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

Titus H. Klinge
Department of Mathematics and Computer Science, Drake University, Des Moines, IA, USA
titus.klinge@drake.edu

James I. Lathrop
Department of Computer Science, Iowa State University, Ames, IA, USA
jil@iastate.edu

Sonia Moreno
Department of Computer Science, Carleton College, Northfield, MN, USA
morenos@carleton.edu

Hugh D. Potter
Department of Computer Science, Iowa State University, Ames, IA, USA
hdpotter@iastate.edu

Narun K. Raman
Department of Computer Science, Carleton College, Northfield, MN, USA
ramann@carleton.edu

Matthew R. Riley
Department of Computer Science, Iowa State University, Ames, IA, USA
mrriley@iastate.edu

Abstract

In 2015 Schiefer and Winfree introduced the chemical reaction network-controlled tile assembly model (CRN-TAM), a variant of the abstract tile assembly model (aTAM), where tile reactions are mediated via non-local chemical signals. In this paper, we introduce ALCH, an imperative programming language for specifying CRN-TAM programs. ALCH contains common features like Boolean variables, conditionals, and loops. It also supports CRN-TAM-specific features such as adding and removing tiles. A unique feature of the language is the *branch* statement, a nondeterministic control structure that allows us to query the current state of tile assemblies. We also developed a compiler that translates ALCH to the CRN-TAM, and a simulator that simulates and visualizes the self-assembly of a CRN-TAM program. Using this language, we show that the discrete Sierpinski triangle can be strictly self-assembled in the CRN-TAM. This solves an open problem that the CRN-TAM is capable of self-assembling infinite shapes at scale one that the aTAM cannot. ALCH allows us to present this construction at a high level, abstracting species and reactions into C-like code that is simpler to understand. Our construction utilizes two new CRN-TAM techniques that allow us to tackle this open problem. First, it employs the branching feature of ALCH to *probe* the previously placed tiles of the assembly and detect the presence and absence of tiles. Second, it uses scaffolding tiles to precisely control tile placement by occluding any undesired binding sites.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases Tile assembly, Chemical reaction network, Sierpinski triangle

Digital Object Identifier 10.4230/LIPIcs.DNA.2020.6

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

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

Acknowledgements We thank the three anonymous reviewers for their helpful comments and suggestions. We especially thank reviewer 3 for their detailed insights, comments, and suggestions.

1 Introduction

Molecular programming is a relatively new field that weaves together biology and computer science to specify the behavior of molecules at the nanoscale. Early research in the field was sparked in 1982 by Seeman's pioneering work employing DNA crossover tiles to self-assemble crystals at the nanoscale [13]. Seeman's work was later extended by Erik Winfree to include *cooperative* DNA tile self-assembly to construct more complex shapes and patterns [15]. Winfree formalized the abstract tile assembly model (aTAM) in his Ph.D. thesis, where he proved it is Turing complete [15]. As a result, the aTAM is considered a programming language for self-assembling two and three-dimensional nanoscale patterns and is still actively investigated today [8, 3, 10, 7].

Another model commonly used to study biomolecular computation is the *chemical reaction network* (*CRN*), which models the interactions of chemical species. The CRN model assumes the solution is well-mixed, and therefore computations are amorphous and do not rely on geometry or structure. Two common variants of the CRN model are *stochastic CRNs* and *deterministic CRNs*. Stochastic CRNs are modeled with discrete species counts, and their reactions are probabilistic. In contrast, deterministic CRNs model the species' state continuously with real-valued concentrations governed by a system of autonomous ordinary differential equations (ODEs). The law of mass action determines the rates of reactions in both models. For more information on these models, see [6, 5, 2].

In 2015, Schiefer and Winfree introduced the *chemical reaction network-controlled tile assembly model* (*CRN-TAM*) [11, 12]. Their model combines the amorphous properties of stochastic CRNs with the spatial self-assembly of complex structures afforded by the aTAM. More specifically, a chemical reaction network interacts with tiles from the aTAM model to exert non-local control over the self-assembly process.

