of tiles in $\mathcal{S}$ represent individual tiles in $\mathcal{T}$, and there is a "representation function" which is able to interpret the assemblies of $\mathcal{S}$ as assemblies in $\mathcal{T}$.

From this point on, let T be a tile set and let $m \in \mathbb{Z}^+$. An *m-block supertile* over T is a partial function $\alpha : \mathbb{Z}_m^3 \dashrightarrow T$, where $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. Let B_m^T be the set of all m-block supertiles over T. The m-block with no domain is said to be *empty*. For a general assembly $\alpha : \mathbb{Z}^3 \dashrightarrow T$ and $(x', y', z') \in \mathbb{Z}^3$, define $\alpha^m_{(x',y',z')}$ to be the m-block supertile defined by $\alpha^m_{(x',y',z')}(i_x, i_y, i_z) = \alpha(mx' + i_x, my' + i_y, mz' + i_z)$ for $0 \leq i_x, i_y, i_z < m$. For some tile set S, a partial function $R : B_m^S \dashrightarrow T$ is said to be a *valid m-block supertile representation* from S to T if for any $\alpha, \beta \in B_m^S$ such that $\alpha \sqsubseteq \beta$ and $\alpha \in \text{dom } R$, then $R(\alpha) = R(\beta)$.

For a given valid m-block supertile representation function R from tile set S to tile set T, define the *assembly representation function*[1] $R^* : \mathcal{A}^S \to \mathcal{A}^T$ such that $R^*(\alpha') = \alpha$ if and only if $\alpha(x, y, z) = R\left(\alpha'^m_{(x,y,z)}\right)$ for all $(x, y, z) \in \mathbb{Z}^3$. For an assembly $\alpha' \in \mathcal{A}^S$ such that $R(\alpha') = \alpha$, α' is said to map *cleanly* to $\alpha \in \mathcal{A}^T$ under R^* if for all non empty blocks $\alpha'^m_{(x,y,z)}$, $(x, y, z) + (u_x, u_y, u_z) \in \text{dom } (\alpha)$ for some $(u_x, u_y, u_z) \in U_3$ such that $u_x^2 + u_y^2 + u_z^2 \leq 1$, or if α' has at most one non-empty m-block $\alpha^m_{0,0}$. In other words, α' may have tiles on supertile blocks representing empty space in α, but only if that position is adjacent to a tile in α. We call such growth "around the edges" of α' *fuzz* and thus restrict it to be adjacent to only valid supertiles, but not diagonally adjacent (i.e. we do not permit *diagonal fuzz*).

In the following definitions, let $\mathcal{T} = (T, \sigma_T, \tau_T)$ be a TAS, let $\mathcal{S} = (S, \sigma_S, \tau_S)$ be a TAS, and let R be an m-block representation function $R : B_m^S \to T$.

Definition 2.1. We say that $\mathcal{S}$ and $\mathcal{T}$ have *equivalent productions* (under R), and we write $\mathcal{S} \Leftrightarrow \mathcal{T}$ if the following conditions hold:

1. $\{R^*(\alpha') | \alpha' \in \mathcal{A}[\mathcal{S}]\} = \mathcal{A}[\mathcal{T}]$.
2. $\{R^*(\alpha') | \alpha' \in \mathcal{A}_\Box[\mathcal{S}]\} = \mathcal{A}_\Box[\mathcal{T}]$.
3. For all $\alpha' \in \mathcal{A}[\mathcal{S}]$, α' maps cleanly to $R^*(\alpha')$.

Definition 2.2. We say that $\mathcal{T}$ *follows* $\mathcal{S}$ (under R), and we write $\mathcal{T} \dashv_R \mathcal{S}$ if $\alpha' \to^{\mathcal{S}} \beta'$, for some $\alpha', \beta' \in \mathcal{A}[\mathcal{S}]$, implies that $R^*(\alpha') \to^{\mathcal{T}} R^*(\beta')$.

Definition 2.3. We say that $\mathcal{S}$ *models* $\mathcal{T}$ (under R), and we write $\mathcal{S} \models_R \mathcal{T}$, if for every $\alpha \in \mathcal{A}[\mathcal{T}]$, there exists $\Pi \subset \mathcal{A}[\mathcal{S}]$ where $R^*(\alpha') = \alpha$ for all $\alpha' \in \Pi$, such that, for every $\beta \in \mathcal{A}[\mathcal{T}]$ where $\alpha \to^{\mathcal{T}} \beta$, (1) for every $\alpha' \in \Pi$ there exists $\beta' \in \mathcal{A}[\mathcal{S}]$ where $R^*(\beta') = \beta$ and $\alpha' \to^{\mathcal{S}} \beta'$, and (2) for every $\alpha'' \in \mathcal{A}[\mathcal{S}]$ where $\alpha'' \to^{\mathcal{S}} \beta'$, $\beta' \in \mathcal{A}[\mathcal{S}]$, $R^*(\alpha'') = \alpha$, and $R^*(\beta') = \beta$, there exists $\alpha' \in \Pi$ such that $\alpha' \to^{\mathcal{S}} \alpha''$.

The previous definition essentially specifies that every time $\mathcal{S}$ simulates an assembly $\alpha \in \mathcal{A}[\mathcal{T}]$, there must be at least one valid growth path in $\mathcal{S}$ for each of the possible next steps that $\mathcal{T}$ could make from α which results in an assembly in $\mathcal{S}$ that maps to that next step.

Definition 2.4. We say that $\mathcal{S}$ *simulates* $\mathcal{T}$ (under R) if $\mathcal{S} \Leftrightarrow_R \mathcal{T}$ (equivalent productions), $\mathcal{T} \dashv_R \mathcal{S}$ and $\mathcal{S} \models_R \mathcal{T}$ (equivalent dynamics).

2.3 Intrinsic Universality

Now that we have a formal definition of what it means for one tile system to simulate another, we can proceed to formally define the concept of intrinsic universality, i.e., when there is one general-purpose tile set that can be appropriately programmed to simulate any other tile system from a specified class of tile systems.

Let REPR denote the set of all supertile representation functions (i.e., m-block supertile representation functions for some $m \in \mathbb{Z}^+$). Define $\mathfrak{C}$ to be a class of tile assembly systems, and let U be a tile set. Note that each element of $\mathfrak{C}$, REPR, and $\mathcal{A}^U_{<\infty}$ is a finite object, hence encoding and decoding of simulated and simulator assemblies can be defined to be computable via standard models such as Turing machines and Boolean circuits.

[1] Note that R^* is a total function since every assembly of S represents *some* assembly of T; the functions R and α are partial to allow undefined points to represent empty space.

Definition 2.5. We say U is *intrinsically universal* for $\mathfrak{C}$ *at temperature* $\tau' \in \mathbb{Z}^+$ if there are computable functions $\mathcal{R} : \mathfrak{C} \to \mathsf{REPR}$ and $S : \mathfrak{C} \to \mathcal{A}^U_{<\infty}$ such that, for each $\mathcal{T} = (T, \sigma, \tau) \in \mathfrak{C}$, there is a constant $m \in \mathbb{N}$ such that, letting $R = \mathcal{R}(\mathcal{T})$, $\sigma_\mathcal{T} = S(\mathcal{T})$, and $\mathcal{U}_\mathcal{T} = (U, \sigma_\mathcal{T}, \tau')$, $\mathcal{U}_\mathcal{T}$ simulates $\mathcal{T}$ at scale m and using supertile representation function R.

That is, $\mathcal{R}(\mathcal{T})$ outputs a representation function that interprets assemblies of $\mathcal{U}_\mathcal{T}$ as assemblies of $\mathcal{T}$, and $S(\mathcal{T})$ outputs the seed assembly used to program tiles from U to represent the seed assembly of $\mathcal{T}$.

Definition 2.6. We say that U is *intrinsically universal* for $\mathfrak{C}$ if it is intrinsically universal for $\mathfrak{C}$ at some temperature $\tau' \in Z^+$.

Definition 2.7. We say that $\mathfrak{C}$ is intrinsically universal if there exists some U such that U is instrinsically universal for $\mathfrak{C}$.

3 3DaTAM Intrinsic Universality Construction

Theorem 3.1. The 3DaTAM is intrinsically universal.

