ALCH: An Imperative Language for Chemical Reaction Network-Controlled Tile Assembly

Titus H. Klinge · James I. Lathrop · Sonia Moreno · Hugh D. Potter · Narun K. Raman · Matthew R. Riley

Abstract Schiefer and Winfree recently introduced the chemical reaction network-controlled tile assembly model (CRN-TAM), a variant of the abstract tile assembly model (aTAM). In the CRN-TAM, tile reactions are mediated via non-local chemical signals controlled by a chemical reaction network. This paper introduces ALCH, an imperative programming language for specifying CRN-TAM programs that can be compiled and simulated. ALCH includes standard language features such as Boolean variables, conditionals, loops, and CRN-TAM-specific constructs such as adding and removing tiles. ALCH also includes the *branch* and *parallel* structures which harness the nondeterministic and parallel nature of the CRN-TAM. ALCH also supports functional tileset specification. Using ALCH, we show that the discrete Sierpinski triangle and the discrete Sierpinski carpet can be strictly self-assembled in the CRN-TAM, which shows the CRN-TAM can self-assemble infinite shapes at scale 1 that the aTAM cannot. ALCH allows us to present these constructions at a high level, abstracting species and reactions into C-like code that is simpler to understand. We employ two new CRN-TAM techniques in our constructions. First, we use ALCH's nondeterministic branching feature to *probe* previously placed tiles of the assembly and detect the presence and absence of

T. Klinge
Drake University
E-mail: titus.klinge@drake.edu

J. Lathrop
Iowa State University
E-mail: jil@iastate.edu

S. Moreno
E-mail: smsoniamoreno@gmail.com

H. Potter
Iowa State University
E-mail: hdpotter@iastate.edu

N. Raman
E-mail: narun.raman@gmail.com

M. Riley
Iowa State University
E-mail: mrriley@iastate.edu

tiles. Second, we use scaffolding tiles to precisely control tile placement by occluding any undesired binding sites. This paper is an extension of our previous work, updated to include a Sierpinski carpet construction and the *parallel* command.

Keywords Tile Assembly · Chemical Reaction Network · Sierpinski Triangle · Sierpinski Carpet · Molecular Programming Language

1 Introduction

Molecular programming is a relatively new field that weaves together biology and computer science to specify the behavior of molecules at the nanoscale. Early research in the field was sparked in 1982 by Seeman's pioneering work employing DNA crossover tiles to self-assemble crystals at the nanoscale [24]. Seeman's work was later extended by Erik Winfree to include *cooperative* DNA tile self-assembly to construct more complex shapes and patterns [28]. Winfree formalized the abstract tile assembly model (aTAM) in his Ph.D. thesis, where he proved it is Turing complete [28]. As a result, the aTAM is considered a programming language for self-assembling two and three-dimensional nanoscale patterns and is still actively investigated today [19,7,21,12,15,13].

Another commonly used model for biomolecular computation is the *chemical reaction network* (*CRN*), which models the interactions of abstract chemical species. The CRN model assumes the solution is well-mixed, so computations are amorphous and do not rely on geometry or structure. Two common variants of the CRN model are *stochastic CRNs* and *deterministic CRNs*. Stochastic CRNs [14,5, 25,29] are modeled with discrete species counts, and their reactions are probabilistic; they are also closely related to other prominent models of computation such as population protocols [1,6]. Deterministic CRNs [9,11,17] model the species' state continuously with real-valued concentrations governed by a system of autonomous ordinary differential equations (ODEs). Both variants are known to be Turing complete [26,10] and can be compiled into DNA strand displacement systems that approximate their behavior [2].

In 2015, Schiefer and Winfree introduced the *chemical reaction network-controlled tile assembly model* (*CRN-TAM*) [22,23]. Their model combines the amorphous properties of stochastic CRNs with the spatial self-assembly of complex structures afforded by the aTAM. More specifically, a CRN interacts with tiles from the aTAM model to exert non-local control over the self-assembly process.

Overall, molecular programming is a rich field for algorithmic study. However, it is challenging, tedious, and error-prone to manually specify chemical species, tiles, or reactions when developing a molecular algorithm. To mitigate this, new software tools and molecular programming languages are being developed that offer a higher level of abstraction to molecular programmers. For example, Vasić, Soloveichik, and Khurshid developed CRN++, a high-level language for implementing deterministic CRN programs [27]. The CRN++ language provides a toolset for manipulating concentrations as numerical variables, with some support for conditionals and loops. This simplifies the development of high-level deterministic CRNs by abstracting away many low-level details. Similarly, Liekens and Fernando's Chemical Bare Bones (CBB) is a hypothetical chemical implementation of the simple but Turing complete Bare Bones programming language [20]. CBB

implements increment, decrement, and loop instructions using a catalytic particle model in which a single multistate particle catalyzes reactions based on its state. Another example is Cardelli's Kaemika App [4], which incorporates a simple functional programming language to streamline the generation of complex CRNs.

Several high-level abstractions have also been investigated for the tile self-assembly model. For example, Becker developed a geometry-based system for generating shapes in the aTAM [3]. It allows users to describe how information and assembly construction propagate along vectors defined in the physical space of the assembly. Users can then generate an aTAM system by designing a system of vectors and applying a well-defined procedure to convert it into tiles. Doty and Patitz also developed an aTAM toolset, abstracting how information is shared via the connections between individual tiles [8]. Users can generate an aTAM system by specifying variables that are transmitted from tile to tile via bond labels and transformation functions to "modify" those variables within a tile. Both of these tools focus on the parallel, semi-uncoordinated concept of tile self-assembly typical of aTAM constructions. In the CRN-TAM, on the other hand, the CRN component allows precise control over which tiles are added and when.

CRN-TAM constructions often rely on sequences of reactions and tile attachments, with sequential execution enforced by associating a chemical species with each reaction in the chain. For this reason, the CRN-TAM is a natural fit for a high-level imperative programming language. In this paper, we present the Algorithmic Language for Chemistry (ALCH), an imperative language for specifying CRN-TAM programs. ALCH targets the specific CRN-TAM design paradigm described above, where the CRN component mediates a strictly controlled sequence of tile actions. However, it also includes facilities to harness the massively parallel nature of the CRN-TAM. We implemented a software compiler to translate ALCH code into proper CRN-TAM programs as well as a simulator that visualizes the assembly of a CRN-TAM program[1].

ALCH is reminiscent of other popular imperative languages, supporting loops and conditionals but omitting numerical computation and function calls. ALCH also contains many CRN-TAM specific statements that abstract away low-level details of the model's underlying semantics while maintaining that statements are executed in sequence. Although ALCH is an imperative language, it supports parallel assembly via the *parallel* statement, which generates multiple threads, each with their own sequence of instructions to be executed in parallel. ALCH also includes a *branch* statement, which is a control structure that nondeterministically chooses between a finite number of self-assembly paths. In contrast to the parallel construct, nondeterministic branching in ALCH is implemented *sequentially* with a reversible random walk so that only the effects of the succeeding branch remain on the final assembly. ALCH also includes syntax for specifying tilesets in a functional way, which eases the burden of specifying CRN-TAM programs with a large number of tiles. We have designed ALCH to specifically target sequential CRN-TAM constructions, with chemical parallelism entering in a controlled way through multithreading. Although we are not aware of any shape that can be constructed in the CRN-TAM but not in ALCH, we do not claim that ALCH is as

[1] The ALCH compiler and the CRN-TAM simulator, together with examples and visual illustrations, are available at `http://web.cs.iastate.edu/~lamp`.

general as the CRN-TAM; we expect that some valid CRN-TAM constructions will be impossible to represent in ALCH.

Using ALCH, we demonstrate that the CRN-TAM can construct infinite shapes that the aTAM cannot. For example, the discrete Sierpinski triangle is a well-known self-similar fractal that can be *weakly* self-assembled in the aTAM [28] but cannot be *strictly* self-assembled [19]. Weak self-assembly allows for "filler" tiles to be used to propagate information through an assembly, whereas strict self-assembly forbids this. Using ALCH, we show that the non-local communication provided by the CRN-TAM is sufficient to overcome this limitation and specify a CRN-TAM program that strictly self-assembles the discrete Sierpinski triangle. Our construction relies on the ability to add and remove scaffolding tiles and self-assembles the fractal by probing previous placed tiles on the assembly. We also use the scaffolding tiles to occlude any spurious bonding sites, giving precise control over the placement of the next tile. The construction proceeds in a sequence of stages where each stage successfully self-assembles a subset of the discrete Sierpinski triangle. After the completion of a stage, all scaffolding tiles are removed, leaving only the Sierpinski triangle tiles. Thus, in the limit, only the Sierpinski triangle remains, since the scaffolding tiles are removed infinitely often. In fact, the ratio of scaffold tiles to Sierpinski triangle tiles approaches zero as the self-assembly process proceeds.

The discrete Sierpinski carpet is a self-similar discrete fractal related to the Sierpinski triangle. The kernel for this fractal is a 3 by 3 grid with only the center missing, and was shown to assembly weakly in [16]. In this paper we use the ALCH language to define a CRN-TAM that strictly assembles the infinite discrete Sierpinski carpet (DSC). It is unknown if the infinite DSC can be strictly self-assembled in the aTAM. The ALCH programming language and simulator simplifies the development process and the specification of the CRN-TAM system.

We first defined ALCH and presented the Sierpinski triangle construction in our prior work [18]. This paper presents the same material, and additionally extends ALCH to support parallelism via the *parallel* command. We have also added a strict Sierpinski carpet construction to demonstrate the generality of our triangle construction technique. Readers may consult the appendix of [18] for a more in-depth explanation of the triangle construction than we present below.

The rest of the paper is organized as follows. Section 2 gives an overview of the CRN-TAM model. Section 3 presents a detailed description of the ALCH programming language, including how each statement is compiled to the CRN-TAM. Section 4 gives an overview of the construction for the discrete Sierpinski triangle using the ALCH language, with examples to illustrate key concepts such as probing using nondeterministic branching. Section 5 discusses how to construct the Sierpinski carpet using ALCH in relation to the construction of the Sierpinski triangle. Finally, Section 6 discusses some conclusions from this work.

2 Preliminaries

We now review the chemical reaction network-controlled tile assembly model (CRN-TAM), which combines the notions of the abstract tile-assembly model (aTAM) [28] and the stochastic chemical reaction network (sCRN) [5]. For a complete introduction to the model, see Schiefer and Winfree's original paper [22].

A *tile type* is a tuple $\boxed{t} = (N, E, S, W)$ consisting of four *bonds* for the north, east, south, and west sides of the tile, respectively. Each bond is a tuple $B = (\ell_B, s_B)$ where ℓ_B is the *label* and s_B is the *binding strength* which is a non-negative integer. Given a finite set of tile types T, an *assembly* is a partial function $\alpha : \mathbb{Z}^2 \to T$ that encodes the positions of tiles in two-dimensional space. If $\alpha(i, j)$ is undefined, then we say that (i, j) is *unoccupied* in the assembly α. When two adjacent tiles in α have matching bond labels ℓ_N on their abutting sides, we say that they *interact* with a strength determined by their bond strengths s_B. We should note that, unlike in the original CRN-TAM model, we allow adjacent bonds with the same label to interact with strength s, where s is given by the minimum of the bond strengths. Our probe mechanism, discussed in a subsequent section, relies on such asymmetric bonds. However, this is not fundamental to the construction and only serves to simplify the construction.