Molecular programming provides a rich field for algorithmic study. However, it is often time-consuming and complex to generate algorithmic constructions at the level of chemical species, tiles, or reactions. Recently, Vasić, Soloveichik, and Khurshid introduced CRN++, a high-level language for implementing deterministic CRN programs [14]. The CRN++ language provides a toolset for manipulating concentrations as numerical variables, with some support for conditionals and loops. This simplifies the development of high-level deterministic CRNs by abstracting away many low-level details. Other such languages exist such as Liekens and Fernando's Chemical Bare Bones (CBB), a hypothetical chemical implementation of the simple but Turing complete Bare Bones programming language [9]. CBB implements increment, decrement, and loop instructions using a catalytic particle model in which a single multistate particle catalyzes reactions based on its state. However, these languages cannot be used for CRN-TAM programs, since they have no provision for tile self-assembly.

On the tile self-assembly side, we have seen several forms of abstraction. Becker presents a geometry-based system for generating shapes in the aTAM [1]. This system allows users to describe how information and assembly construction propagate along vectors defined in the physical space of the assembly. Users can then generate an aTAM system by designing a system of vectors and applying a well-defined procedure to convert it into tiles. Doty and Patitz provide a toolset at a lower level of abstraction, focusing on the connections between individual tiles and how information is shared across them [4]. Users can define variables to be transmitted from tile to tile via bond labels and transformation functions to "modify" those variables within a tile while specifying which sets of tiles can bond with which. The provided software then automatically generates an aTAM system. Both of these tools focus on the parallel, semi-uncoordinated concept of tile self-assembly typical of aTAM constructions. In the CRN-TAM, on the other hand, the CRN component allows precise control over which tiles are added and when.

CRN-TAM constructions often rely on sequences of reactions and tile attachments, with sequential execution enforced by associating a chemical species with each reaction in the chain. For this reason, the CRN-TAM is a natural fit for a high-level imperative programming language. In this paper, we present the Algorithmic Language for Chemistry (ALCH), an imperative language for specifying CRN-TAM programs. ALCH targets the specific CRN-TAM design paradigm described above, where the CRN component mediates a strictly controlled sequence of tile actions We do not intend ALCH in its current form to be used for highly parallel aTAM-style constructions.

ALCH is reminiscent of other popular imperative languages, supporting loops and conditionals but omitting numerical computation and function calls. ALCH also contains many CRN-TAM specific statements that abstract away low-level details of the model's underlying semantics while maintaining that statements are executed in sequence. ALCH also includes a *branch* statement, a control structure that allows CRN-TAM programs to nondeterministically choose between a finite number of self-assembly paths. We are not aware of any shape that can be constructed in the CRN-TAM but not in ALCH, but we do not claim that ALCH is as general as the CRN-TAM. We have implemented an ALCH compiler that translates ALCH code into a proper CRN-TAM program and a simulator that visualizes the assembly process of a CRN-TAM program[1].

Using ALCH, we demonstrate that the CRN-TAM can construct infinite shapes that the aTAM cannot. For example, the discrete Sierpinski triangle is a well-known self-similar fractal that can be *weakly* self-assembled in the aTAM [15] but cannot be *strictly* self-assembled [8]. Weak self-assembly allows for "filler" tiles to be used to propagate information through an assembly, whereas strict self-assembly disallows this. We show that the non-local communication provided by the CRN-TAM is sufficient to overcome this limitation. Using ALCH, we construct a CRN-TAM program that strictly self-assembles the discrete Sierpinski triangle. Our construction relies on the ability to add and remove scaffolding tiles and self-assembles the fractal in a natural way, using only localized information contained in the current assembly. We achieve this by using ALCH's nondeterministic branch feature to probe previously placed tiles to inform which tiles are placed next. We also use the scaffolding tiles to occlude any spurious bonding sites, giving precise control over the placement of the next tile. The construction proceeds in a sequence of stages where each stage successfully self-assembles a subset of the discrete Sierpinski triangle. After the completion of a stage, all scaffolding tiles are removed, leaving only the Sierpinski triangle tiles. Thus, in the limit, only the Sierpinski triangle remains, since the scaffolding tiles are removed infinitely often. In fact, the ratio of scaffold tiles to Sierpinski triangle tiles approaches zero as the self-assembly process proceeds. The ALCH programming language and simulator simplifies the development process and the specification of the CRN-TAM program.

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

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

2 Preliminaries

We now review the chemical reaction network-controlled tile assembly model (CRN-TAM), which combines the notions of the abstract tile-assembly model (aTAM) [15] and the stochastic chemical reaction network (sCRN) [2]. For a complete introduction to the model, see Schiefer and Winfree's original paper [11].