To prove Theorem 3.1, we must show that there exist functions $\mathcal{R}$ and S (of Definition 2.5) and some tile set U such that, for each $\mathcal{T} = (T, \sigma, \tau)$ which is a TAS in the 3DaTAM, there is a constant $m \in \mathbb{N}$ such that, letting $R = \mathcal{R}(\mathcal{T})$, $\sigma_\mathcal{T} = S(\mathcal{T})$, and $\mathcal{U}_\mathcal{T} = (U, \sigma_\mathcal{T}, \tau')$, $\mathcal{U}_\mathcal{T}$ simulates $\mathcal{T}$ at scale m and using supertile representation function R. To do so, we will set $\tau' = 2$ (i.e. the simulations by U will all be at temperature $= 2$), and we will explicitly define U and give the algorithms which implement $\mathcal{R}$ and S.

We first provide a high-level overview of the main components of the construction and the way they are combined to create $\mathcal{U}_\mathcal{T}$ which simulates arbitrary 3D aTAM system $\mathcal{T}$.

3.1 High-level overview

Let $\alpha \in \mathcal{A}[\mathcal{T}]$ be an arbitrary producible assembly of $\mathcal{T}$, $\beta \in \mathcal{A}[\mathcal{U}_\mathcal{T}]$ such that $R(\beta) = \alpha$ (i.e. β is a producible assembly in $\mathcal{U}_\mathcal{T}$ which maps to α) and let $\vec{v} \in \partial^\mathcal{T} \alpha$ be a location in α's frontier. We will discuss the growth of tiles in $\mathcal{U}_\mathcal{T}$ into $\beta^m_{\vec{v}}$, which is the m-block supertile in $\mathcal{U}_\mathcal{T}$ representing the frontier space $\vec{v}$. Without loss of generality, assume that given β, no further tile attachments can occur in any of the m-block supertile locations adjacent to $\beta^m_{\vec{v}}$ (i.e. the adjacent supertile locations are currently "complete") and that no tiles have been placed inside of $\beta^m_{\vec{v}}$.

To explain how our construction works, we will break it into logical *modules*, or sub-assemblies which can be thought of as logical groups of tiles which perform specific computations or transfer of information.

The bulk of the explanation of our construction will revolve around simulating one time step, i.e. the placement of a single tile in $\mathcal{T}$, and the communication of the result (i.e. which tile was placed) to the tile's neighbors. The computation required to determine which tiles can be placed in which frontier locations is performed by a component we call the Adder. The Adder receives input from each of the neighboring tiles and performs a calculation to determine which tile or tiles can be placed given the input glues. When the adder determines that a tile has enough input glues to be placed it outputs a signal to a component called the bracket. The bracket ensures that only one tile type claims the location if there are multiple potential tile types that can be placed. All the information required to perform this computation, to construct the adder and the bracket, and to communicate the result with the neighboring tiles are encoded in a self-replicating structure called the genome. However, we will first describe the encoding of the structures which transfer information between components and supertiles.

3.1.1 Orientation

All tiles in $\mathcal{U}_\mathcal{T}$ use a relative orientation system used to establish direction of motion. This convention allows us to reorient easily, keeping the idea of "up", "forward", and "right" independent of the cardinal directions which might cause confusion. There are 24 distinct orientations, 4 orientations with respect to each of the

6 compass directions (North, East, South, West, Up, and Down). There are three orientation axes, D_1, D_2, and D_3. D_1 is called the primary or forward direction. D_2 is called the secondary or right direction. D_3 is called the tertiary or up direction. We describe each orientation using an ordered pair of letters from $\{N, E, S, W, U, D\}$. The first letter represents the forward direction. The second letter represents the upward direction. The orientation UN denotes that $+D_1$=Up, -D_1=Down, $+D_3$=North, -D_3=South, $+D_2$=East, -D_2=West. Notice that the upward direction must be orthogonal to the forward direction, thus given a forward direction, there are only 4 choices for the upward direction.

Major/Minor Orientation If we switch the direction of D_1, then D_2 and D_3 should stay constant. In other words, orientations with the opposite D_1 and the same D_3 should be differently handed. We arbitrarily define that orientations with N, E, or U as D_1 are major, left-handed orientations, and orientations with S, W, or D as D_1 are minor, right-handed orientations.

Critical Orientation The orientation NU is called the critical orientation in this construction. NU is the most natural orientation to work in, thus several components prefer this orientation. The Adder Array is entirely designed in the NU orientation, and the genome only activates datapaths when it is travelling in the NU orientation.

3.1.2 Movement Patterns

Many components in this construction use similar glue arrangements to move. While these patterns will be well-known to those intimately familiar with the aTAM, it is convenient to describe several commonly used glue arrangements to avoid confusion.

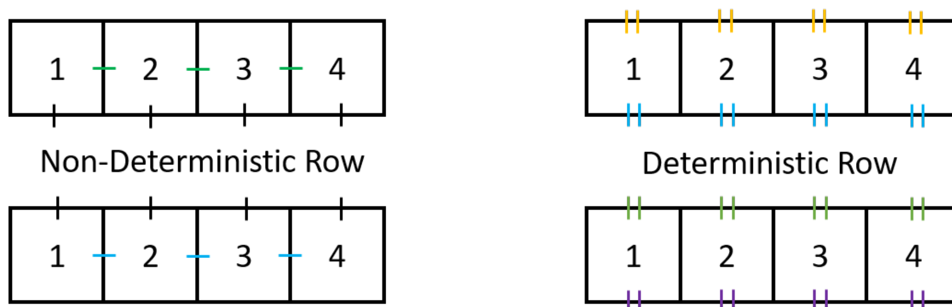


Figure 1: Comparison of collision using cooperation(left) vs strength 2 glues(right)

Collision Tolerance Collision tolerance is important in the growth of the genome. A collision will occur if a region of tiles grows from both the beginning and the end at the same time. The collision happens when the two growth directions meet across a single tile gap, as seen in *Figure 1*. In the left movement pattern in *Figure 1*, the black strength 1 glues communicate the data contained in tiles 1-4, while the different colored glues communicate a signal down the length of the row. In the collision row, forward/backward facing black glues are enough to cooperate across the gap without need of the signal. This will cause the signal that is passed along that row to choose non-deterministically between the available signals. The signal that would have been carried by that row if not for the collision might not be communicated correctly, which results in an error. The right movement pattern uses strength 2 glues, and so does not suffer from the problem of erroneous signals jumping the gap. In this case, there is only one variant of tiles with the cyan and green strength 2 glues, so that row maintains correctness and determinism throughout the collision.

Diagonal Advance The diagonal advance movement pattern uses a single strength 2 glue per row to advance into the next row and cooperation to fill in the row. The first tile into the row signals the tile

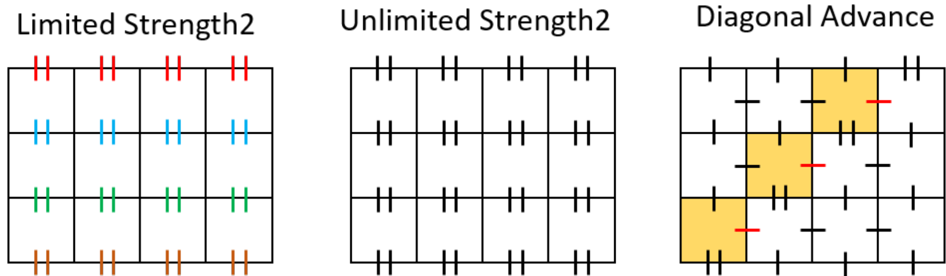


Figure 2: Three general movement strategies

immediately to its right or left to be the next tile to advance into a new row, thus the next row advancing tile can always be found one tile forward and one tile right or left of the previous advancing tile forming a diagonal line of tiles. This movement pattern is bi-directional but it is not collision tolerant.

Limited Strength 2 In the limited strength 2 movement pattern, each tile in a row uses a strength 2 glue on its forward face and a different strength 2 glue on its backward face. Using different glues means that the limited strength 2 movement strategy can be used to advance a constant distance. The limited strength 2 movement pattern is collision tolerant. If growth occurs from row 0 to row 1 and row 2 to row 1 at the same time, row 1 will always be the same. This pattern is desirable when a constant distance must be covered, but is useless if the required distance ever varies. This movement pattern is bi-directional and collision tolerant and is mainly used for movement where growth may occur from the start and end at the same time.