The *binding graph* of an assembly α is a two-dimensional lattice of vertices representing the tiles of α where two vertices are connected by an undirected edge with weight s if their corresponding tiles in α interact with strength s. For $\tau \in \mathbb{N}$, we say that an assembly is τ-stable if the minimum cut of its binding graph is at least τ. We also denote assemblies using $\boxed{\boxed{\alpha}}$, and given a tile type $\boxed{t}$, use $\boxed{\boxed{t}}$ to denote the singleton assembly that consists of only a single tile of type t placed at the origin. Note that the number of tiles of a given tile type $\boxed{t}$ available in solution is finite but unbounded. This is in contrast to the aTAM which assumes an unlimited supply of all tile types throughout the self-assembly process.

A *signal species* is an abstract molecule type. In contrast to tiles, signal species have no geometry and are used to facilitate non-local communication in the self-assembly process. Every tile $\boxed{t}$ has a unique *removal species* t^*, and given a finite set T of tile types, we write $T^* = \{t^* \mid \boxed{t} \in T\}$ to denote the set of all tile removal species of T.

A *CRN-TAM program* is a tuple $\mathcal{P} = (S, T, R, \tau, I)$ where T is a finite set of tile types, S is a finite set of signal species that satisfies $T^* \subseteq S$, $\tau \in \mathbb{N}$ is the *temperature*, $I : S \cup T \to \mathbb{N}$ is the *initial state* which specifies how many tiles and signal molecules are initially present, and R is a finite set of reactions that are of the following six types.

1. **Signal** reactions are of the form $X_1 + X_2 \to Y_1 + Y_2$ where $X_1, X_2, Y_1, Y_2 \in S \cup \{\epsilon\}$. The ϵ symbol denotes the absence of a species, therefore $X + \epsilon \to Y_1 + Y_2$ is equivalent to $X \to Y_1 + Y_2$. Since these reactions only consist of signal species, their semantics are identical to those in the traditional sCRN model. The species on the left-hand-side are called *reactants* and are consumed by the reaction and the species on the right-hand-side are called *products* and are produced by the reaction.
2. **Deletion** reactions are of the form $X + \boxed{t} \to Y_1 + Y_2$ where $X, Y_1, Y_2 \in S \cup \{\epsilon\}$ and $\boxed{t} \in T$. These reactions consume a tile, treating it as if it were a signal species. Note, deletion reaction cannot consume tiles bound to the assembly.
3. **Creation** reactions are of the form $X_1 + X_2 \to \boxed{t} + Y$ where $X_1, X_2, Y \in S \cup \{\epsilon\}$ and $\boxed{t} \in T$. These reactions produce tiles, making them available to interact with assemblies.
4. **Relabelling** reactions are of the form $X + \boxed{t_1} \to Y + \boxed{t_2}$ where $X, Y \in S \cup \{\epsilon\}$ and $\boxed{t_1}, \boxed{t_2} \in T$.

5. **Activation** reactions are of the form $X + \boxed{t} \to \boxed{\boxed{t}} + t^*$ where $X \in S$, $\boxed{t} \in T$, and t^* is the signal removal species for $\boxed{t}$. These reactions use tile $\boxed{t}$ to seed a new assembly with $\boxed{t}$ placed at the origin.
6. **Deactivation** reactions are of the form $\boxed{\boxed{t}} + t^* \to \boxed{t} + Y$ where $\boxed{t} \in T$, t^* is the removal signal for $\boxed{t}$, and $Y \in S \cup \{\epsilon\}$. These reactions remove the tile $\boxed{t}$ from the singleton assembly $\boxed{\boxed{t}}$, thereby deactivating it.

In addition to the reactions above, for each $\boxed{t} \in T$, the following two reactions included in the set of reactions R.

1. **Addition** reactions of the form $\boxed{\boxed{\alpha}} + \boxed{t} \to \boxed{\boxed{\beta}} + t^*$ where $\boxed{\boxed{\beta}}$ and $\boxed{\boxed{\alpha}}$ are $\tau-$stable assemblies that differ by one copy of $\boxed{t} \in T$ and $t^* \in T^*$ is the removal signal for $\boxed{t}$.
2. **Removal** reactions of the form $\boxed{\boxed{\beta}} + t^* \to \boxed{\boxed{\alpha}} + \boxed{t}$ where again $\boxed{\boxed{\beta}}$ and $\boxed{\boxed{\alpha}}$ are $\tau-$stable assemblies that differ by one copy of $\boxed{t} \in T$ and $t^* \in T^*$ is the removal signal for $\boxed{t}$. These reactions can only remove $\boxed{t}$ from $\boxed{\boxed{\beta}}$ if there is an instance of $\boxed{t}$ that is bound at exactly τ strength.

A CRN-TAM program $\mathcal{P}$ is initialized with nonnegative counts of each tile and signal species type, according to I. In an execution of $\mathcal{P}$, the reactions above occur in a stochastic sequence. The species or assemblies on the left-hand side of a reaction are the *reactants* and those on the right are the *products*. A reaction is *enabled* if all of its reactants are present in solution. The subsequent reaction to execute is always chosen randomly from the set of all enabled reactions. The likelihood of choosing a particular reaction is proportional to the product of its reactant counts, as with regular stochastic CRNs. If an execution reaches a state where no reactions are enabled, we say that it has *terminated*. Some CRN-TAM programs, like the discrete Sierpinski fractals in this work, do not terminate and continue indefinitely. For more information on the kinetics of the CRN-TAM model, see [23].

The CRN-TAM distinguishes between free tiles in solution and tiles that are part of activated assemblies. Free tiles can bond to assemblies, but two free tiles cannot bond together. All tiles come into being as free tiles, including those in the initialization; immediately after initialization, then, only signal, creation, deletion, and relabeling reactions are possible. We refer to these reactions as the *CRN component* of the CRN-TAM program. The CRN component usually serves to coordinate activation, deactivation, addition, and removal reactions and guide tile assembly growth.

In most CRN-TAM constructions, the CRN component is engineered to execute at least one activation reaction, which creates a new tile assembly so tiles can be added. Tiles created with creation reactions (or present in solution from the start) can then bond via their addition reactions, and potentially later be removed via their removal reactions. As discussed above, a tile can bond at any site on an activated assembly where it would interact with strength at least τ; tiles are subject to removal reactions when their interaction strength does not exceed τ.

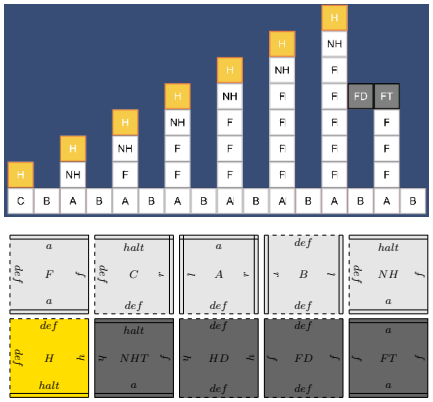
Note that if tile $\boxed{t}$ has a removal signal t^*, then adding $\boxed{t}$ releases t^*, and removing $\boxed{t}$ requires and consumes t^*. This allows the CRN component to interact more precisely with the addition and removal reactions. Some constructions also employ the deactivation reaction to eliminate existing (singleton) assemblies; unlike in the aTAM, the number of concurrent assemblies can increase or decrease over time.

3 The ALCH Programming Language

We present an overview of the features of the ALCH language and its implementation. ALCH is an imperative language with provisions specific to the CRN-TAM model such as the **add**, **remove**, **activate**, and **deactivate** statements which all take a tile type as a parameter and execute the corresponding tile actions. ALCH provides high-level features such as conditions, loops, and variable declaration and assignment. To guarantee the proper sequential execution of the code, special *line number species* are used to track progress through the ALCH program. At this time, ALCH only supports global variables and three datatypes: **bool**, **BondLabel**, and **Tile**. Additionally, ALCH includes a toolset for functionally specifying tilesets. Variables of type **bool** may be reassigned throughout the computation, but all **BondLabel** and **Tile** variables are immutable and final. One unique feature of ALCH is the **branch** statement, which nondeterministically chooses and executes multiple independent code blocks of tile addition and removal statements until one block finishes execution. Effects from uncompleted blocks are reversed, so only the code from the completed block remains. The **branch** statement also returns a **bool** associated with the block that finished successfully. Using **branch**, it is possible to query the state of tile assemblies without permanently attaching tiles to them. Each block in a **branch** statement is implemented as a reversible random walk. As an optimization, blocks can be given different weights to make them more likely to be chosen at the nondeterministic branch point. ALCH also supports multithreading via the **parallel** construct. We have designed ALCH to target sequential and multithreaded CRN-TAM constructions specifically; though it can describe a wide variety of constructions, ALCH is not designed to represent all possible CRN-TAM programs, and may not be suitable for constructions that leverage molecular parallelism in less straightforward ways.

We developed a software compiler in C# that compiles ALCH programs into CRN-TAM programs. We also developed a simulator for the CRN-TAM that includes the following two extensions to the model which are used only for optimization purposes: (1) it supports reactions with arbitrary arity, relaxing the CRN-TAM requirement that reactions are at most bimolecular; (2) it allows any reaction to add, remove, or activate a tile as a side effect and removes the requirement for the specific per-tile add and remove actions. Note that the output of the ALCH compiler is strictly compliant with the original CRN-TAM as specified in [22].

To demonstrate the expressiveness of ALCH, we will show that the CRN-TAM can strictly self-assemble an infinite shape at temperature 2 that the aTAM cannot. Consider an infinite staircase, visualized in Fig. 1, where for each $k \in \mathbb{N}$, the $(2k)$th column is $2+k$ tiles tall and the $(2k+1)$th column is one tile tall. The gaps between steps (even-numbered columns) prevent an aTAM program from directly transferring information about the height of one step to the next. Consequently,



```
bool at_top;
activate C;
add H; add B; add A;
while (true) {
  at_top = branch {
    true()  { add NHT; add HD; }
    false() { add FT;  add FD; }
  };

  if (at_top) {
    remove HD; remove NHT;
    add NH; add H;
    add B; add A;
  } else {
    remove FD; remove FT;
    add F;
  }
}
```

Fig. 1: An ALCH simulation of the infinite staircase is shown in the upper left. ALCH code for the staircase is shown on the right-hand side. The definitions of the tile types are not shown but are provided visually with bond labels and strengths in the lower left. On the right-most column of the simulation, the $\boxed{FT}$ and $\boxed{FD}$ tiles probe the previous column to detect which tile should be placed. These probe tiles are temporary and are eventually removed. The chemical reactions that ALCH outputs are shown in Figure 2. Note that the temperature τ of the CRN-TAM program is 2.

all information about the height of steps must be passed along the base of the assembly; an infinite tileset is required. However, the CRN-TAM can build and remove probe tiles that allow the assembly to query the previous column. We take advantage of this and show that the CRN-TAM can self-assemble this infinite shape, as shown in Fig. 1. Note that we omit the tile and bond declarations but include a graphical representation of the tile species used in the construction. See Figure 2 for the resulting list of reactions that ALCH outputs (these reactions are easier to interpret after reading the language implementation, described in detail in this section).