A *tile type* is a tuple $\boxed{t} = (N, E, S, W)$ consisting of four *bonds* for the north, east, south, and west sides of the tile, respectively. Each bond is a tuple $B = (\ell_B, s_B)$ where 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 $\alpha : \mathbb{Z}^2 \dashrightarrow T$ that encodes the positions of tiles in two-dimensional space. If $\alpha(i, j)$ is undefined, then we say that (i, j) is *unoccupied* in the assembly 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.

The literature is unclear about whether it is permissible to have bonds with the same label but asymmetric bond strengths; we have made the choice to allow it in this work. We adopt the prescription that adjacent bonds with the same label have interaction strength s, where s is given by the minimum of the bond strengths. Note that this prescription is physically plausible; if we view a bond site as an exposed single DNA strand, a stronger bond corresponds to a longer exposed area. We can then choose the base pairs exposed by a weaker bond to be a subset of those exposed by a stronger bond. Our probe mechanism, discussed in a subsequent section, relies on such asymmetric bonds.

The *binding graph* of an assembly 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. Note that the definitions in Schiefer and Winfree's papers [11, 12] allow tile removal species to be shared or even omitted. However, it is convenient for the compiler to always generate tile removal species and for them to be unique.

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

Signal reactions are of the form $X_1 + X_2 \to Y_1 + Y_2$ where $X_1, X_2, Y_1, Y_2 \in S \cup \{\epsilon\}$. The 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.

Deletion reactions are of the form $X + \boxed{t} \to Y_1 + Y_2$ where $X, Y_1, Y_2 \in S \cup \{\epsilon\}$ and $\boxed{t} \in T$. These reactions consume a tile, treating it as if it were a signal species. Note, deletion reaction cannot consume tiles bound to the assembly.

Creation reactions are of the form $X_1 + X_2 \to \boxed{t} + Y$ where $X_1, X_2, Y \in S \cup \{\epsilon\}$ and $\boxed{t} \in T$. These reactions produce tiles, making them available to interact with assemblies.

Relabelling reactions are of the form $X + \boxed{t_1} \to Y + \boxed{t_2}$ where $X, Y \in S \cup \{\epsilon\}$ and $\boxed{t_1}, \boxed{t_2} \in T$.

Activation reactions are of the form $X + \boxed{t} \to \boxed{\boxed{t}} + t^*$ where $X \in S$, $\boxed{t} \in T$, and t^* is the signal removal species for $\boxed{t}$. These reactions use tile $\boxed{t}$ to seed a new assembly with $\boxed{t}$ placed at the origin.

Deactivation reactions are of the form $\boxed{\boxed{t}} + t^* \to \boxed{t} + Y$ where $\boxed{t} \in T$, t^* is the removal signal for $\boxed{t}$, and $Y \in S \cup \{\epsilon\}$. These reactions remove the tile $\boxed{t}$ from the singleton assembly $\boxed{\boxed{t}}$, thereby deactivating it.

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

Addition reactions of the form $\boxed{\boxed{\alpha}} + \boxed{t} \to \boxed{\boxed{\beta}} + t^*$ 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 $\boxed{t}$.

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 DST construction in this work, do not terminate and continue indefinitely. For more information on the kinetics of the CRN-TAM model, see [12].