Unlimited Strength 2 The unlimited strength 2 movement pattern uses the same strength 2 glue on the forward and backward face. Once started this growth pattern will move in a straight line unless it collides with a previously placed "stopper" tile. This is desirable when there is a variable distance between two components. However, it introduces a dependency that the stopper tile(s) being placed *before* the unlimited strength 2 movement starts, otherwise it is impossible to guarantee this movement pattern won't result in infinite growth. This movement pattern is reversible and collision tolerant, though it generally only used from one direction.

Guide Rail The Guide Rail movement pattern has two sections, a single tile wide guide rail section and a payload section which can carry an arbitrary amount of data. The guide rail section may use either a limited or unlimited strength 2 movement pattern to move linearly in a single axis. The payload section presents a strength 1 glue representing the data being carried in the forward and backward direction. As the guide rail moves, the guide rail extends a strength 1 glue orthogonal to the direction of movement which allows the payload to use cooperation to fill in each new row that the guide rail has grown into. The payload can be thought of as "riding" the guide rail. The guide rail is used when the relative distance between two points is fixed or only varies in one axis, which mainly occurs within components. The adder array and the bracket use this movement pattern extensively.

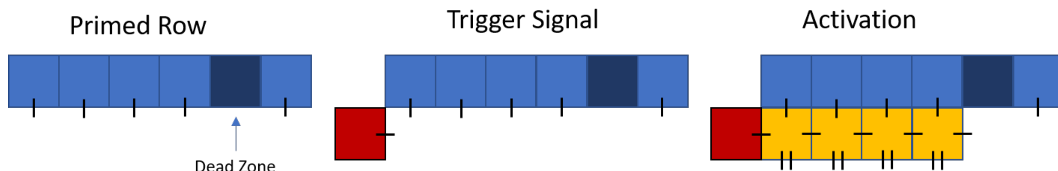


Figure 3: The Delayed Activation movement pattern

Delayed Activation Delayed activation, also known as priming, is a technique that allows a row of tiles to be activated by a single tile, like the fuse on a bomb. There are two stages to the priming movement pattern. The first stage, called the priming stage, happens when a row that is growing in a plane dangles strength 1 glues out of the plane. The second stage is called activation. Activation occurs when some event, like a query or a tile winning the bracket, triggers a tile that cooperates with the primed strength 1 glues. The activated row has strength 2 glues that start a datapath or trigger the growth of the genome into a neighboring cube. The cooperation chain in the activation stage will stop if it reaches a "dead zone" in which there is no dangling primed glue to continue the chain. Dead zones are important for callbacks in the initialization phase and for activating only the desired sections of the glue genome.

Latches and Keys The key and latch pattern appears in several different contexts in this construction. A latch is simply a row of tiles presenting only strength 1 glues into the next row. The data in the latch cannot grow into the next row until a key begins cooperation. A diagram of a latch can be found in *Figure 8*. In that figure, latch rows are denoted as L_{-1}. The Key and Latch movement pattern is used in three contexts for three different purposes. In the delayed activation movement pattern, the latch is used to allow data to be primed and the key is used as the activation trigger. In the circular latch, described later, the latch is used to allow growth in one direction and prevent growth in the other direction, similar to a one-way valve. In the tile placement variant of the datapath the latch is used to ensure a tile is placed before the datapath can continue to advance, a requirement which will be discussed in the datapath section.

3.2 Datapath

Throughout the construction, it will be very important to be able to both move data and place tiles in very specific locations. A datapath is a commonly used sub-assembly which allows data to be carried along a path encoded by a linear series of tiles. The encoding is done by tiles that represent instructions which describe how the datapath should move. As the datapath grows, its forward facing glues, relative to the movement of the datapath, encode data in strength 1 glues which can be read through cooperation once the datapath is finished growing. The datapath also has special left and right boundary tiles that signify the ends of the datapath during its growth. These are used during the execution of some of the instructions and to allow special callback tiles, which will be described later, to grow along the edges of the datapath.

3.2.1 Execution

A datapath consists of a contiguous line of tiles that encode the instructions that will be introduced momentarily. The left and rightmost tiles in this line are special boundary tiles. Each instruction tile has an active and inactive version. The inactive tile simply has a strength 1 glue, in the forward direction, encoding the type of instruction and data being carried, while the active version has a strength 2 glue, in the forward direction, encoding the same information. The forward propagation of a datapath is done using something very similar to the diagonal advance movement pattern with the only exceptions being during the execution of certain instructions. At any given time, there is only a single active instruction which causes the datapath to advance a single tile. Then, the next tile in the sequence becomes the active instruction. One thing to note is that, in our implementation of the datapaths, the instructions are executed from right to left. This choice is arbitrary however, and left to right executing datapaths could easily be constructed.

The only time when the datapaths propagation is not done using diagonal advance is during the execution of the forward, turn, and place instructions. In these cases, the propagation is handled specially as described below. However once these instructions finish, the diagonal advance pattern is continued starting at the next instruction in the sequence.

3.2.2 Instructions

The following are the various instructions which can be used by a datapath to either place data or move. Each instruction tile, in addition to encoding how the datapath should move, also encodes a single piece of information in its forward glue to be carried across the datapath.

Buffer The buffer instruction simply moves the datapath one tile forward. Typically this instruction is used to pad datapaths to a fixed length but it can also be used to move small distances forward. When active, this instruction is the quintessential diagonal advance tile with a strength 2 glue in the forward direction.

Forward The forward instruction is always followed by a series of tiles representing a number c in binary. Using a standard, fixed width, binary decrementer, the forward instruction causes the datapath to move forward c tiles as the number encoded is decremented until it equals 0. During the execution of a forward instruction, the forward propagation of the datapath is done by the strength 2 glues in the decrementer. Strength 1 glues along the left and right sides of the fixed with decrementer allow the tiles beyond the decrementer to fill in using cooperation. Once the decrementer is done, the instruction to the right of the tiles that used to represent c becomes the active instruction and propagation returns to the diagonal advance pattern.

One thing to note is that, in our implementation of the datapaths, the decrementer used in the forward instruction decrements the count on every other row of tiles. Furthermore, after the last row of the decrementer, so that the next instruction can be activated, another row of tiles is placed. Thus, for a number c represented in the tiles after the forward instruction, the datapath, in our implementation, propagates $2c+1$ tiles forward. This, however, is not an issue because the distances encoded in the tiles can simply be halved, plus or minus a single row using the buffer instruction, to account for the final single row.

Left/Right Turn The left and right turn instructions tell the datapath to turn in the respective direction relative to forward using a standard data rotation tile set. After the turn is done, the forward direction of the datapath's orientation is changed but the upward direction is not. For example a datapath with the orientation NU would have the orientation WU after a left turn.

Rise/Fall These instructions encode an upward or downward turn respectively. Once interpreted, these instructions place a row of tiles forward with glues facing either up or down corresponding to the forward facing glues encoding the data in the datapath. After one of these instructions, the orientation of the datapath is changed accordingly. For example after a rise instruction, a NU oriented datapath would face US. Likewise after a fall instruction, NU would become DN.

Place There are actually many place instructions, one corresponding to each different tile type that needs to be placed. When a place instruction is executed, the datapath is moved forward 1 tile and the corresponding tile is placed below, or downward relative to, the rightmost tile of the new row.

Stop This instruction causes the datapath to stop growing forward and ignore all of the remaining instruction. The data encoded in these instructions is still available to be read as strength 1 glues hanging off of the front of the datapath.

Variable The variable instruction, when executed, stops the growth of the datapath. The datapath remains stationary until some input data, growing orthogonally to the datapath using the unlimited strength 2 movement pattern, collides with the datapath at the variable instruction tile. This collision causes the data encoded in the input to replace the data that the datapath was previously carrying and begins the execution of the remaining instructions in the datapath.