Intuitively, the self-assembly of the infinite staircase is implemented with a single infinite loop that repeatedly adds tiles to the assembly. Each execution of the loop begins by probing the previous column using the **branch** statement, which nondeterministically attempts to add the sequence of tiles $\boxed{FT}$ and $\boxed{FD}$ or the sequence of tiles $\boxed{NHT}$ and $\boxed{HD}$. If the latter succeeds, the variable at_top is set to **true**, and if the former succeeds, the variable is set to **false**. Notice that the **true**() branch will succeed if and only if the current column is the same height as the previous column because of the top tile $\boxed{H}$. The variable at_top is then used to either (a) finish the current column and initialize the next column or (b) add a single filler tile $\boxed{F}$ and continue with the current column. Using **branch** to query local structural information during the assembly is powerful; we employ a similar technique to show that the discrete Sierpinski triangle and discrete Sierpinski carpet can be strictly self-assembled in the CRN-TAM.

Initialization

- (S_t, S_f) with counts $(1, 0)$
- (TOP_t, TOP_f) with counts $(1, 0)$
- (R_t, R_f) with counts $(1, 0)$
- X_i with count 1 if $i = 0$, otherwise count 0
- all tiles with count 0

activate C;

$$X_0 \to X_1 + X_3$$
$$X_1 \to X_2 + \boxed{C}$$
$$X_2 + \boxed{C} \to \boxed{\boxed{C}} + C^*$$
$$X_3 + C^* \to X_4$$

add H; **add** B; **add** A;

$$X_4 \to X_5 + \boxed{H}$$
$$X_5 + H^* \to X_6$$
$$X_6 \to \cdots \to X_{10}$$

while(**true**)

$$X_{10} + S_f \to X_{\text{end}} + S_f$$
$$X_{10} + S_t \to X_{11} + S_t$$

true(){**add** NHT; **add** HD;}

$$X_{11} \leftrightarrow X_{12}$$
$$X_{12} \leftrightarrow X_{13} + \boxed{NHT}$$
$$X_{13} + NHT^* \leftrightarrow X_{14}$$
$$X_{14} \leftrightarrow X_{15} + \boxed{HD}$$
$$X_{15} + HD^* \leftrightarrow X_{16}$$
$$X_{16} + R_f \to X_{22} + R_t$$
$$X_{16} + R_t \to X_{22} + R_t$$

false(){**add** FT; **add** FD;}

$$X_{11} \leftrightarrow X_{17}$$
$$X_{17} \leftrightarrow X_{18} + \boxed{FT}$$
$$X_{18} + FT^* \leftrightarrow X_{19}$$
$$X_{19} \leftrightarrow X_{20} + \boxed{FD}$$
$$X_{20} + FD^* \leftrightarrow X_{21}$$
$$X_{21} + R_f \to X_{22} + R_f$$
$$X_{21} + R_t \to X_{22} + R_f$$

Assign to at_top

$$X_{22} + R_t \to X_{23} + R_t$$
$$X_{23} + TOP_f \to X_{25} + TOP_t$$
$$X_{23} + TOP_t \to X_{25} + TOP_t$$
$$X_{22} + R_f \to X_{24} + R_f$$
$$X_{24} + TOP_f \to X_{25} + TOP_f$$
$$X_{24} + TOP_t \to X_{25} + TOP_f$$

if(at_top)

$$X_{25} + TOP_t \to X_{26} + TOP_t$$
$$X_{25} + TOP_f \to X_{38} + TOP_f$$

remove HD; **remove** NHT; **add** NH; **add** H; **add** B; **add** A;

$$X_{26} \to X_{27} + HD^*$$
$$X_{27} + \boxed{HD} \to X_{28}$$
$$X_{28} \to \cdots \to X_{10}$$

remove FD; **remove** FT; **add** F;

$$X_{38} \to \cdots \to X_{10}$$

Fig. 2: Reactions for the staircase program in Figure 1. (S_t, S_f) is a constant Boolean value, and (R_t, R_f) holds the return value of **branch**. Note that the **branch** command has no specific reaction set. We can see its effects in the X_{11} species, which, when present, nondeterministically triggers the first reaction in either the true or the false branch. We have elided several **add** and **remove** reaction sets to save space. Finally, note that there are a number of unnecessary intermediate species like X_4. These arise because of ALCH's modular design, which conveniently allows us to describe the reaction sets for different commands independently from each other. It may be profitable to consider an optimizing ALCH compiler that reduces inefficiencies like this.

We now define each of the language features of the ALCH programming language and explain how they are implemented in the ALCH compiler. We begin by discussing how variables are implemented and define some useful notation that we use to specify what reactions and species are created for each language construct.

The ALCH compiler processes all variable declarations at compile-time. All **BondLabel** and **Tile** variables are added to a symbol table for later reference in **add**, **remove**, **activate**, and **deactivate** statements. Since **BondLabel** and **Tile** variables are immutable and cannot be reassigned, this simple treatment is sufficient. **bool** variables are implemented using two chemical species that are created at compile-time, and we commonly refer to them as *Boolean flags*. A Boolean flag x represents two chemical species $(x, \overline{x})$, where at any given time one of x and $\overline{x}$ has population 0 and the other has population 1. Unlike **BondLabel** and **Tile** variables, **bool** variables are mutable and can be reassigned by switching which species has population 1.

Most ALCH statements are implemented with a set of reactions, and each of their corresponding reactions includes its *line number species* as a reactant. When two statements are executed in sequence, the first statement emits the corresponding line number species of the second when it is finished. This allows the sequential execution of statements and avoids race conditions during the program execution. For statements that return a **bool**, the compiler creates a dedicated Boolean flag $(x, \overline{x})$ (or, in some cases, links an existing flag) for that line of code and guarantees that when the statement is executed, the associated flag contains the correct value.

When defining how each syntactical element of ALCH is implemented, it is convenient to use notation such as <block> to denote compound ALCH statements and expressions. For example, in the ALCH program in Fig. 1, the **if** statement and surrounding code can be written abstractly as:

```
<block1>
if (<block2>) {
    <block3>
}
<block4>
```

Notice how each <block> represents a sequence of statements. Here <block1> must emit the appropriate line number species for the conditional, and similarly, the **if** statement must emit the appropriate line number species for <block4> when it is finished. Since most of these language constructs are implemented with chemical species and reactions, the following notation is convenient:

$$X_{\text{start}} \to \text{<block>} \to X_{\text{end}} \tag{1}$$

Intuitively this notation means that if the line number species X_{start} is produced, then all the statements corresponding to <block> will be executed. The line number species X_{end} will be produced afterward. It is important to note that <block> abstractly represents a *sequence* of ALCH instructions, which may themselves use many intermediate line number species. Since some statements return a Boolean flag, we also use $T_{\text{<block>}}$ and $F_{\text{<block>}}$ to denote the true and false species of the returned Boolean flag after <block> is executed.

3.1 Boolean Expressions and Variable Assignment

We now discuss how Boolean expressions such as (val1 && val2) || !val3 are evaluated as well as Boolean assignment statements such as **bool** a = <block>. We begin with the logical operations of negation, conjunction, and disjunction.

Given an abstract Boolean expression represented by <block>, we consider the implementation of the logical negation !<block>. Recall that, at compile-time, <block> is given a dual-rail Boolean flag $(x, \overline{x})$. To implement negation, we simply need to return the negated flag $(\overline{x}, x)$. We handle this at compile-time when we link the ! syntax element with the flag of its child element <block>. Intuitively, the compiler will "cross the wires" of <block>'s Boolean flag when it encounters !<block> so that its output flag is negated. Thus negation does not introduce any new species or reactions but rather modifies the output of <block> directly at compile-time so that $T_{\text{<block>}}$ and $F_{\text{!<block>}}$ are the same species and $F_{\text{<block>}}$ and $T_{\text{!<block>}}$ are the same species.

To process a conjunction of logical expressions, we evaluate each expression from left to right and immediately return a false Boolean flag if an expression is false. Only when all expressions have evaluated to true will a true Boolean flag be returned. Below is how the statement <exp1> && <exp2> is implemented:

$$X_{\text{start}} \to \text{<exp1>} \to X_1 \tag{2}$$
$$X_1 + T_{\text{<exp1>}} \to X_2 + T_{\text{<exp1>}} \tag{3}$$
$$X_1 + F_{\text{<exp1>}} \to X_f + F_{\text{<exp1>}} \tag{4}$$
$$X_2 \to \text{<exp2>} \to X_3 \tag{5}$$
$$X_3 + T_{\text{<exp2>}} \to X_t + T_{\text{<exp2>}} \tag{6}$$
$$X_3 + F_{\text{<exp2>}} \to X_f + F_{\text{<exp2>}} \tag{7}$$

Notice how <exp1> is evaluated first, which emits the line number species X_1. The line number species together with the species $T_{\text{<exp1>}}$ and $F_{\text{<exp1>}}$ are used to determine whether the expression should immediately return false by producing the X_f line number species or continue by producing X_2 to start evaluating <exp2>. This process continues until one expression evaluates to false, or all expressions are true, and the X_t line number species is produced. A dedicated Boolean flag for the conditional is needed for output because the compiler cannot identify any preexisting child element that is guaranteed to hold the correct return value after execution. This Boolean flag is added to the CRN at compile-time, along with the following reactions to update the flag according to whichever X_t or X_f line number species is produced:

$$X_t + T_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{8}$$
$$X_t + F_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{9}$$
$$X_f + T_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{10}$$
$$X_f + F_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{11}$$

Here the species T_{result} and F_{result} correspond to the unique Boolean flag generated for this conjunction statement, and X_{end} is the line number species that initiates the block immediately following the conjunction. We implement logical disjunction

in a very similar way: the first time an expression returns true, we immediately return true; if all expressions return false, we return false.