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

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

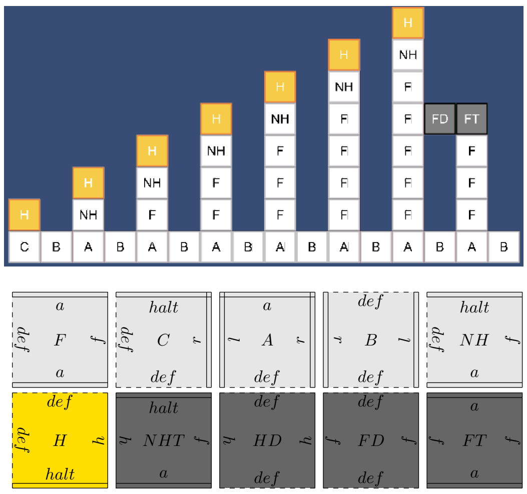
3 The ALCH Programming Language

We present an overview of the features of the ALCH language and its implementation. ALCH is an imperative language with provisions specific to the CRN-TAM model such as the **add**, **remove**, **activate**, and **deactivate** statements which all take a tile type as a parameter and execute the corresponding tile actions. ALCH provides high-level features such as conditions, loops, and variable declaration and assignment. To guarantee the proper sequential execution of the code, special *line number species* are used to track progress through the ALCH program. By ensuring that only a single line number species is present at any given time[2], the CRN-TAM program can transition from instruction to instruction without introducing any race conditions. At this time, ALCH only supports global variables and three datatypes: **bool**, **BondLabel**, and **TileSpecies**. Variables of type **bool** may be reassigned throughout the computation, but all **BondLabel** and **TileSpecies** variables are immutable and final. One unique feature of ALCH is the **branch** statement, which nondeterministically chooses and executes multiple independent code blocks of tile addition and removal statements until one block finishes execution. Effects from uncompleted blocks are reversed, so only the code from the completed block remains. The **branch** statement also returns a **bool** associated with the block that finished successfully. Using **branch**, it is possible to query the state of tile assemblies without permanently attaching tiles to them. Each block in a **branch** statement is implemented as a reversible random walk. As an optimization, blocks can be given different weights to make them more likely to be chosen at the nondeterministic branch point.

We developed a software compiler in C# that compiles ALCH programs into CRN-TAM programs. We also developed a simulator for the CRN-TAM that includes the following two extensions to the model which are used only for optimization purposes: (1) it supports reactions with arbitrary arity, relaxing the CRN-TAM requirement that reactions are at most bimolecular; (2) it allows any reaction to add, remove, or activate a tile as a side effect and removes the requirement for the specific per-tile add and remove actions. Note that the output of the ALCH compiler is strictly compliant with the original CRN-TAM as specified in [11]. We have not yet implemented tile deactivation in the simulator.

To demonstrate the expressiveness of ALCH, we will show that the CRN-TAM can strictly self-assemble an infinite shape at temperature 2 that the aTAM cannot. Consider an infinite staircase, visualized in Figure 1, where for each $k \in \mathbb{N}$, the $(2k)$th column is $2 + k$ tiles tall and the $(2k + 1)$th column is one tile tall. The gaps between steps (even-numbered columns) prevent an aTAM program from directly transferring information about the height of one step to the next. Consequently, all information about the height of steps must be passed along the base of the assembly; an infinite tileset is required. However, the CRN-TAM can build and remove probe tiles that allow the assembly to query the previous column. We take advantage of this and show that the CRN-TAM can self-assemble this infinite shape, as shown in Figure 1. Note that we omit the tile and bond declarations but include a graphical representation of the tile species used in the construction. We also omit the CRN species and reactions that ALCH outputs.

[2] See Subsection 3.3 for the one exception.



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

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

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

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

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

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

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

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

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

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

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

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 evaluates to false. Only when all expressions have evaluated to true will a true Boolean flag be returned. Below is how the conjunction statement <exp1> && <exp2> is implemented:

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

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

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

Here the species T_{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 with the conventional syntax as shown below:

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

The implementation below is similar to the previous constructions above.

$$X_{\text{start}} \to \text{<exp>} \to X_1 \tag{19}$$

$$X_1 + T_{\text{<exp>}} \to X_t + T_{\text{<exp>}} \tag{20}$$

$$X_1 + F_{\text{<exp>}} \to X_{\text{end}} + F_{\text{<exp>}} \tag{21}$$

$$X_t \to \text{<block>} \to X_{\text{end}} \tag{22}$$

We also support **else** blocks by modifying Reaction (21) to output an X_f molecule and adding an additional reaction $X_f \to X_2$ where X_2 is the line number species for the **else** block. ALCH also supports **while** loops which are implemented in a similar fashion but alternates between the line number for <exp> and the internal <block>.

3.3 Tile Addition, Removal, Activation, and Deactivation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4 Nondeterministic Branch Construct

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

It is important to note that our notion of reversibility is not complete. For example, suppose we execute **add** tileA inside a branch path. If this statement is reversed, the system will attempt to remove the tile $\boxed{A}$. However, if there are multiple instances of $\boxed{A}$ bonded to the assembly, it is not guaranteed to remove the same tile added earlier in the branch. Additionally, if we add a tile at a strength greater than 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.