3.2.3 Callback

During the initialization phase of the construction, which will be covered in more detail in following sections, some datapaths need to grow fully before it's safe for others to grow in order to preserve directedness. In order to do this, certain datapaths perform a callback in which, once a datapath has fully grown, a single tile wide path grows along its right or left boundary. This growth requires strength 1 glues available along the far end of the boundary and requires a single tile wide open space in which the callback can grow.

Since the boundary tiles don't require any tiles past the boundaries to operate these conditions are easily met. It should also be noted that it's not necessary for a datapath to grow callbacks. In order to allow for datapaths with callbacks and datapaths without callbacks, different left and right boundaries can be given to the datapath. Some boundaries contain the necessary glues for the callback to grow and the others do not.

Furthermore, there are three different types of callbacks. The first two types are the right and left end callbacks. These begin once the datapath has finished growing and they grow along the right and left boundaries respectively. The final kind of callback is a right variable callback. This grows once a variable instruction has caused the datapath to stop in order to wait for input. The reason why this variable is needed is because, during initialization, some of the datapaths will have variable instructions that will not receive input until later phases of the simulation. It's important that these datapaths are present before their input data begins growing otherwise the paths might not collide properly. Therefore the right variable callback allows for the datapath with the variable instruction to signal to the next path in the initialization sequence when it is ready to receive data, even though it has not finished fully growing.

Because callbacks have to grow backwards along the edges of their datapaths, it's important that they can perform all of the same turns that the datapath can. It's not difficult to see how a callback can propagate along the edge of a straight section of datapath, however the turn, rise, and fall instructions are a bit more complicated. [TODO describe how the callback can grow along turns]

3.3 Genome

We use the term *genome*, denoted G, to specify the the information necessary for U to simulate the given $\mathcal{T}$. Specifically, this includes encoding (1) the definition of the glues and tiles of $\mathcal{T}$, and (2) the information necessary to construct the Adder and Bracket modules with dimensions and locations dictated by the size of $\mathcal{T}$. The particular ways in which this information is encoded is discussed in this section.

3.3.1 Layout

The basic building block of the genome is a row, laid out in D_2. Rows are a sequence of tiles which have either a data or logistical role. Logistical roles help the genome perform a function but do not ever leave the genome; data roles are part of the instructions for a datapath which will, when activated, leave the genome to transfer data throughout the supercube. There are three sections of the Genome. G_1 contains instructions on how the genome should propagate and decides how each row of the genome should behave. G_2 contains an encoding of the glue relationship between tiles in $\mathcal{T}$ and the datapaths required to transmit that information to the adder array. G_3 contains datapaths which initialize the internal components of the supercube.

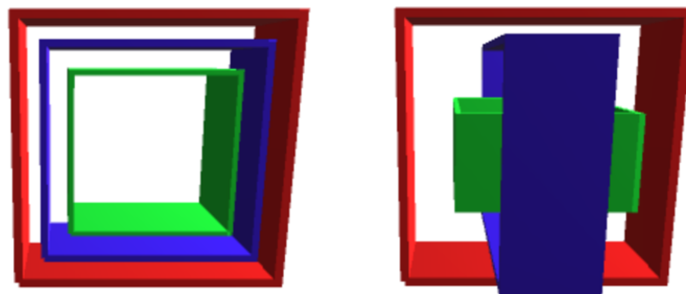


Figure 4: Bands (0:Red, 1:Blue, 2:Green) nest and rotate to form movement pattern

Bands The genome wraps around the supercube in three nested bands as seen in *Figure 4*. Each band contains four intersections. These intersections occur at the meeting of two unique orientations. The names take the form ABxCD where B and D are non-opposite faces and A is the opposite of D and B is the opposite of C (Example NUxDS and SUxDN). A detailed layout of the bands can be found in *Figure 5*. Each band has eight regions where movement instructions are read and four regions where the genome transfers to another band. Both of these regions must use the diagonal advance movement pattern. However, data can move in either direction around the band, which is problematic because the diagonal advance pattern is not bi-directional. This requires both the turn regions and the movement instruction regions to use a movement pattern called a circular latch to make the diagonal advance pattern bi-directional. These circular latch regions are interfaced with short regions using the limited strength 2 movement pattern.

Cross Band Communication The six square regions in *Figure 5* are called cross band communication regions or turn regions. In these regions, bands are separated from each other by a constant distance. One band will pass under the other band moving in the perpendicular direction. Turn regions are L rows wide and L rows long, where L is the number of tiles in the genome. Turn regions use a diagonal advance movement pattern. Along the diagonal data in the upper band lines up with the data in the lower band. At these points the upper band drops the data down to the lower band and the lower band raises the data up to the upper band using a limited strength 2 movement pattern to interface. After the turn is complete, it is impossible to tell whether the upper band or lower band grew in first.

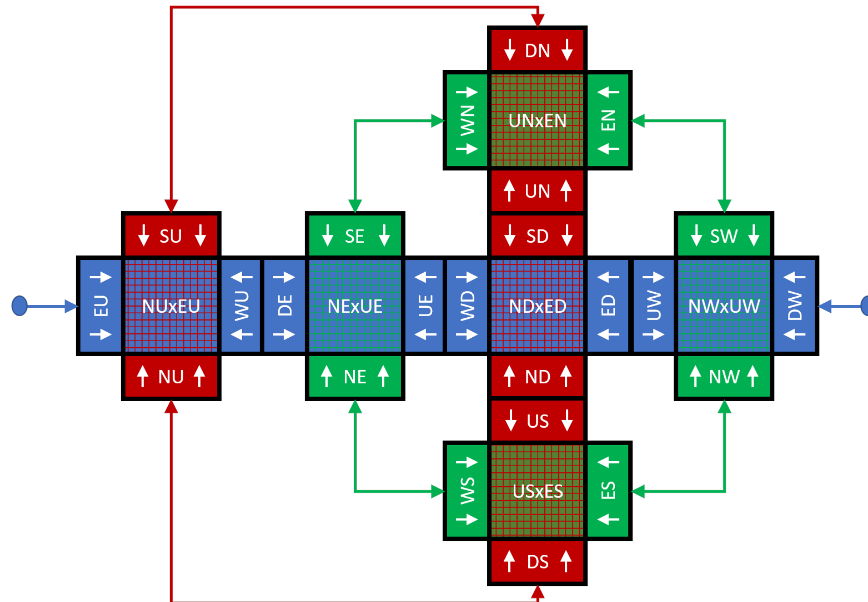


Figure 5: Layout of bands with orientation labels

3.3.2 G_1 - Movement

The first section of the genome is devoted to moving the genome around. This section contains four groups of movement instructions, one group for each band and one special group for the critical orientation. Each orientation knows which group it belongs to and reads the instructions in that instruction group one by one in a diagonal advance pattern. The signal that is read in each row propagates downstream (to the right), and the tiles rows behave differently depending on which instruction that row is performing. Because movement occurs in a diagonal advance pattern, the reading of the instructions must occur in a circular latch. Instructions in the non-preferred direction of the circular latch that would normally result in a primed row are ignored and treated as inert.