We now describe how Boolean assignment statements such as a = <block> are implemented. To execute this command, we evaluate the right-hand side of the assignment. As discussed above, <block> has an associated Boolean return flag; when <block> finishes execution, this flag is guaranteed to hold the correct return value. We then use the flag species as catalysts to direct execution to the lines of code that set the variable a to true or to false accordingly. Below are the reactions that implement the assignment a = <block>:

$$X_{\text{start}} \to \text{<block>} \to X_1 \tag{12}$$
$$X_1 + T_{\text{<block>}} \to X_t + T_{\text{<block>}} \tag{13}$$
$$X_1 + F_{\text{<block>}} \to X_f + F_{\text{<block>}} \tag{14}$$

The line number species X_t and X_f encode the Boolean return value of <block>, and the following four reactions copy this result into the global Boolean flag for the variable a:

$$X_t + T_a \to X_{\text{end}} + T_a \tag{15}$$
$$X_t + F_a \to X_{\text{end}} + T_a \tag{16}$$
$$X_f + T_a \to X_{\text{end}} + F_a \tag{17}$$
$$X_f + F_a \to X_{\text{end}} + F_a \tag{18}$$

Here T_a and F_a are the species representing the global Boolean flag associated with the variable a. Since we do not know whether a is true or false at compile-time, we must account for both possibilities. Note that we use the <block> Boolean flag species only as catalysts, so the dual-railed representation is preserved.

Since the CRN-TAM requires all reactions to be at most bimolecular, we can use at most one non-line-species product and one non-line-species reactant per reaction. To process information, we must often split computations across several reactions and pass information down in the line number species. Above, for example, the intermediate line number species X_t and X_f serve to temporarily store the return value so we can process it in the following reactions. This and similar patterns frequently occur throughout our implementation of ALCH.

3.2 Conditionals and Loops

ALCH also supports conditional execution and is implemented similarly to the previous constructions, as shown below. We also support **else** blocks by modifying reaction (21) to output an X_f molecule and adding an additional reaction $X_f \to X_2$ where X_2 is the line number species for the **else** block. ALCH also supports **while** loops which are implemented similarly but alternates between the line number for <exp> and the internal <block>.

```
if (<exp>) {
    <block>
}
```

$$X_{\text{start}} \to \text{<exp>} \to X_1 \tag{19}$$
$$X_1 + T_{\text{<exp>}} \to X_t + T_{\text{<exp>}} \tag{20}$$
$$X_1 + F_{\text{<exp>}} \to X_{\text{end}} + F_{\text{<exp>}} \tag{21}$$
$$X_t \to \text{<block>} \to X_{\text{end}} \tag{22}$$

3.3 Tile Addition, Removal, Activation, and Deactivation

In order to enforce a controlled, sequential execution of commands, our tile instructions must pause program execution until they succeed and have cleaned up after themselves. When ALCH executes an **add** command, for example, we must not proceed to the next command in sequence until the tile is added, and we have cleaned up any excess species involved in that addition. This ensures that no undesired tile or signal species are present in solution, and execution can proceed predictably.

Recall that in the CRN-TAM, every tile species $\boxed{A}$ is associated with at most 1 tile removal signal A^*, and the following two sets of reactions.

$$\boxed{\boxed{\alpha}} + \boxed{A} \rightarrow \boxed{\boxed{\beta}} + A^* \tag{23}$$

$$\boxed{\boxed{\beta}} + A^* \rightarrow \boxed{\boxed{\alpha}} + \boxed{A} \tag{24}$$

Assemblies $\boxed{\boxed{\alpha}}$ and $\boxed{\boxed{\beta}}$ differ only by one instance of $\boxed{A}$, placed in $\boxed{\boxed{\beta}}$. We are given the option to have tiles with no removal signals in the CRN-TAM, but ALCH gives each tile type a unique removal signal. Therefore, we can add a tile by placing it in solution and relying on the first reaction above to attach it to the assembly. We then wait to proceed until we can clean up the tile removal signal that the new tile releases when it bonds to an assembly. The implementation of **add** tileA is as follows where X_{start} is the line number species of the **add** statement and X_{end} is the line number species of the statement that immediately follows.

$$X_{\text{start}} \rightarrow X_1 + \boxed{A} \tag{25}$$

$$X_1 + A^* \rightarrow X_{\text{end}} \tag{26}$$

Note that we cannot produce X_{end} until we have both added $\boxed{A}$ and cleaned up its removal signal A^*. This ensures that **add** pauses program execution until it succeeds and cleans up after itself, as discussed at the start of this section.

The implementation of **remove** tileA is similar, but it relies on the existence of Reaction (24) discussed earlier:

$$X_{\text{start}} \rightarrow X_1 + A^* \tag{27}$$

$$X_1 + \boxed{A} \rightarrow X_{\text{end}} \tag{28}$$

Assembly activation is more difficult. The CRN-TAM allows only activation reactions of the form: $X + \boxed{A} \rightarrow \boxed{\boxed{A}} + A^*$. There are two difficulties here. First, it is challenging to guarantee that $\boxed{A}$ is activated as a new assembly instead of being added to a preexisting assembly. In order for an activation reaction for $\boxed{A}$ to proceed, we must already have $\boxed{A}$ in solution; if $\boxed{A}$ is in solution, we cannot prevent it from bonding to an existing compatible site. Instead of guaranteeing this explicitly, we rely on users of ALCH to prevent these situations. The second difficulty is that tile activation reactions cannot output a line number species, so we have no easy way of passing execution to the next reaction in our desired

sequence. We handle this issue by producing the desired line number species in advance, as shown in the implementation of **activate** tileA below.

$$X_{\text{start}} \rightarrow X_1 + X_3 \tag{29}$$

$$X_1 \rightarrow X_2 + \boxed{A} \tag{30}$$

$$X_2 + \boxed{A} \rightarrow \boxed{\boxed{\alpha}} + A^* \tag{31}$$

$$X_3 + A^* \rightarrow X_{\text{end}} \tag{32}$$

Although the line number species X_3 is present initially, the last reaction cannot execute until the end, when A^* is also present.

We straightforwardly implement tile deactivation, subject to similar constraints. Instead of temporarily having two line number species in solution, we temporarily have none as we wait for the deactivation reaction to return one.

3.4 Nondeterministic Branch Construct

We allow nondeterminism in our language through the **branch** construct. A branch statement contains multiple branch paths; a branch path is a sequence of tile addition and removal instructions collectively associated with a Boolean value. At the start of a branch statement, a program nondeterministically chooses one of the branch paths and begins executing it. Broadly speaking, **branch** returns the Boolean value of the path that ultimately finishes successfully. Each path contains only reversible commands, so if one path is impossible to complete, execution will ultimately reverse out of it and proceed down a different path. Since we require branch paths to be reversible, we allow only **add** and **remove** commands inside **branch** paths. It is straightforward to modify the **add** and **remove** reaction sets to be reversible; we discuss this below. It is possible to support additional commands by making other language constructs reversible, but for our purposes here, **add** and **remove** statements are sufficient.

It is important to note that our notion of reversibility is not complete. For example, suppose we execute **add** tileA inside a branch path. If this statement is reversed, the system will attempt to remove the tile $\boxed{A}$. However, if there are multiple instances of $\boxed{A}$ bonded to the assembly, it is not guaranteed to remove the same tile added earlier in the branch. Additionally, if we add a tile at a strength greater than τ, we will not be able to remove it when attempting to reverse the addition. Any ALCH programmer should exercise caution when using the **branch** statement to avoid such side effects.

The **branch** statement is implemented with a single branch point that can lead to any one of the branch paths, as shown in Fig. 3. From that branch point, we execute only one branch path at a time. Since each branch path is reversible, if execution proceeds down a branch that is incapable of completing, it will eventually return to the branch point via random walk. When a branch finishes execution, we return the Boolean flag that corresponds with the path that completed.

Consider the following **branch** statement where <trueblock> and <falseblock> are arbitrary sequences of **add** and **remove** statements.

```
result = branch {
    true(2) {
        add tileA;
        remove tileB;
    }
    false(1) {
        remove tileC;
    }
};
```

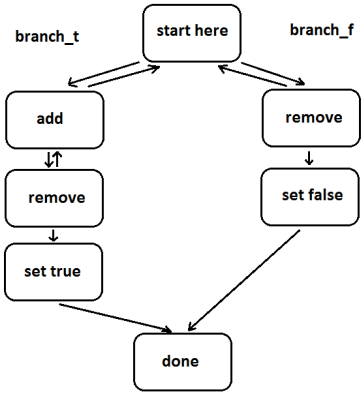


Fig. 3: Possible execution paths through a branch statement. Instructions associated with **true** and instructions associated with **false** are executed nondeterministically via a random walk. The **branch** statement terminates when one path runs to completion, and it returns the corresponding Boolean flag. The integers inside the parentheses of the **true** and **false** bias the random walk by that factor.

```
branch {
    true()  { <trueblock>  }
    false() { <falseblock> }
}
```

The above **branch** statement is implemented in ALCH with the chemical reactions:

$$X_{\text{start}} \leftrightarrow \text{<trueblock>} \rightarrow X_t \tag{33}$$

$$X_{\text{start}} \leftrightarrow \text{<falseblock>} \rightarrow X_f \tag{34}$$

A few things should be noted about the above implementation. First, both the <trueblock> and <falseblock> use the same line number species X_{start}. Second, those reactions are reversible, as indicated by the bidirectional arrows. Third, once one of the blocks finishes, it is completed with an irreversible reaction that terminates the **branch** statement. Fourth, the **add** and **remove** commands outside of **branch** are not reversible; inside branch paths, we modify each add and remove command to make them reversible. In our original description of tile actions in Subsection 3.3, we required tile commands like **add** to succeed and clean up after themselves before proceeding to the next ALCH statement. A reversible **add** command might not ever succeed, so we must update this requirement. Reversible **add** may return to the previous command without adding a tile; if it does so, we require it to clean up any excess species that it has created in the addition attempt. This guarantees that a reversible **add** command never leaves any extra species in solution, regardless of whether its branch path fails or succeeds.

The reversible implementation **add** statement consists of the two reversible reactions $X_1 \leftrightarrow \boxed{A}$ and $A^* \leftrightarrow X_2$. Note that execution cannot pass from X_1 to X_2 without adding $\boxed{A}$ and cleaning up the resulting A^*, and execution cannot reverse from X_2 to X_1 without detatching $\boxed{A}$ from the assembly and cleaning its species up from solution. A reversible **remove** statement is implemented similarly.

The last thing to note about the **branch** statement is that it returns a Boolean flag. Therefore a dedicated flag must be created at compile time and be appropriately set after the execution is completed. Therefore the following reactions are also needed to set this Boolean flag.

$$X_t + T_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{35}$$
$$X_t + F_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{36}$$
$$X_f + T_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{37}$$
$$X_f + F_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{38}$$

3.5 Multithreading in ALCH

Thus far, we have focused on the "single-threaded" component of ALCH, which allows us to quickly implement sequential algorithms one instruction at a time. The CRN-TAM, on the other hand, has at its disposal natural many-way chemical parallelism. We leverage this parallelism in ALCH via the **parallel** command, which allows multiple ALCH threads to execute simultaneously. Within the scope of the **parallel** command, users can specify an arbitrary number of threads as either **single** or **unbounded**. The **single** thread executes one block of code on a single thread. The **unbounded** command continually spawns threads, all of which execute the same block of code, until one of those threads calls the **finish** command to halt thread spawning. All threads of both types must complete fully before execution continues past **parallel**. (When an **unbounded** block completes its task and one thread calls **finish**, there may be a number of other threads from the same block that have not concluded and may become stuck. We allow the user to handle these; in most cases, we expect that these threads will terminate themselves via the same conditional block that checks for completion and calls **finish**.)