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

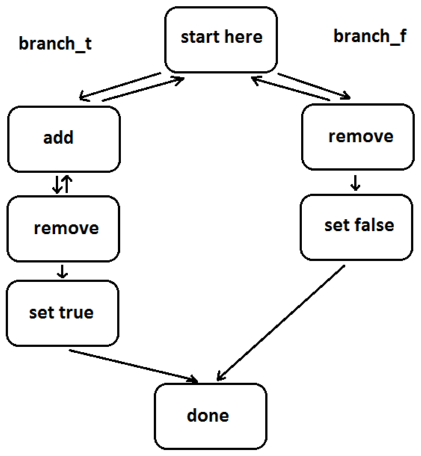


Figure 2 Possible execution paths through a branch statement. Instructions associated with **true** and instructions associated with **false** are executed nondeterministically via a random walk. The **branch** statement terminates when one path runs to completion, and it returns the corresponding Boolean flag. The integers inside the parentheses of the **true** and **false** branches correspond to *weights* that bias the random walk.

The **branch** statement is implemented with a single branch point that can lead to any one of the branch paths, as shown in Figure 2. From that branch point, we execute only one branch path at a time. Since each branch path is reversible, if execution proceeds down a branch that is incapable of completing, it will eventually return to the branch point via random walk. When a branch finishes execution, we return the Boolean flag that corresponds with the path that completed.

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

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

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

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

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

A few things should be noted about the above implementation. First, both the <trueblock> and <falseblock> use the same line number species X_{start}. Second, those reactions are reversible, as indicated by the bidirectional arrows. Third, once one of the blocks finishes, it is completed with an irreversible reaction that terminates the **branch** statement. Fourth, the **add** and **remove** commands outside of **branch** are not reversible; inside branch paths, we modify each add and remove command to make them reversible. The reversible implementation for the **add** statement is shown below.

$$X_1 \leftrightarrow \boxed{A} \tag{35}$$

$$A^* \leftrightarrow X_2 \tag{36}$$

A reversible **remove** statement is implemented in a similar way but is not shown.

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

$$X_t + T_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{37}$$

$$X_t + F_{\text{result}} \to X_{\text{end}} + T_{\text{result}} \tag{38}$$

$$X_f + T_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{39}$$

$$X_f + F_{\text{result}} \to X_{\text{end}} + F_{\text{result}} \tag{40}$$

4 Strict Self-Assembly of the Discrete Sierpinski Triangle

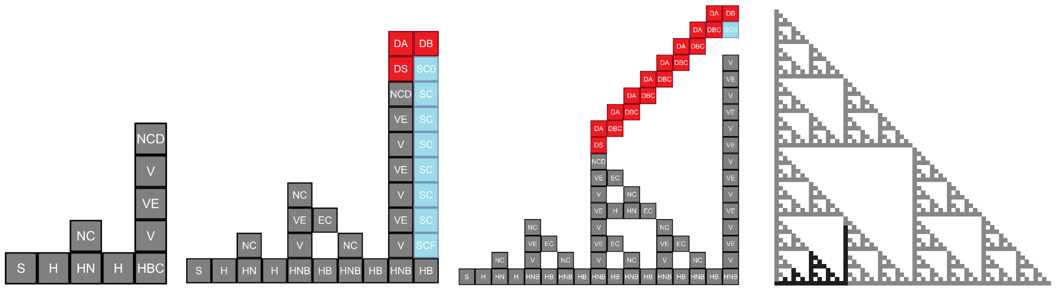
We now present the CRN-TAM construction that strictly self-assembles the discrete Sierpinski triangle (DST) using ALCH. Our discussion here is complete but brief; see Appendix A for a more detailed description of our algorithm. To see the complete specification of the construction in ALCH, along with a video visualization of the self-assembly, see `http://web.cs.iastate.edu/~lamp/`.

We begin with an overview of tile types and a brief description of their purpose and then describe the DST construction algorithm in detail. Since the DST is symmetric about the line $f(x) = x$, we refer to the two symmetric halves as the lower symmetric triangle (LST) and the upper symmetric triangle (UST). We first discuss the techniques to strictly self-assemble the LST, which can be easily modified to construct the UST in parallel. In our construction, it is useful to distinguish between three types of tiles: (1) structural tiles, (2) scaffold tiles, and (3) probe tiles. Structural tiles are permanent and form the DST itself. Scaffold tiles are used to construct temporary auxiliary structures to facilitate the DST construction. Probe tiles are rapidly added and removed to query existing information of previously placed structural tiles. To avoid unwanted crosstalk between the symmetric halves, we duplicate the set of structure tiles into a symmetric group with bonds that are incompatible with the LST tiles. We also differentiate the tile types of even and odd columns to prevent a partially constructed column from interfering with the construction.