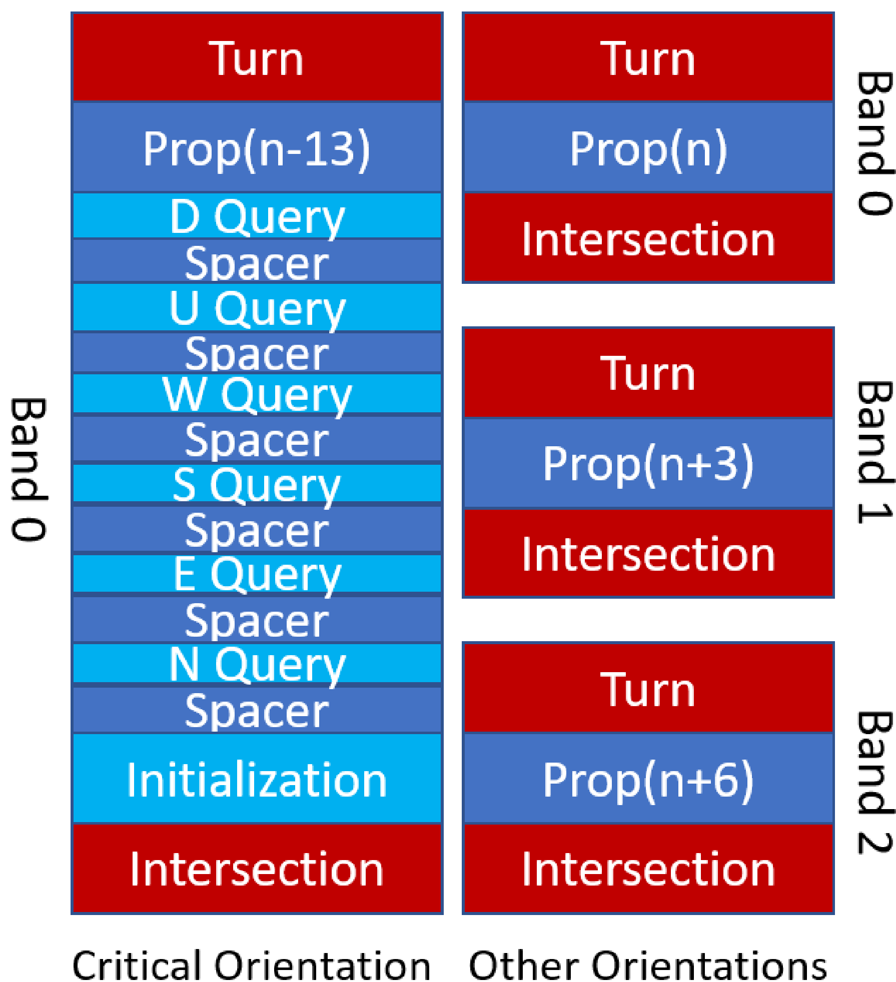


Figure 6: Four instruction groups, one for each of the three bands and one for the critical orientation.

Intersection This instruction signifies the end of a circular latch region and triggers the limited strength 2 interface to an intersection. This instruction occurs once per orientation, or 24 times per supercube.

Turn This instruction signifies the end of a circular latch region and triggers the limited strength 2 interface to a cross band communication region. This instruction occurs once per orientation, or 24 times per supercube.

Query This instruction signifies that a query may occur at this row of the genome. This row of the genome is a priming row that primes all datapaths in G_2. Although all datapaths are primed, the delimiters between each tile and each side within each tile are not primed which creates dead zones, ensuring that only the desired datapaths are activated. This instruction occurs six times per supercube only in the critical NU orientation. Queries that occur in the non-preferred direction of a circular latch are ignored

Initialize This instruction primes G_3 and generates the activation signal for G_3, immediately beginning initialization once this instruction is reached. This instruction occurs only once per supercube in the critical NU orientation.

Prop This instruction signifies an inert section of the genome which simply advances one tile. Since the A sequence of these forms a unary counter. Differences in the amount of these instructions is responsible for the nesting of the bands.

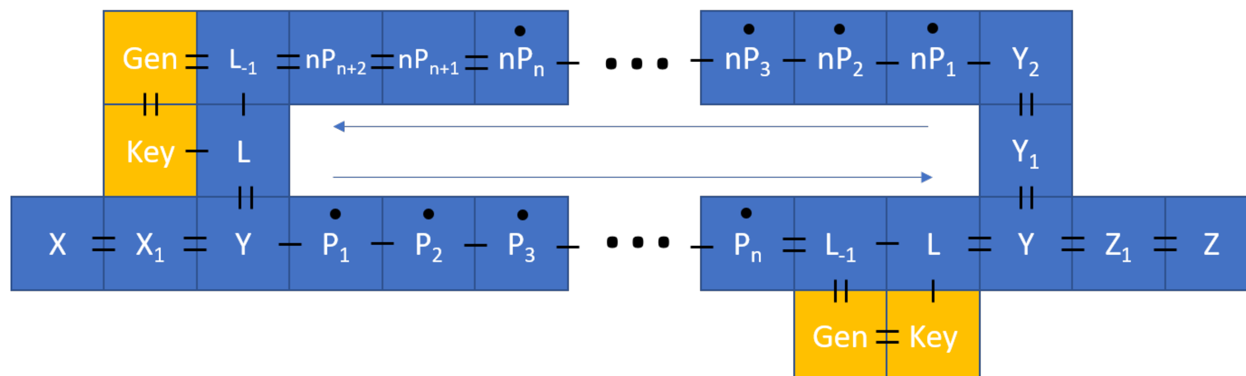


Figure 7: Side view of circular latch with glues. Each square represents a full row of data. Rows with dots use a diagonal advance movement pattern.

Circular Latch The Circular Latch is used exclusively by the genome to allow multiple supercubes to trigger genome growth at the same time. This can be solved using a combination of collision tolerant "two-way" regions and pairs of collision intolerant "one-way" regions, as shown in *Figure 8*. The two-way regions operate using strength 2 glues to move a constant distance. The one-way regions, called *circular latches*, use a single strength 2 glue per row to move forward and then cooperation to fill the row. Each circular latch has a *preferred* and a *non-preferred* direction. If data is to move in the non-preferred direction, it must rise out of its current plane and move forward until it reaches the next two-way region, at which point it can drop back down into the original plane. Dropping back into the regular plane will then trigger the preferred direction to grow until it collides with the original path. At the end of the process, it is impossible to tell whether the data grew from the preferred or non-preferred direction.

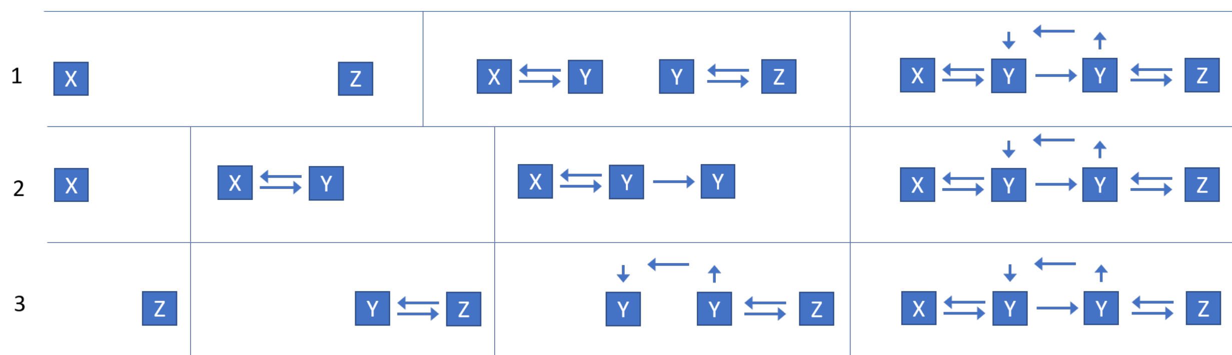


Figure 8: Three possible growth sequences between point X and point Z

Circular Latch Growth Sequences There are three types of growth patterns that can occur during a circular latch growth sequence, shown in *Figure 8*. In *Figure 8* Points X and Z represent some checkpoint in the genome movement between which there exists a circular latch; points labeled Y represent the interface between the two-way and one-way sections of the circular latch.

3.3.3 G_2 - Glue Table

Content and Generation G_2 encodes the glue relations between each t in $\mathcal{T}$ and encodes those relations in a datapath. To generate G_2 for t_0, first take the t_{0_N} glue and compare it to the t_{i_S} for all n tiles in $\mathcal{T}$. For each t_{i_S} that is a match, generate a datapath pointing to the A_{i_S}. Repeat, comparing all six opposite pairs of directions. Repeat these steps for each t_i in $\mathcal{T}$. Using this method, glues can be described solely as relations between tiles and can be represented as only the value of the glue strength. This information is encoded in the genome in a sequence as shown in *Figure 9*.

Queries After a neighboring tile decides what tile type it is, the current supercube receives a packet containing the neighboring tile number. That packet navigates to particular location and performs a glue table query. The query can be fully described as a duple of tile number and direction labeled as Q(i,d). The glue table hierarchy guarantees that all the information that a single query needs is located contiguously. Tiles in the glue table are separated by a tile called a "Tile Delimiter". Within each tile there are six regions representing each of the six faces of the tile which are separated by a tile called a "Side Delimiter". Following the example shown in *Figure 9*, Q(2,E) signals that the Eastern neighbor completed as t_2. The query increments a counter each time it encounters a Tile Delimiter and ignores everything until the next Tile Delimiter unless the counter is equal to i. When the correct tile is found, the query advances down the glue table until it encounters the correct Side Delimiter, at which point it generates a tile which activates all the datapaths until the next Side Delimiter or Tile Delimiter. These datapaths contain binary information about the strength of the glue and routing information required to get to the correct Adder Unit without colliding with any other datapath.