The **parallel** command spawns the specified single threads or unbounded collections of threads and then waits for all of them to complete. A **parallel** command can contain any combination of **single** and **unbounded** commands, as illustrated in the example below, where each <block> is an arbitrary block of code:

```
parallel {
  single    { <block1> }
  single    { <block2> }
  unbounded { <block3> }
}
```

When implementing a **parallel** command, then, we must trigger each of the **single** and **unbounded** commands that it contains. We will defer implementation of continuous thread spawning to our discussion of the implementation of **unbounded**, discussed below; the initial triggering mechanism in **parallel** is the same for **single** and **unbounded** commands, so the implementation details of **parallel** do not need to distinguish.

Let S_i be the first line number species in the ith component of a **parallel**, and let M_i be the line number species that the ith component produces after terminating. To initialize all the **single** and **unbounded** thread components, we sequentially add each S_i species into solution. We must then collect all M_i species so that we know each thread is completed. We again do so sequentially (using Y_i as line number

species for clarity). Once we have collected all M_i species, we pass execution to the next statement after **parallel** by creating a Y_f molecule. All of this is accomplished with the following reactions.

$$\begin{aligned} X_0 &\to X_1 + S_0 & Y_0 + M_0 &\to Y_1 \\ X_1 &\to X_2 + S_1 & Y_1 + M_1 &\to Y_2 \\ &\dots & &\dots \\ X_{n-1} &\to Y_0 + S_{n-1}. & Y_{n-1} + M_{n-1} &\to Y_f. \end{aligned}$$

Similar to other imperative languages that support multithreading, ALCH does not guarantee the absence of *cross talk* between threads. For example, simultaneous additions or removals of the same tile can cancel out and produce unexpected results. In general, programmers should exercise caution to prevent race conditions and errors caused by multiple threads interfering with one another. We provide a more detailed treatment of these issues in our discussion below on the correctness of ALCH's multithreading.

3.5.1 Single Threads and Unbounded Threads

Within a **parallel** block, a thread labeled **single** will be executed exactly once in parallel with the other threads created in the **parallel** block. We implement a **single** thread by generating reactions for that command's code block as normal, and link them with the appropriate S_i and M_i species as described above. Parallel **single** blocks can also be nested within each other.

An **unbounded** block will continuously spawn threads until one of them executes the **finish** command. At that point, **unbounded** block stops spawning threads and waits for all spawned threads to complete before signaling to the containing **parallel** block that it is done. The **unbounded** block can be used to quickly fill in large regions of an assembly with tiles at a rate comparable to the original aTAM. The difficulty in implementing an unbounded number of threads is that we can define only a bounded number of chemical species. Combined with the difficulty of absence detection in chemical systems, this makes it challenging to coordinate an unbounded number of threads and determine when all have finished. Fortunately, we can leverage the spatial structure capabilities of the CRN-TAM to store how many threads we have created and signal when all have finished.

At the beginning of our ALCH program, we activate one single-tile assembly for each **unbounded** command. We refer to the seed tile for **unbounded** command C as T_{0C}, and the assembly it forms as A_C. We define unique tile types for each **unbounded** command to avoid crosstalk.

When executing C, we first attach tile T_{1C} to T_{0C}. Tile T_{1C} denotes the boundary of our spatial storage region; when we encounter it later, we will know we know we have reached the end of the region. We can then spawn threads using a looping construction similar to the **while** loop implementation discussed above. The condition of this while loop is a single Boolean variable, which we will later set to false on executing the **finish** command. For each thread that we spawn, we add the tile T_{2C} onto our A_C assembly to form a growing line extending out from T_{1C}, as shown in Fig. 4.

Fig. 4: An assembly that stores the count of currently running threads spawned by an **unbounded** block. Each ALCH program generates an assembly like this for each **unbounded** command it contains. The five $\boxed{T_{2C}}$ tiles indicate that five threads are currently running in parallel. When each thread completes, it will remove its $\boxed{T_{2C}}$ tile. When all threads have completed, the $\boxed{T_{1C}}$ tile is removed, signaling the CRN that it is safe to continue.

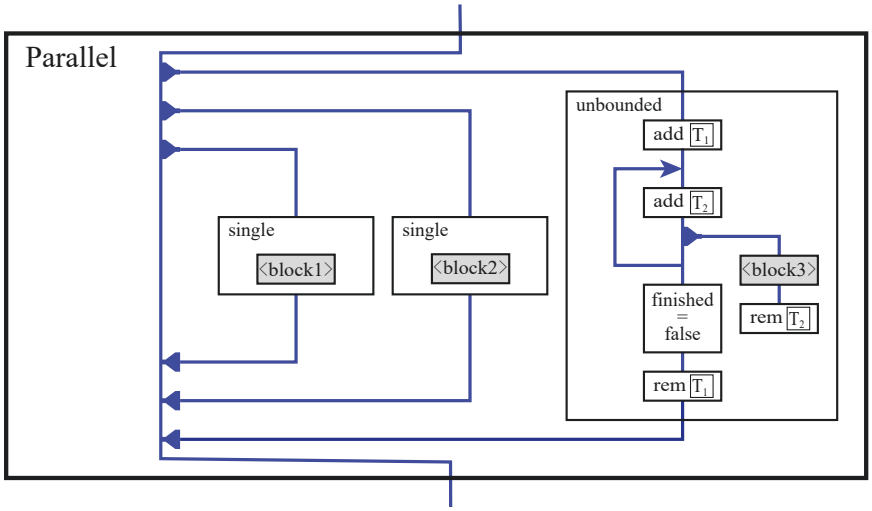


Fig. 5: A flowchart showing how control passes through an example `parallel` block. Note that the unbounded `<block3>` threads never merge back; ALCH relies on the $\boxed{T_2}$ counting mechanism to coordinate unbounded threads.

Recall that a thread stops executing when it reaches the end of its code block; when each thread completes, we remove a T_{2C} tile from A_C. The number of **unbounded** threads from command C that are currently executing in solution, then, is the same as the number of T_{2C} tiles in A_C. When one thread executes the **finish** command, we set the loop variable to false, and stop spawning new threads. We must then wait for all currently executing threads to finish and remove the remaining T_{2C} tiles. On the main, coordinating thread that spawned all the unbounded threads, then, we attempt to remove T_{1C}. This can only happen once all unbounded threads have completed. When we successfully remove T_{1C} and detect its signal in solution, we merge back into the containing **parallel** command.

The implementation of **unbounded** relies on constructs like while loops and tile additions. Since we have already described in detail how to create these in ALCH, we will present the implementation of **unbounded** as ALCH code. Inside the scope of a **parallel** block, the command **unbounded** { <block> } compiles into

```
add T_1C;
while(!finished) {
```

```
    <spawn thread for block>
    add T_2C;
  }
  remove T_1C;
  finished = false;
```

The implementation of <spawn thread for block> is straightforward; we need only create the first line number species S of the new thread with a reaction: $X_0 \rightarrow X_1 + S$. We must also generate reactions for the statements in <block>, with a remove command at the end:

```
<block>
remove T_2C;
```

Finally, when the **finish** command is reached, we simply set finished = **true**.

Note that this implementation prevents the use of nested **unbounded** commands, since we require a unique threading tiles for each **unbounded** block. However, **unbounded** blocks can be nested inside **single** blocks. Note also that any Boolean variables we use are global; without access to an unbounded number of chemical species, we cannot define thread-local Boolean flags using the dual-rail mechanism described earlier.

3.5.2 Correctness of Multithreading in ALCH

In our discussion of multithreading, we have thus far assumed that that we can create arbitrary threads by adding new line number species without worrying about the line number species interfering with each other. We will now discuss the correctness of ALCH's multithreading and present several caveats.

We first discuss correctness without taking tile actions into account, and then consider the impact of tile actions. To begin, we note that Boolean flag species are not a concern. All modifications to Boolean flag species are atomic: i.e., similar to $X_0 + F_t \rightarrow X_1 + X_1$, where both flag species are updated within a single reaction. (Flag species are all global, so threads can communicate with each other via Boolean variables; this is intended behavior. We cannot associate CRN flags with each of an unbounded number of threads, so it would be extremely challenging to implement thread-local variables without complex tile mechanisms.)

Threads spawned from **single** commands do not share line number species, and so present no risk of cross-talk. Threads spawned from the same **unbounded** commands do share line number species, but this also does not introduce any failure modes. If line number species X_a can only transform into line number species X_b or X_c, then it is guaranteed to do so regardless of the presence or absence of additional X_a in solution. (We choose a nondeterministic example for generality.) The thread is therefore guaranteed to proceed as intended from X_a to either X_b or X_c.

We now consider tile actions. All tile actions in ALCH are *transactional*; the CRN component produces a tile species and then consumes its removal signal, or vice versa. The tile assembly component facilitates the transaction, and can (in general) convert any tile species to its removal signal and vice versa. We do not distinguish between **add** and **activate** or between **remove** and **deactivate**; as discussed above, we rely on the user to take this into account manually.

Suppose we have two threads A and B; thread A attempts to send a tile to the assembly (by addition or activation), producing $\boxed{t}$ and later attempting to

consume removal signal t^*. We identify several ways in which B might potentially interfere with this process.

One possibility is that B might consume $\boxed{t}$ before it can bond to an assembly. In this case, B must already have produced a t^* molecule, which A can consume instead of the t^* which it was to receive from the assembly. Both threads can therefore complete their transactions. The total number of assembly tiles is unchanged; this is the intended affect on the assembly, as A sends a tile and B retrieves one. We can conceptualize this as the send and retrieval "cancelling each other out".

The difficulty, of course, is that no tiles have actually been exchanged with the assembly. This poses several issues.

- In a situation where no $\boxed{t}$ is available for removal on any assembly and no $\boxed{t}$ can be added anywhere, the ALCH commands to add and remove $\boxed{t}$ would ideally block forever. Under our implementation of multithreading, if addition and removal are both executed, then both can continue without blocking.
- Similarly, suppose $\boxed{t}$ would have bonded at strength greater than τ. In this case, B should block, A should continue without blocking, and $\boxed{t}$ should be stuck on the assembly. Under our implementation, the assembly is unchanged and B doesn't block.
- If A is executing an activation command, the activation line number species (X_2 in equation (32) above) is left over and can later convert an intended tile addition into an activation.
- If B is executing a deactivation command, the deactivation line number species (the equivalent of X_2 in reverse) never appears and B blocks forever.