We now discuss the construction for the strict self-assembly of the DST. The first step in our construction unpacks the initial structure shown in Figure 3a with hard-coded tile activation and addition statements. This is easily accomplished by adding tiles in a specific order that avoids ambiguity in placement. After the initial structure tiles are placed, we then construct the LST column by column, adding structure tiles one-at-a-time, completing each column before proceeding to the next. We also use a variable to track whether we are currently constructing an even or odd column. The process of adding one structure tile at a time is akin to a dot-matrix printer, placing dots of ink one line at a time.

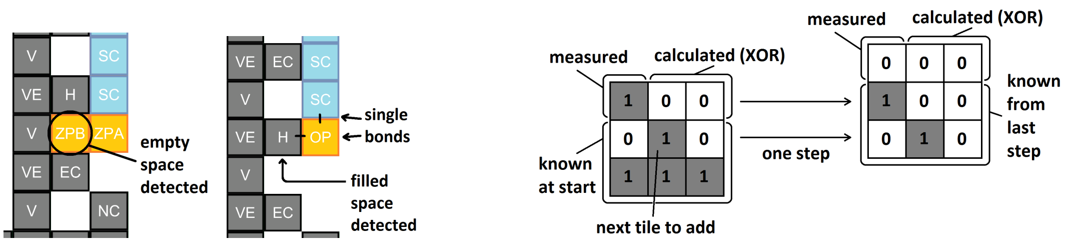
4.1 Scaffold Construction

We construct two types of scaffolds. The diagonal scaffold, shown in red in Figure 3, runs along the diagonal of the DST and provides an anchor for the vertical scaffold, which is shown in cyan. The vertical scaffold covers up potential bond sites that we do not wish to bond to, as illustrated in Figure 3b. The diagonal scaffold is straightforward to construct; before constructing each column, we extend it out by two more tiles. For the vertical scaffold, we must extend it only as far as the base of the DST. We extend the DST base row out by



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

Figure 3 (a) The initial hard-coded structure upon which we build the lower half of the DST. (In the final program that constructs the whole DST, this structure has a symmetric upper half.) (b) Demonstrates how our construction extends a vertical scaffold down to occlude all the potential tile bond sites on column currently being constructed. (c) Shows the diagonal scaffold before erasing itself and starting a new diagonal scaffold. (d) Shows a section of the Sierpinski triangle that includes the lower and upper symmetric halves; the part corresponding to (c) is highlighted.



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

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

Figure 4 Visualization of the probe querying nearby tiles and updating the 3×3 window.

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

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

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

4.2 Adding Structure Tiles with the Probe

When beginning to place tiles on a new column i, the vertical scaffold must be completely initialized as in Figure 3b. We must know which tile, if any to add to the DST at each vertical position: T-joint, straight connector, etc. To that end, after constructing the vertical scaffold, we initialize a 3×3 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 $i - 1$, which we have already constructed. We do this by nondeterministically attempting to build two structures in parallel, as shown in Figure 4a, and can deduce the value of $(i - 1, 2)$ based on which one succeeds. If the upper left space $(i - 1, 2)$ is empty, then it is possible to place a tile there; using double-bonded probe tiles, we build south from the scaffold and then west into the potential empty space. If this construction succeeds, we know that the space is empty. We exploit cooperative bonding to determine if $(i - 1, 2)$ is filled. Structure tiles connect to each other with double bonds; each structure tile, however, has at least a single bond on its east edge. Our probe tile, then, has a single bond on its north and west edges. It can bond cooperatively with the scaffold and space $(i - 1, 2)$ only if $(i - 1, 2)$ is filled. We use ALCH's `branch` structure to nondeterministically try both paths until one succeeds, at which point our program knows the upper-left space of the 3×3 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 Figure 4b. The lowest six spaces of the new grid overlap with the old grid, so we already know them. As during initialization, we can calculate the upper-left space using the probe method and the remaining two using XOR. We proceed in this fashion up the entire column until it is completed. Note that when adding tiles in the middle of column i, we must make sure they do not bond into column $i + 1$ using bond sites on the part of column i that we have already constructed. We use even and odd bond types to prevent this; the tiles we add for column i are incompatible with the bond sites in column j.