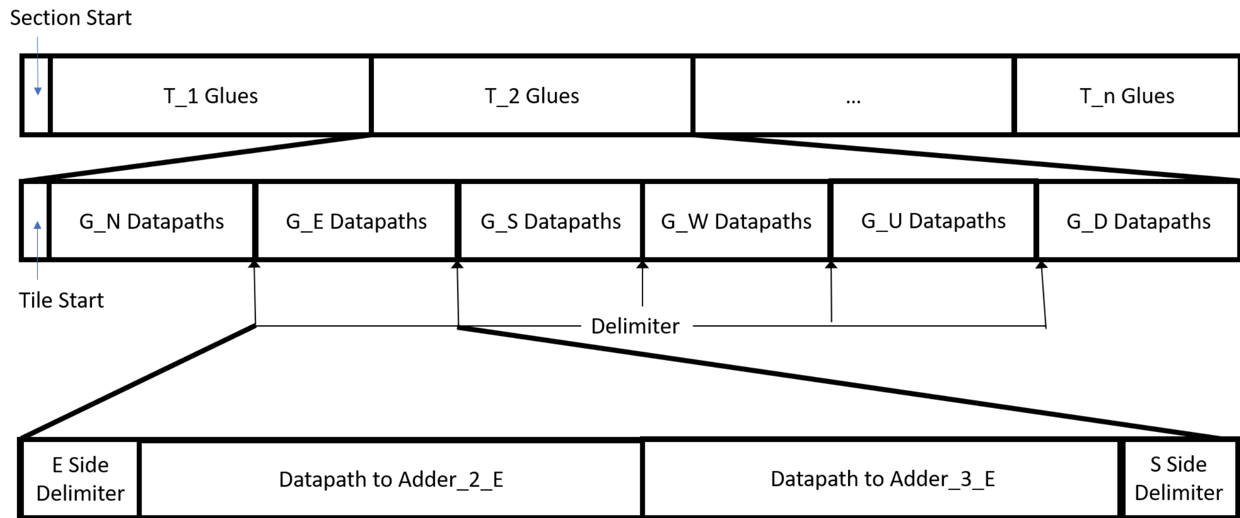


Figure 9: Glue table layout

3.3.4 G_3 - Initialization

Initialization consists of a sequence of datapaths that place tiles which seed the adder array, bracket, and external communication components. The order in which these datapaths grow is particularly important because some of these components will, at some point, grow tiles using unlimited strength 2 propagation. The datapaths must be guaranteed to place tiles that will stop the unlimited strength 2 propagation before the propagation begins, otherwise it's possible that the propagating tiles will grow beyond where they should have been stopped.

Since an adder in the adder array will, when it succeeds, propagate the successful tile information to the bracket using strength 2 glues, the bracket must be in place to catch the propagating signal before the adder

can determine if a tile can be placed. This is handled by making sure that the datapaths that place the bracket, from the initialization, grow fully before the datapaths that will place the adder seed. Moreover, one of the signals propagating with strength 2 will make it out of the bracket. This signal will be intercepted by the external communication datapaths that will grow to query the adjacent supercubes. [Daniel - We probably need a separate section talking about how the datapaths that do this use the variable instructions] This interception will allow them to, using the variable datapath instruction, carry the information regarding which simulated tile won the bracket to the adjacent supercubes. Therefore, the datapaths for external communication need to grow before the datapaths that will place the bracket, which themselves need to grow before the datapaths that will place the adder seed.

The datapaths for external communication will be the first datapaths along the activation strip. These will be activated immediately once the activation strip grows. They grow to a fixed position directly below where the bracket will be placed. They then will stop using a variable instruction until they receive the signal containing the winning tile number from the bracket. The variable instruction will cause a callback signal to grow along the right boundary of the datapath which will activate the first of the datapaths that will place the bracket. The bracket is placed using $\lceil \log_2 |T| \rceil$ datapaths which will place each of the levels in the bracket. Each of these datapaths will, once finished, grow a callback signal along the right boundary to cause the next one to grow. These datapaths need to be grown sequentially because the entire bracket needs to be placed before the adder should be allowed to grow. Once the final bracket placing datapath grows, its callback signal will activate the datapaths that will place the adder seed. The datapaths that place the adder seed can all be grown simultaneously so there does not need to be a callback between each of these datapaths.

3.4 Adder Array

During the simulation, it is necessary to determine whether or not a tile, in the simulated system, can bind with sufficient strength to grow into a given location. The adder array is responsible for calculating the total glue strength with which a tile can grow and comparing that strength to $\tau_{\mathcal{T}}$.

3.4.1 Hierarchy

Adder Array - A - The complete collection of adder units, with one adder unit for each t in T. 1 per supercube.

Adder Unit - A_i - An adder unit is a collection of component adders. If one component adder ends in a non-negative result then the adder unit succeeds. Inputs are denoted as A_i(d), where d is one of the six input directions.

Component Adder - A_i[x] - A component adder performs a sum of a subset of the six input glues and compares to τ. There are 63 component adders in one adder unit. Here, x is a 6-bit binary number between 1 and 63 where each bit represents whether to consider or ignore one of the input directions (LSB - N E S W U D - MSB).

Partial Adder - A_i[x][d] - Each component adder has seven partial adders, one for each of the six directions and one for comparison with tau.

Overview The Adder Array contains an Adder Unit (A_i) for each (t_i) in T. Each Adder Unit is responsible for determining if t_i is valid in the current frontier location. Since we do not know if or when a neighbor will complete, we must check each of the 6 choose n combinations of inputs. Disregarding 6 choose 0 there are 63 possible input combinations. Each Adder Unit contains 63 Adder components which are each responsible for checking one combination of inputs. Each component adder will not complete until *(*all) of its inputs are present, so it is necessary to consider each combination of inputs independently.

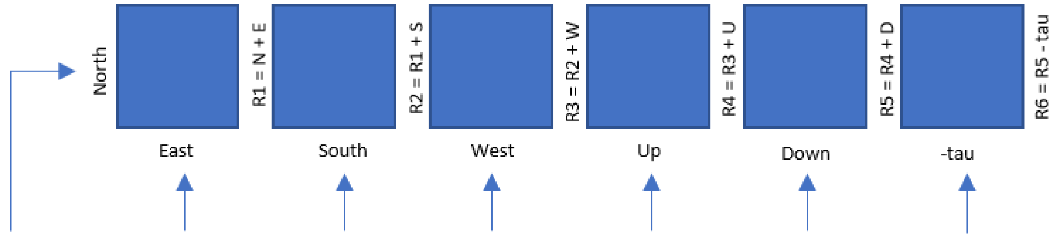


Figure 10: Partial Adders layout within a Component Adder

Success When an adder component receives all its required inputs it sums its inputs and subtracts τ. If the result is greater than or equal to zero then that adder component succeeds. If any of the component adders in A_i succeeds then A_i succeeds.

Inputs A_i receives a binary representation of a strength ($1 \leq S \leq \tau$) in one of six input slots which represent the neighbor direction. Each of the 6 inputs is delivered to each of the 63 component adders but will be ignored if that component adder doesn't consider it.

Output If A_i succeeds then A_i sends a binary representation of i to the bracket. The bracket is positioned such that the bracket input for t_i is directly below the output region of A_i and is guaranteed to be fully formed before A_i succeeds, so the output data rides to the bracket using strength 2 glues.