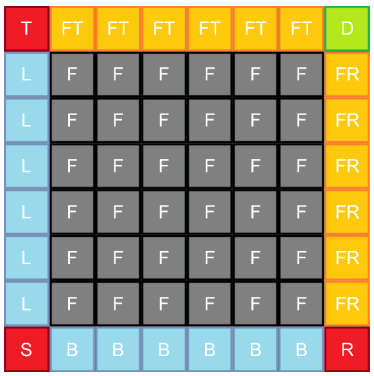
In practice, however, we do not expect that these possibilities will be an issue very often; we rely on the user to handle any issues.

The other possibilities for interference from B on A are similar; we omit the analysis and conclude that, with the caveats mentioned above, ALCH's multithreading system functions as intended.

3.5.3 A Multithreaded Construction of an 8×8 Square

To demonstrate multithreading in ALCH, we exhibit a fast construction of an 8×8 square. We can construct such a square sequentially in ALCH, but only slowly, one tile at a time. Since **add** commands are blocking in ALCH, any single-threaded construction must know the correct tile to add before adding it and proceeding to the next. Any single-threaded construction will therefore likely either be very complex or specify each tile position manually. Our multithreaded implementation avoids these pitfalls, and is shown below; due to the simplicity of the tile structure, we omit the tile listing.

We begin by activating a seed tile $\boxed{S}$ to serve as the lower left corner and initializing the variable not_done to false to track square completion. We then perform 7 tile additions to construct the south and west edges (terminating in special tiles $\boxed{T}$ and $\boxed{R}$, which serve as starting points for the east and north edges.) We use the **single** command to construct the south and west edges in parallel. Also in parallel with the edge construction, we use three unbounded regions to continually add $\boxed{F}$ to fill the square and $\boxed{FT}$ and $\boxed{FR}$ to construct the remaining edges. Each unbounded region adds the corresponding tile inside a while loop,




```
activate S;
bool not_done = true;
parallel{
  single{ add B; add B; add B; add B;
           add B; add B; add R; }
  single{ add L; add L; add L; add L;
           add L; add L; add T; }
  unbounded{
    while(not_done){
      branch {
       true(){add F;}
       true(){add FT;}
       true(){add FR;}
       true(){} }}
    finish; }
  single{ add D;
           not_done = false; } }
activate Q;
```

Fig. 6: This is the finished 8×8 square constructed by our multithreaded example code. Tile $\boxed{S}$ in the lower left is the seed tile; all tiles $\boxed{L}$, $\boxed{B}$, $\boxed{T}$, and $\boxed{R}$ are harcoded. Addition of $\boxed{D}$ in the upper right signals that the square is complete. This construction is at temperature $\tau = 2$ to enable cooperative bonding. All bonds between two tiles in the set $\{\boxed{T}, \boxed{L}, \boxed{S}, \boxed{B}, \boxed{R}\}$ are strength two (i.e., the red and blue tiles that make up the west and south edges). This allows us to build the west and south edges without additional support. All other bonds are single-strength to force cooperative bonding.

simulating an unlimited supply of each type. The **add** commands take place inside branch constructs with alternate empty paths; this prevents threads from being stuck inside branch statements after no more tiles can be added.

Tile $\boxed{D}$ can only be placed in the north-east corner of the square, bonded to $\boxed{FT}$ and $\boxed{TR}$; see Fig. 6 for an image of the completed square. We add a final **single** thread in parallel, which attempts to add $\boxed{D}$; when it succeeds, it toggles not_done, ending the while loops in the unbounded threads. With the while loops ended, threads from each **unbounded** region reach the **finish** command, preventing the **unbounded** regions from spawning more threads. Once all threads clean themselves up, the parallel command ends, and our construction activates a final $\boxed{\boxed{Q}}$ assembly to signal that the construction is complete.

Recall from Subsection 3.4 that **add** commands inside **branch** are reversible, and clean up after themselves. We therefore need not worry about excess $\boxed{F}$, $\boxed{FT}$, or $\boxed{FR}$ remaining in solution after execution. Every time any thread begins the command **add** F, for example, that thread will either complete the command or reverse back past it; in either case, ALCH cleans up all excess species, as discussed in Subsection 3.4.

This program leverages the chemical parallelism of the aTAM to speed construction, but is still fully modular: it knows when it has finished, and cleans up all parts of itself to set the stage for any subsequent ALCH commands.

In the general case, where we construct an $n \times n$ square using this method, we would expect an asymptotic speedup (as compared to any single-threaded ALCH construction, all of which must perform $O(n^2)$ tile additions sequentially on the same thread.). We omit a detailed analysis of the 8×8 square runtime, however, in favor of an intuitive argument. We construct the west and south edges each in $O(n)$ sequential **add** instructions. The bulk of the remaining **add** instructions are dedicated to filling in the $O(n^2)$ inner spaces in the triangle. With a linear number of threads present, each calling add repeatedly, ALCH can place $O(t^2)$ number of tiles in t time.

Even so, a quadratic speedup is minimal compared to the potential many-way parallelism that the CRN-TAM is capable of. Schiefer and Winfree[23] suggest a "tree counter" program for producing large species counts. This pattern uses a hard-coded set of reactions of the form $S_i \rightarrow S_{i+1} + S_{i+1}$ to produce exponential species counts in linear time. ALCH does not support this exponential technique; it may be profitable to leverage this pattern to greatly accelerate thread spawning in ALCH **parallel** sections. If users specify a tree counter depth at compile time, ALCH has enough information to generate an exponential number of **single** thread copies or **unbounded** thread counters in linear time. We believe that with this enhancement, ALCH could emulate the efficient copy-tolerant language decider from [23]; we are unsure about the nondeterministic decider from that work, but suspect that ALCH would require further development to support it.

3.6 Functional Tileset Specification

We have demonstrated a variety of ALCH features designed to handle program logic on the CRN side; we will now discuss a feature that is designed to expedite tile set design. A common tendency in aTAM and CRN-TAM constructions is to rely on sets of tiles that are duplicated with minor changes. In our discrete Sierpinski triangle construction, for example, we create separate tile species both for odd and even rows to prevent crosstalk and for the two symmetric halves (split along the diagonal) of the triangle. This gives us four sets of tiles, each of which is very similar to the others. Ideally, we would abstract out the similarities between these four sets to avoid tedious work and potential copy-paste-modify errors.

In ALCH, we use the TileSet system to avoid this type of code duplication. We can think of a TileSet as a function that accepts bond labels as arguments and returns a set of tiles. A TileSet contains a list of tiles, with bond labels that can be TileSet arguments or previously defined labels. TileSets can also include other TileSets as components, and can pass their arguments down in the expected way. TileSets can nest arbitrarily deep, but we require a TileSet to be defined before it is referenced, so reference loops are impossible.

We also include the modifiers transpose, flipx , and flipy , which perform the corresponding transformations on component TileSets. When constructing the symmetric halves of the discrete Sierpinski triangle, for example, it is convenient to use transpose to generate one half from the other. The **register** command supplies a TileSet with arguments and adds all resulting tiles to the ALCH program.

We must also specify a naming scheme so that we can reference TileSet tiles later in the program. As part of every TileSet invocation, whether for a component of another TileSet or via the **register** command, the user must provide an

```
BondLabel a;
BondLabel b;
BondLabel c;
BondLabel d;

TileSet Set0(x, y) {
  Tile t0(x, 2, a, 0, a, 2, a, 0, "Ti", "red");
  Tile t1(b, 0, y, 2, b, 0, b, 2, "Tj", "blue");
}

TileSet Set1(z) {
  Tile t2(a, 2, a, 0, a, 0, a, 0, "Tk", "gray");

  Set0 set(c, z);
  transpose(Set0 set_t(d, z));
}

register Set1 tiles(d);
```

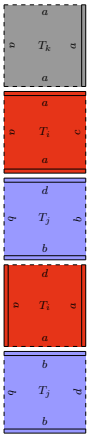


Fig. 7: The five tiles generated by the presented code sample.

invocation name. The user is also responsible for giving each component tile in each TileSet a name. (The TileSet system is functional in nature, but this naming scheme is reminiscent of the practice of naming class members in object oriented languages.) For each generated tile, then, ALCH concatenates all the relevant tile invocation names, delimited with the underscore character, and appends the component name of the tile itself.

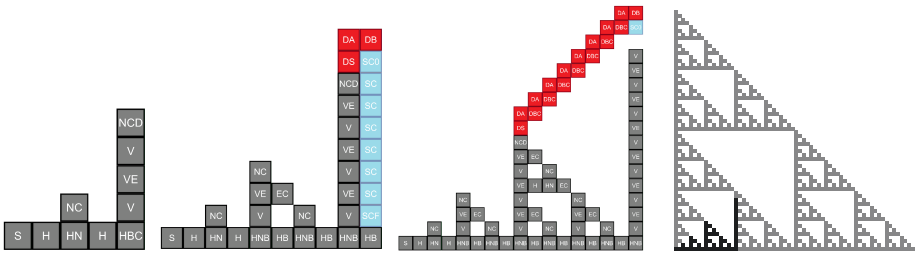
We illustrate this scheme with code sample from Fig. 7. The five tiles generated by this code are named: tiles_t2 , tiles_set_t0 , tiles_set_t1 , tiles_set_t_t0 , and tiles_set_t_t1 .

4 Strict Self-Assembly of the Discrete Sierpinski Triangle

We now present the CRN-TAM construction that strictly self-assembles the discrete Sierpinski triangle (DST) using ALCH. To see the complete specification of the construction in ALCH, along with a video visualization of the self-assembly, see `http://web.cs.iastate.edu/~lamp/`.

Before we give the details of the construction, we will define what we mean by self-assembling a shape. We say that a CRN-TAM system N assembles a shape $S \subseteq \mathbb{Z}^2$ if the following is true: for each position $p = (x, y) \in \mathbb{Z}^2$, there exists a time t in every fair execution of N such that at all times $t' > t$, position p is filled if $p \in S$ and empty otherwise. Intuitively a fair execution is such that any reaction which is enabled infinitely often must occur at some finite time t.

We begin with an overview of tile types and a brief description of their purpose and then describe the DST construction algorithm in detail. Since the DST is symmetric about the line $f(x) = x$, we refer to the two symmetric halves as the lower symmetric triangle (LST) and the upper symmetric triangle (UST). We first discuss the techniques to strictly self-assemble the LST, which can be easily modified to construct the UST in parallel. In our construction, it is useful to distinguish between three types of tiles: (1) structural tiles, (2) scaffold tiles, and



(a) Seed (b) Vertical scaffold (c) Diagonal scaffold (d) Sierpinski triangle

Fig. 8: (a) The initial hard-coded structure upon which we build the lower half of the DST. (In the final program that constructs the whole DST, this structure has a symmetric upper half.) (b) Demonstrates how our construction extends a vertical scaffold down to occlude all the potential tile bond sites on column currently being constructed. (c) Shows the diagonal scaffold before erasing itself and starting a new diagonal scaffold. (d) Shows a section of the Sierpinski triangle that includes the lower and upper symmetric halves; the part corresponding to (c) is highlighted.