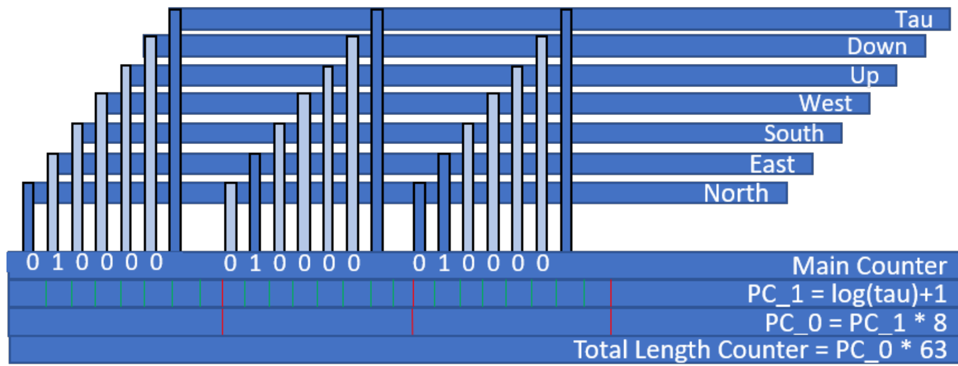


Figure 11: Adder schematic (side)

Periodic Counters The adder unit has several repeating structures which must appear multiple times, at regular intervals. Component adders must be regularly spaced along the adder, and partial adders must be regularly spaced along component adders. We use a variant of binary decrementer called a *periodic counter* to efficiently provide the required periodic structure. A periodic counter functions like a regular binary decrementer, except that it preserves its starting value throughout its count and resets to that value once it hits zero. If unrestricted, a generic periodic counter will continue repeating infinitely many times, which is not a desired behaviour in this construction. We use a simple (non-periodic) decrementer initialized with the total desired length of the periodic counters to restrict the periodic counters. A periodic counter receives signals from the layer below it and send signals to the the layer above it. Periodic counters have two states: zero and non-zero. When non-zero, the periodic counter passes a "continue" signal to the layer above. When zero, the periodic counter sends a "zero" signal unique to its layer to the the layer above. When multiple periodic counters are stacked, as in *Figure 11*, multiple zero signals may occur in several layers at the same point. In this case, the bottom-most periodic counter's zero signal takes precedence and is displayed to the uppermost layer (See *Figure 12*).

Active		Foo		Foo		Bar		Foo		Foo		Bar		Foo
PC_1 (P=2)	1	0	1	0	1	0	1	0	1	0	1	0	1	0
PC_0 (P=6)	5	4	3	2	1	0	5	4	3	2	1	0	5	4
Total (L=14)	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 12: Example of a 2-layer periodic counter

Main Counter The main counter layer contains the binary representation of the component adder. Recall that a component adder is addressed by a six bit binary number where each bit represents whether to check or ignore input from a particular neighbor. The main counter begins with the component adder address 000001 and increments each time the PC_0 sends a zero signal. Within each component adder, partial adders are constructed to be $log_2(\tau) + 1$ bits wide. PC_1 is set to repeat this distance plus one to allow a gap for the result of the addition to advance. The component adder always considers the least significant bit (LSB) of the main counter first. If the LSB is 1, then the partial adder is constructed to wait for input from the northern neighbor. If the LSB is 0, then the partial adder is constructed to ignore the input from the northern neighbor and present zero as the input. When the PC_1 sends a zero signal, the main counter leaves a gap row and then constructs the partial adder of the next highest bit.

Inputs The top half of *Figure 11* shows that each input is sent into each component adder. Each input is presented to each component adder, either to be rejected or accepted depending on the address of the component adder. Not shown in *Figure 11* is that each of the inputs is guided by a copy of the Total Length Counter and PC_0. Whenever the PC_0 zero signal is sent, the input data forks itself and drops down into the corresponding partial adder. The guide rails are offset in the the forward direction by a multiple of the period of PC_1 to ensure that the input drops into the correct location.

3.5 Bracket

Once an adder A_i succeeds, a binary representation of i is propagated to the bracket. This propagation signifies that tile $t_i \in \mathcal{T}$ is a valid tile which could grow into the simulated tile location. In fact, because the simulated system $\mathcal{T}$ might have multiple valid tiles that can grow into any specific location, there may be multiple successful adders, in which case a number of binary representations will be propagated to the bracket. This propagation is done by a single tile wide carrier backbone that moves using strength 2 glues. The binary information is carried down the backbone using strength 1 cooperation (Perhaps an image would be useful). Because the backbone moves using strength 2 glues, it must move in a straight line and cannot stop without being blocked. The bracket, by blocking and cooperating with these propagating backbones, guides them through paths resembling those in a tournament bracket. This is done in such a way that at any points where multiple paths might intersect, only the first arriving backbone will be allowed to propagate further into the bracket. This tournament styled bracket is the mechanism by which the supercube determines which of the potential tiles from $\mathcal{T}$ to represent. Furthermore when simulating a directed system in which there are never multiple distinct tiles that can grow into a location, the bracket will allow the single binary representation to pass through without any non-determinism.

Guiding the backbone is done using two different two-tile wide blockers called turn barriers and merge barriers. These barriers are placed during the initialization phase of the simulation and are guaranteed to be in place before the adders can start the propagation of the binary information. The backbone has strength 2 glues in the downward direction allowing it to propagate indefinitely downward. It's East and West glues however, have strength 1. Once the backbone collides with a one of the tiles of a barrier, these strength 1 glues will allow the backbone to cooperate with the second tile of the barrier to change direction. The data

riding along the backbone also has strength 1 glues to the East and West so that when the backbone changes direction, the data can change direction and continue to ride along the backbone.

3.5.1 Turn Barriers

If the barrier is a turn barrier, the backbone, after cooperating, will begin to propagate using strength 2 glues in the East or West direction depending on which side it cooperated. This propagation is temporary though and will only last until the backbone collides with a merge barrier. This is extremely similar behavior to the *rise* and *fall* instructions in the datapath however there is no secondary orientation associated with the data being moved. During this sideways propagation, the backbone will have a strength 1 glue in the upward direction to allow it to cooperate with the merge barrier once it collides.

3.5.2 Merge Barriers

[TODO]

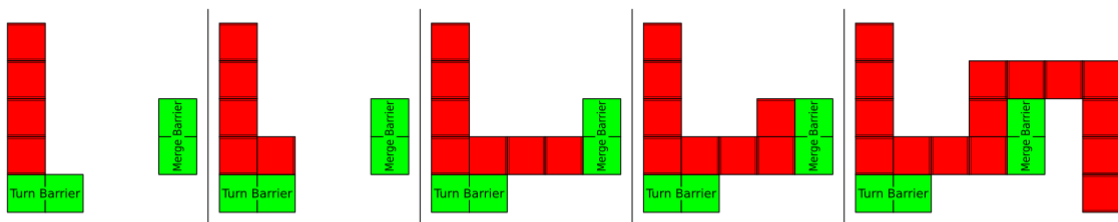


Figure 13: (1) Backbone propagates until it collides with turn barrier (2) Cooperation between backbone and barrier (3) Eastward propagation until collision with merge barrier (4) Cooperation with merge barrier (5) Strength 2 path up and around merge barrier after which data propagates downward again

3.6 Supercube layout

TODO: describe how the main components are organized within a supercube, and the process of creating each of them from an "input" provided by an adjacent supercube.

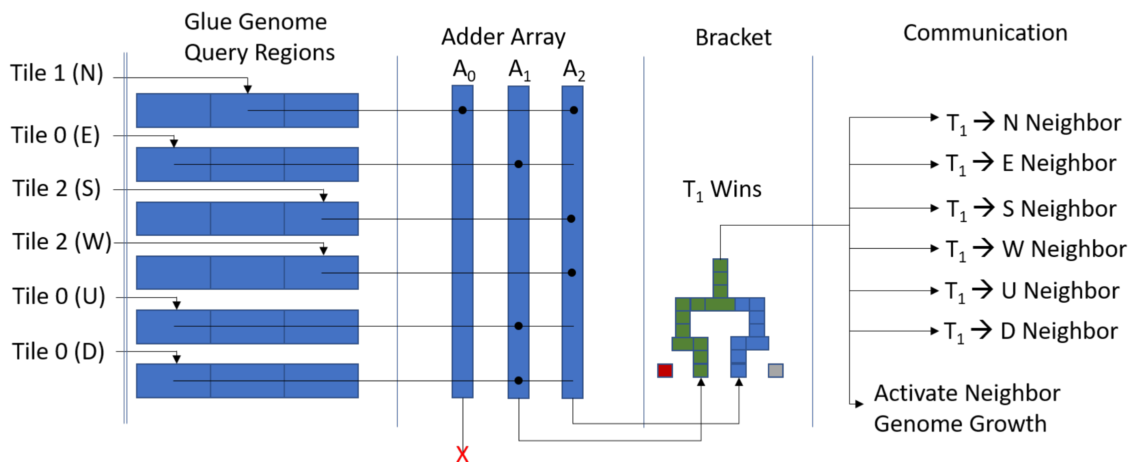


Figure 14: Complete data flow from query to communication. Initialization assumed to have completed.

Sequence The construction may be thought of as occuring sequentially in the following order. However, in reality several of these steps may occur in parallel. Parallelism is only a problem if unexpected or out of control growth can occur if a dependency is not present. There are only two processes in this construction that have that dependency: the adder output to bracket input and the bracket output to external communication input. These two processes use strength 2 glues to propagate until collision with a geometric blocker. If the expected blocker is not placed before the strength 2 glues begin to propagate then the construction will fail. This issue is resolved in the initialization stage which uses callbacks to ensure that all dependencies are fully complete before the adder array can send an output to the bracket.

1. Genome Growth
2. Initialization
   (a) External Communication
   (b) Bracket
   (c) Adder Array
3. Query
4. Adder Success
5. Bracket
6. Neighbor Genome Growth Activation
7. External Communication

Initial Growth Phase A target supercube has six neighboring supercubes. The target supercube has one genome input region from each of its neighbors and has one genome output region to each of its neighbors. Except for seed supercubes, a target supercube is empty until one of its genome input regions receive data from the corresponding neighbor's genome output region. Once the genome input region has received data, genome growth begins in the target supercube as described in [reference movement section] and will continue until the genome shell is complete. As part of the initial growth phase the genome will reach a specific orientation called the critical orientation. The critical orientation, in our system arbitrarily chosen to be NU, is the only orientation that exposes "Query" and "Initialize" instructions. Once the critical orientation is reached the initialization and query instructions may occur in parallel.

Query A query activates datapaths which carry glue strength data from the glue genome to adders in the adder array. The mechanism for activation is described in section [reference query section]. The query may occur in parallel with the initialization step, but will be inert until the target adder is initialized.

Adder Success Each adder in the adder unit will wait for input from the six potential queries. When an input enters an adder, each of the component adders will receive that input. If the current combination of inputs causes at least one component adder to succeed, then the adder will succeed and pass its tile number to the bracket.

Bracket The bracket receives tile numbers from the successful adders in the adder array. The tile numbers propagate through the points of competition in the bracket until one tile number reaches the final point of competition. The target supercube is now differentiated. Differentiation starts two processes, genome growth activation and external communication.

Genome growth activation When a tile number wins the bracket, the first external communication variable datapath receives the tile number. Upon reception of data, a callback is initiated which sends a signal back to the genome. This signal propagates along the edge of the genome using cooperation to reach each of the twelve intersections. Upon completion of genome growth, each intersection has two internal directions and two external directions. The external directions are primed and ready to be activated when this callback signal reaches the intersection. For each face of the supercube, only one output region and only one input region is required. These regions are chosen arbitrarily, with the only constraint being that the input direction of a face must be the opposite direction as the output region of the opposite face. This results in only 12 of the 24 orientations being involved in genome growth activation, therefore it is possible to entirely eliminate parts of the three band structure to decrease the number of tiles in the IU tileset. After crossing a small inter-tile gap the output tiles place a copy of one of the intersections of the neighboring supercube, which begins growth phase 1 for that neighbor.

External communication When a tile number wins the bracket, that tile number is placed into the payload of the six external communication datapaths, one for each neighboring supercube. The datapaths' instructions describe a path from the pickup location to the correct query location in the neighboring supercube. The direction the query came from is encoded spatially by the layout of the genome, therefore by navigating to the correct location in the neighboring supercube and depositing the correct payload the external communication datapath initiates the correct query Q(i,d) where i is the winning tile number and d is the opposite direction of the neighbor.

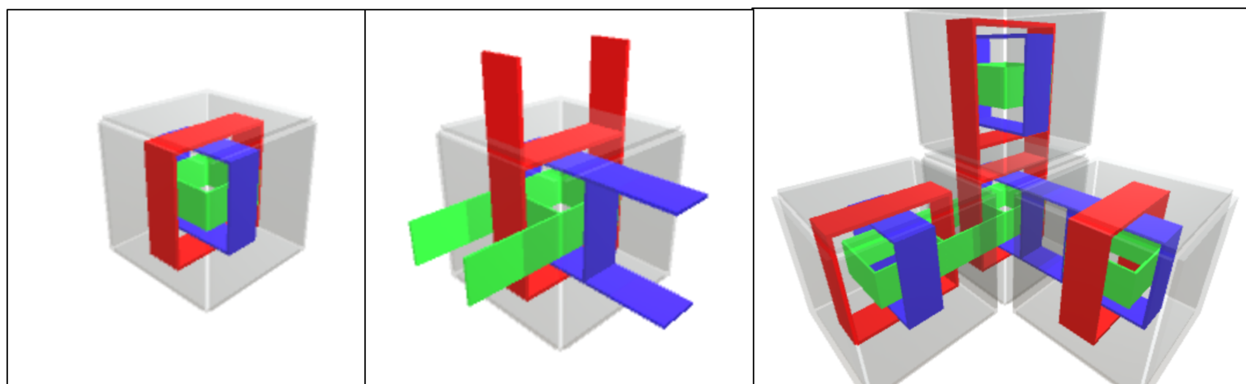


Figure 15: External Communication Stages

3.7 Dynamics and correctness

3.7.1 Seed

The seed supercube(s) consists of a single row of the genome and a seeded bracket.

Genome Any correctly placed row of the genome will grow into the full three bands of the supercube genome, so we arbitrarily picked the intersection between NU and DS to seed the genome.

Bracket The tile that wins the bracket The bracket must be seeded such that the seed tile is guaranteed to win all of its points of competition. This solves a problem in maintaining determinism that arises in multi-seeded assemblies.

3.7.2 Correctness

Input/Output Loop The outputs of one supercube are the inputs of another supercube. The genome growth activation step triggers genome growth in each of the neighboring supercubes and the external communcation step triggers a query to each of the neighboring supercubes. This loop will continue until there are no more frontier supercubes which have enough inputs to differentiate into a tile, at which point the simulation terminates.

References

[1] Erik D. Demaine, Matthew J. Patitz, Trent A. Rogers, Robert T. Schweller, Scott M. Summers, and Damien Woods, *The two-handed tile assembly model is not intrinsicall universal*, Algorithmica **74** (2016), no. 2, 812–850.

[2] David Doty, Jack H. Lutz, Matthew J. Patitz, Robert T. Schweller, Scott M. Summers, and Damien Woods, *The tile assembly model is intrinsically universal*, Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, 2012, pp. 302–310.

[3] Jacob Hendricks, Matthew J. Patitz, and Trent A. Rogers, *Universal simulation of directed systems in the abstract tile assembly model requires undirectedness*, Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016), New Brunswick, New Jersey, USA October 9-11, 2016, pp. 800–809.

[4] Pierre-Étienne Meunier, Matthew J. Patitz, Scott M. Summers, Guillaume Theyssier, Andrew Winslow, and Damien Woods, *Intrinsic universality in tile self-assembly requires cooperation*, Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA 2014), (Portland, OR, USA, January 5-7, 2014), 2014, pp. 752–771.

[5] Matthew J. Patitz, *Simulation of self-assembly in the abstract tile assembly model with isu tas*, Tech. Report 1101.5151, Computing Research Repository, 2011.

[6] Paul W. K. Rothemund and Erik Winfree, *The program-size complexity of self-assembled squares (extended abstract)*, STOC '00: Proceedings of the thirty-second annual ACM Symposium on Theory of Computing (Portland, Oregon, United States), ACM, 2000, pp. 459–468.

[7] Erik Winfree, *Algorithmic self-assembly of DNA*, Ph.D. thesis, California Institute of Technology, June 1998.