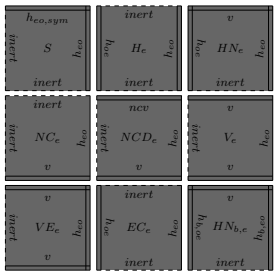
(3) probe tiles. Structural tiles are permanent and form the DST itself. Scaffold tiles are used to construct temporary auxiliary structures to facilitate the DST construction. Probe tiles are rapidly added and removed to query existing information of previously placed structural tiles. To avoid unwanted crosstalk between the symmetric halves, we duplicate the set of structure tiles into a symmetric group with bonds that are incompatible with the LST tiles. We also differentiate the tile types of even and odd columns to prevent a partially constructed column from interfering with the construction.

We now discuss the construction for the strict self-assembly of the DST. The first step in our construction unpacks the initial structure shown in Fig. 8a with hard-coded tile activation and addition statements. This is easily accomplished by adding tiles in a specific order that avoids ambiguity in placement. After the initial structure tiles are placed, we then construct the LST column by column, adding structure tiles one-at-a-time, completing each column before proceeding to the next. We also use a variable to track whether we are currently constructing an even or odd column. The process of adding one structure tile at a time is akin to a dot-matrix printer, placing dots of ink one line at a time.

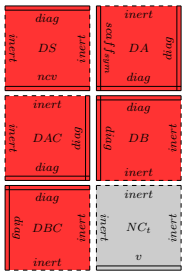
We present a list of even LST tiles in Figure 9a. Note that most structure tiles have an h_{eo} bond of strength at least one on their eastern edges so that probe tiles can attach cooperatively.

4.1 Scaffold Construction

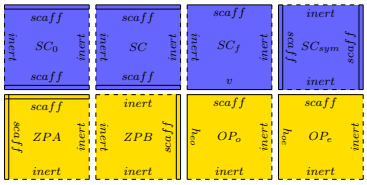
We construct two types of scaffolds. The diagonal scaffold, shown in red in Fig. 8, runs along the diagonal of the DST and provides an anchor for the vertical scaffold, which is shown in cyan. The vertical scaffold covers up potential bond sites that we do not wish to bond to, as illustrated in Fig. 8b. The diagonal scaffold is straightforward to construct; before constructing each column, we extend it out by two more tiles. For the vertical scaffold, we must extend it only as far as the base



(a) These are the structure tiles that form the odd columns of the LST; we omit the even column tiles and the tiles for the entire UST, which are very similar.



(b) These tiles form the scaffolding that runs along regions of the southwest-to-northeast diagonal.



(c) The blue tiles form the vertical scaffolding that obscures bond sites to facilitate adding tiles at specific locations. The yellow tiles form the probes that determine whether a position in the previous column is filled or empty.

Fig. 9: Tiles types used in the DST construction.

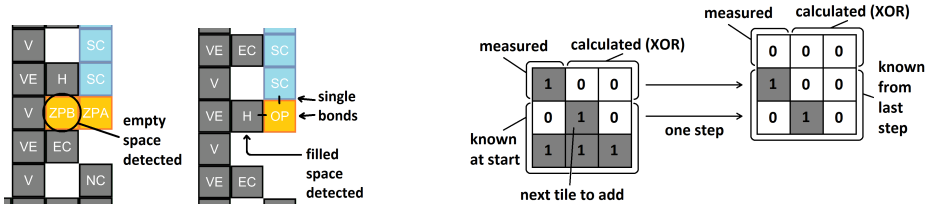
of the DST. We extend the DST base row out by one space to denote the bottom of the vertical scaffold. We begin the vertical scaffold with $\boxed{SC_0}$ and construct most of it from vertically double-bonded $\boxed{SC}$ tiles. We use $\boxed{SC_0}$ so that we know when we are done when removing the scaffold.

The special final tile $\boxed{SC_f}$ has a single bond on its north and south edges; it cannot attach until it can bond cooperatively with the base tile below it and the scaffold tile above it. When our system succeeds at placing $\boxed{SC_f}$, it knows to continue to the next phase. We allow the assembly to remove $\boxed{SC}$ as well, in case $\boxed{SC}$ bonds at the bottom instead of $\boxed{SC_f}$; scaffold construction proceeds as a random walk, which we bias with reaction rates.

Since the diagonal scaffold is not part of the DST, we must periodically clean it up. Some columns in the LST are entirely solid up to the diagonal; when we encounter one of these, we destroy the existing diagonal and begin a new diagonal starting from the top of the solid column. As with $\boxed{SC_0}$, we start with a special diagonal tile so that we can remove the diagonal in a loop and know when to stop.

4.2 Adding Structure Tiles with the Probe

When beginning to place tiles on a new column i, the vertical scaffold must be completely initialized as in Fig. 8b. We must know which tile, if any to add to the DST at each vertical position: T-joint, straight connector, etc. To that end, after constructing the vertical scaffold, we initialize a 3×3 Boolean grid, centered on $(i, 1)$, of Boolean flag variables. This grid stores whether those tile positions are occupied in the full DST; note that if we know the 3×3 grid around a position, we know which tile, if any, goes there. The lower six squares are entirely determined by whether i is even or odd; the lowest row of the LST is solid, and the second-lowest alternates every space between filled and empty. To determine the upper-left space, we use the "probe" to measure whether $(i-1, 2)$ is filled or empty in column $i-1$,



(a) Illustrates how the probe detects empty spaces in the Sierpinski triangle; both paths are done in parallel.

(b) Illustrates how the next 3×3 window around the probe is updated using the previous window and the tile detected by the probe.

Fig. 10: Visualization of querying nearby tiles and updating the 3×3 window.

which we have already constructed. We do this by nondeterministically attempting to build two structures in parallel, as shown in Fig. 10a, and can deduce the value of $(i-1, 2)$ based on which one succeeds. If the upper left space $(i-1, 2)$ is empty, then it is possible to place a tile there; using double-bonded probe tiles, we build south from the scaffold and then west into the potential empty space. If this construction succeeds, we know that the space is empty. We exploit cooperative bonding to determine if $(i-1, 2)$ is filled. Structure tiles connect to each other with double bonds; each structure tile, however, has at least a single bond on its east edge. Our probe tile, then, has a single bond on its north and west edges. It can bond cooperatively with the scaffold and space $(i-1, 2)$ only if $(i-1, 2)$ is filled. We use ALCH's `branch` structure to nondeterministically try both paths until one succeeds, at which point our program knows the upper-left space of the 3×3 grid. We can then calculate the upper-center and upper-right spaces using the XOR characterization of the DST.

With the grid filled in, our program can put the correct tile into solution (or skip forward if no tile is required). All incorrect bond sites in column i are covered by the vertical scaffold, so our tile is guaranteed to bond at the correct location. We must then "slide" the 3×3 grid one space north (updating the Boolean flags accordingly) to process the next tile site, as illustrated in Fig. 10b. The lowest six spaces of the new grid overlap with the old grid, so we already know them. As during initialization, we can calculate the upper-left space using the probe method and the remaining two using XOR. We proceed in this fashion up the entire column until it is completed. Note that when adding tiles in the middle of column i, we must make sure they do not bond into column $i+1$ using bond sites on the part of column i that we have already constructed. We use even and odd bond types to prevent this; the tiles we add for column i are incompatible with the bond sites in column j.

4.3 Constructing the Upper Symmetric Triangle

We have discussed how to construct the lower symmetric triangle (LST); it is straightforward to extend this method to the upper symmetric triangle (UST). Since the DST is symmetric, we need not track any additional information. We generate a symmetric scaffold corresponding to the vertical scaffold discussed above. (Since the diagonal scaffold is off-center, we skip the symmetric version of $\boxed{SC_0}$.)

When we add a structure tile to the LST, we add its symmetric version as well. We must also make a straightforward modification to our method for finishing off the solid columns (rows in the UST) that signal diagonal scaffold cleanup.

5 Strict Self-Assembly of the Discrete Sierpinski Carpet

We have seen that the CRN-TAM can strictly self-assemble infinite shapes at scale 1 that the aTAM cannot. In this section, we show that these techniques can be easily modified to self-assemble other self-similar fractals such as the discrete Sierpinski carpet (DSC). We conjecture that all connected, self-similar fractals defined in [16] can be strictly self-assembled in this way.

Our construction of the discrete Sierpinski carpet uses similar tile types as the ones in our Sierpinski triangle construction. Since the DSC is also symmetric about the diagonal $y = x$, we self-assemble the lower-symmetric triangle (LST) and the upper-symmetric triangle (UST) in a similar way to the DST.

The LST is composed of the seed tile and forty-three structural tiles. The number of tiles is larger than the DST because we must keep track of three states to specify each position in the LST of the carpet, compared to two states in the triangle. Unlike the DST, we do not need to keep track of even and odd columns to mitigate tile conflicts and crosstalk. We resolve this crosstalk issue by modifying the scaffold to occlude bonding sites in the column we are constructing and in the adjacent column. However, we must duplicate the set of structural tiles to construct the UST without interference by transposing the tiles in the LST.

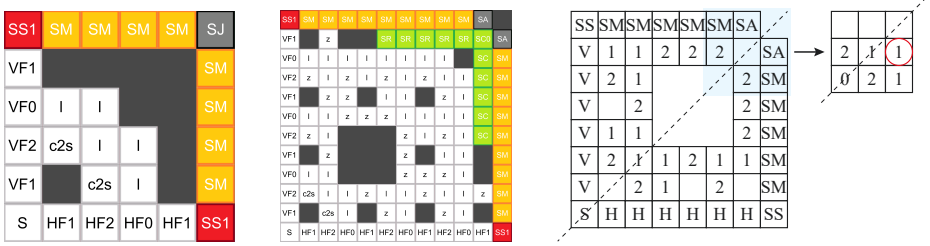
We use probe tiles in much the same way as in the DST, with the main difference that in the DSC, we modify the "filled" probe to detect the state of the tile in addition to its presence. We use the information gained by the probe tiles to compute the state of tiles which have not been placed yet.

Recall that the DST uses a diagonal scaffold to support its construction. In the DSC, two scaffolds are constructed, in vertical and horizontal directions for each layer. (In this construction, layers are concentric squares of increasing size.) As shown in Fig. 11a, we construct the right angle scaffold by building north and east from the frame at the same time. We find the intersection point of these two rays by cooperatively attaching a tile that joins these two structures, denoted as SJ in the figure.

As in the DST, these *right angle scaffolds* allow cooperative bonding of probes with the assembly and occlude bonding sites in the layer being constructed. These scaffolds have two advantages over the DST scaffolds. First, they do not interfere with tile positions that are part of the structure being built. Second, the DSC can be constructed without even and odd unique columns, reducing the number of tile types. This is accomplished by covering the outside edges of the structural tiles in the current layer under construction. This is shown in Fig. 11b.

We construct the right angle scaffold by simultaneously building north from the east edge of the horizontal frame and the building east from the north edge of the vertical frame. We find the intersection point of these two rays by cooperatively attaching a tile that joins these two structures.

Another new technique used in the DSC construction utilizes the diagonal symmetry of the shape to determine information about a tile position without probing. This is especially useful when information about the state of a tile is



(a) Right angle joins (b) Carpet scaffolding (c) Diagonal symmetry

Fig. 11: (a) Shows the right angle scaffold finding the intersection point of the current row and column. This structure enables use to build the DSC outward layer-by-layer. We also fully surround the construction area which prevents crosstalk during construction. (b) Demonstrates the vertical and horizontal construction scaffold. As with the DST this scaffold occludes all potential bonding sites in the previously constructed layer. (c) Illustrates how we exploit symmetry to gather information about the assembly without probing.

required that has not been placed in the assembly. However, symmetry of the shape allows us to determine this information from another tile position that does contain a placed tile. For example, in the DSC we can utilize this technique to determine the tile type to place at position (n, n) on the diagonal during construction of layer n. In this case, in order to determine which tile type to place at (n, n) we must determine the state of the tile at position $(n, n+1)$, which has not yet been placed. However, by symmetry tile position $(n + 1, n)$ is the same state, which we have already calculated and stored in species structured as a 3 by 3 window. Using this calculated symmetric state we can then determine the correct tile type to place at position (n, n). Fig. 11c depicts this situation in the assembly.

6 Conclusion

In this paper, we define ALCH, a programming language for the CRN-TAM. ALCH targets sequential CRN-TAM constructions, but supports multithreading as well. ALCH's notion of multithreading is powerful enough to create unbounded numbers of threads, as in our square construction that mimics the infinite tile supplies of the aTAM, but fully modular in the sense that multithreaded regions clean up after themselves and know when they have finished.

We use ALCH's sequential functionality to exhibit a strict self-assembly of the discrete Sierpinski triangle (DST) and the discrete Sierpinski carpet (DSC). Our use of ALCH allows us to conceptualize these constructions at the level of imperative tile commands and familiar control structures like conditionals and while loops. Furthermore, since it is impossible to strictly self-assemble the DST in the aTAM, our construction serves as a proof that the CRN-TAM can strictly self-assemble infinite shapes that the aTAM cannot.

We utilize two new techniques in our constructions. First, we have use a *probe* mechanism to measure which tiles have been placed, allowing us to derive information from the already-constructed system. The probe technique showcases ALCH's

nondeterministic branch structure, exploring multiple potential executions to find one that can complete. It also enables us to query the parts of the DST we have already constructed. Second, we have use a temporary scaffold to *occlude* undesirable tile bonding sites and precisely control where new tiles are added. Both of these techniques leverage the CRN-TAM's ability to remove tiles and create temporary structures.

We consider an alternate strategy to construct the DST using a CRN-TAM Turing machine implementation to control scaffold construction and tile placement. This entailed maintaining a secondary representation of the DST in the Turing machine tape, updating and querying it as the construction proceeds. The Turing machine would likely require unbounded storage to retain the last-constructed column even if it does not store the whole DST. On the other hand, our CRN-TAM construction acts as a "transformer," converting a stream of local data into a stream of tile placements without retaining unbounded information. The only part of the DST that we store in a computational form is the local 3×3 grid. We update it using the probe mechanism, thereby converting measurements of the existing DST into a bounded representation of the local DST area.

Our second technique, occluding bond sites with a temporary scaffold, is very general; we can apply it to any construction where we have a frontier of potential bond sites and must bond at a precise one. We expect this technique to be useful in constructing a wide variety of infinite shapes in the CRN-TAM. Our DST and DSC constructions do not require a Turing machine, but the full power of CRN-TAM universality is available to use in combination with occlusion scaffolds. We speculate that it is possible to construct every connected recursively enumerable subset of $\mathbb{Z}^2$ using variants of this technique.

Regardless of whether we can do this, however, under our definition of self-assembly from Section 4, we are not strictly *limited* to constructing shapes that are recursively enumerable. We can easily self-assemble basic recursively enumerable sets, such as a comb shape where the ith tine of the comb is present if and only if the ith Turing machine in some standard order halts on an empty input. We can *also* self-assemble the corresponding co-recursively enumerable set, i.e., the comb that has tines for each Turing machine that never halts. We begin creating the comb with all of its tines, and remove tines as our simulated Turing machines halt. (The recursive and co-recursive constructions both involve simulating all Turing machines in parallel using the well-known dovetailing technique: simulate the first for one step, then the first and second for one step, and so on.)

Note that adding or deleting a comb tine is straightforward, even in the middle of the constructed comb. Suppose we wish to add or remove a comb tine at position i. We can destroy the entire comb after position i, rebuild position i appropriately, and then restore the parts we destroyed. For each index $j \in \mathbb{N}$, there exists some count t_j of Turing machine timesteps after which all of the first j Turing machines that halt have done so. When we have simulated the first j Turing machines for at least t_j steps, we will have finalized the comb up to position j in accordance with our definition of shape self-assembly from the start of Section 4.

We find it somewhat unsatisfying that our notion of assembly allows us to assemble undecidable co-recursively enumerable objects. CRN-TAM systems can "cheat" by constructing shapes in such a way that it is never possible to know whether a particular position has been finalized. A more restrictive notion of assembly would force a computability criterion onto tile position completion time.

Since many shapes of interest are completely decidable, including the ones discussed above, we do not address the theoretical ramifications in this work.

We hope that ALCH and the tools and techniques presented here will catalyze research into the CRN-TAM and similar hybrid models.

Acknowledgements We thank the anonymous reviewers of the conference version of this paper for multiple useful suggestions. This research was supported in part by National Science Foundation grants 1900716 and 1545028.

References

1. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, March 2006.
2. Stefan Badelt, Seung Woo Shin, Robert F. Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. A General-Purpose CRN-to-DSD Compiler with Formal Verification, Optimization, and Simulation Capabilities. In Robert Brijder and Lulu Qian, editors, *Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming*, Lecture Notes in Computer Science, pages 232–248. Springer International Publishing, 2017.
3. Florent Becker. Pictures worth a thousand tiles, a geometrical programming language for self-assembly. *Theoretical Computer Science*, 410(16):1495–1515, 2009. Theory and Applications of Tiling.
4. Luca Cardelli. Kaemika app: Integrating protocols and chemical simulation. In Alessandro Abate, Tatjana Petrov, and Verena Wolf, editors, *Computational Methods in Systems Biology*, pages 373–379, Cham, 2020. Springer International Publishing.
5. Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In *Algorithmic Bioprocesses*, Natural Computing Series, pages 543–584. Springer, 2009.
6. David Doty and Mahsa Eftekhari. Efficient Size Estimation and Impossibility of Termination in Uniform Dense Population Protocols. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 34–42, New York, NY, USA, July 2019. Association for Computing Machinery.
7. David Doty, Jack H Lutz, Matthew J Patitz, Robert T Schweller, Scott M Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *Proceedings of the 53rd Symposium on Foundations of Computer Science*, pages 302–310. IEEE, 2012.
8. David Doty and Matthew J. Patitz. A domain-specific language for programming in the tile assembly model. In *Proceedings of the 17th International Conference on DNA Computing and Molecular Programming*, pages 25–34. Springer Berlin Heidelberg, 2009.
9. Irving Robert Epstein and John Anthony Pojman. *An Introduction to Nonlinear Chemical Dynamics: Oscillations, Waves, Patterns, and Chaos*. Oxford University Press, 1998.
10. François Fages, Guillaume Le Guludec, Olivier Bournez, and Amaury Pouly. Strong Turing Completeness of Continuous Chemical Reaction Networks and Compilation of Mixed Analog-Digital Programs. In Jérôme Feret and Heinz Koeppl, editors, *Proceedings of the 14th International Conference on Computational Methods in Systems Biology*, Lecture Notes in Computer Science, pages 108–127. Springer International Publishing, 2017.
11. Martin Feinberg. *Foundations of chemical reaction network theory*. Springer, 2019.
12. David Furcy, Scott M. Summers, and Christian Wendlandt. New bounds on the tile complexity of thin rectangles at temperature-1. In *Proceedings of the 25rd International Conference on DNA Computing and Molecular Programming*, pages 100–119. Springer International Publishing, 2019.
13. David Furcy, Scott M. Summers, and Christian Wendlandt. Self-assembly of and optimal encoding within thin rectangles at temperature-1 in 3D. *Theoretical Computer Science*, 2021.
14. Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, 1976.

15. 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 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2607–2624.
16. Steven M. Kautz and James I. Lathrop. Self-assembly of the Discrete Sierpinski Carpet and Related Fractals. In *Proceedings of the 15th International Conference on DNA Computing and Molecular Programming*, volume 5877 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2009.
17. Titus H. Klinge, James I. Lathrop, and Jack H. Lutz. Robust biomolecular finite automata. *Theoretical Computer Science*, 816:114–143, May 2020.
18. Titus H. Klinge, James I. Lathrop, Sonia Moreno, Hugh D. Potter, Narun K. Raman, and Matthew R. Riley. ALCH: An Imperative Language for Chemical Reaction Network-Controlled Tile Assembly. 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 6:1–6:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
19. James I. Lathrop, Jack H. Lutz, and Scott M. Summers. Strict self-assembly of discrete Sierpinski triangles. *Theoretical Computer Science*, 410(4):384–405, 2009.
20. Anthony M. L. Liekens and Chrisantha T. Fernando. Turing complete catalytic particle computers. In *Advances in Artificial Life*, pages 1202–1211. Springer Berlin Heidelberg, 2007.
21. Pierre-Étienne Meunier and Damien Woods. The non-cooperative tile assembly model is not intrinsically universal or capable of bounded Turing machine simulation. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 328–341. ACM, 2017.
22. Nicholas Schiefer and Erik Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming*, pages 34–54. Springer International Publishing, 2015.
23. Nicholas Schiefer and Erik Winfree. Time complexity of computation and construction in the chemical reaction network-controlled tile assembly model. In *Proceedings of the 22nd International Conference on DNA Computing and Molecular Programming*, pages 165–182. Springer International Publishing, 2016.
24. Nadrian C. Seeman. Nucleic acid junctions and lattices. *Journal of Theoretical Biology*, 99(2):237–247, 1982.
25. Eric E. Severson, David Haley, and David Doty. Composable computation in discrete chemical reaction networks. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 14–23, New York, NY, USA, 2019. Association for Computing Machinery.
26. David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, Dec 2008.
27. Marko Vasić, David Soloveichik, and Sarfraz Khurshid. CRN++: Molecular programming language. *Natural Computing*, 19(2):391–407, June 2020.
28. Erik Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998.
29. Erik Winfree. Chemical Reaction Networks and Stochastic Local Search. In Chris Thachuk and Yan Liu, editors, *Proceedings of the Twenty-Fifth International Conference on DNA Computing and Molecular Programming*, Lecture Notes in Computer Science, pages 1–20. Springer International Publishing, 2019.