4.3 Constructing the Upper Symmetric Triangle

We have discussed how to construct the lower symmetric triangle (LST); it is straightforward to extend this method to the upper symmetric triangle (UST). Since the DST is symmetric, we need not track any additional information. We generate a symmetric scaffold corresponding to the vertical scaffold discussed above. (Since the diagonal scaffold is off-center, we skip the symmetric version of $\boxed{SC_0}$.) When we add a structure tile to the LST, we add its symmetric version as well. We must also make a straightforward modification to our method for finishing off the solid columns (rows in the UST) that signal diagonal scaffold cleanup; see the appendix for details.

5 Conclusion

In this paper, we define ALCH, a programming language for the CRN-TAM, and use it to exhibit a strict self-assembly of the discrete Sierpinski triangle (DST). Our use of ALCH allows us to conceptualize our construction at the level of imperative tile commands and familiar control structures like conditionals and while loops. Furthermore, since it is impossible to strictly self-assemble the DST in the aTAM, our construction serves as a proof that the CRN-TAM can strictly self-assemble infinite shapes that the aTAM cannot.

We have utilized two new techniques in our DST construction. First, we have used a *probe* mechanism to measure which tiles have been placed, allowing us to derive information from the already-constructed system. The probe technique showcases ALCH's nondeterministic branch structure, exploring multiple potential executions to find one that can complete. It also enables us to query the parts of the DST we have already constructed. Second, we have used a temporary scaffold to *occlude* undesirable tile bonding sites and precisely control where new tiles are added. Both of these techniques leverage the CRN-TAM's ability to remove tiles and create temporary structures.

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

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

For the current version of ALCH, we have focused on a very sequential programming model. However, the CRN-TAM, allows for potentially massive parallelism via large chemical populations; it would be interesting to explore additional ALCH features that leverage this capability. For example, the aTAM tileset design toolkit by Doty and Patitz [4] provides an abstraction for highly-parallel tile assembly. Incorporating a similar tool into ALCH could enable powerful constructions that combine chemical parallelism with the coordination capabilities of ALCH's imperative framework. More broadly, we speculate that ideas from classical concurrent programming are relevant to ALCH as well.

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

References

1 Florent Becker. Pictures worth a thousand tiles, a geometrical programming language for self-assembly. *Theoretical Computer Science*, 410(16):1495–1515, 2009. Theory and Applications of Tiling. doi:10.1016/j.tcs.2008.12.011.

2 Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In *Algorithmic Bioprocesses*, Natural Computing Series, pages 543–584. Springer, 2009. doi:10.1007/978-3-540-88869-7_27.

3 David Doty, Jack H Lutz, Matthew J Patitz, Robert T Schweller, Scott M Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *Proceedings of the 53rd Symposium on Foundations of Computer Science*, pages 302–310. IEEE, 2012. doi:10.1109/FOCS.2012.76.

4 David Doty and Matthew J. Patitz. A domain-specific language for programming in the tile assembly model. In *Proceedings of the 17th International Conference on DNA Computing*

and Molecular Programming, pages 25–34. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-10604-0_3.

5 Irving Robert Epstein and John Anthony Pojman. *An Introduction to Nonlinear Chemical Dynamics: Oscillations, Waves, Patterns, and Chaos.* Oxford University Press, 1998. doi:10.1021/ed077p450.1.

6 Martin Feinberg. *Foundations of chemical reaction network theory.* Springer, 2019. doi:10.1007/978-3-030-03858-8.

7 David Furcy, Scott M. Summers, and Christian Wendlandt. New bounds on the tile complexity of thin rectangles at temperature-1. In *Proceedings of the 25rd International Conference on DNA Computing and Molecular Programming*, pages 100–119. Springer International Publishing, 2019. doi:10.1007/978-3-030-26807-7_6.

8 James I. Lathrop, Jack H. Lutz, and Scott M. Summers. Strict self-assembly of discrete Sierpinski triangles. *Theoretical Computer Science*, 410(4):384–405, 2009. doi:10.1016/j.tcs.2008.09.062.

9 Anthony M. L. Liekens and Chrisantha T. Fernando. Turing complete catalytic particle computers. In *Advances in Artificial Life*, pages 1202–1211. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74913-4_120.

10 Pierre-Étienne Meunier and Damien Woods. The non-cooperative tile assembly model is not intrinsically universal or capable of bounded Turing machine simulation. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 328–341. ACM, 2017. doi:10.1145/3055399.3055446.

11 Nicholas Schiefer and Erik Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming*, pages 34–54. Springer International Publishing, 2015. doi:10.1007/978-3-319-21999-8_3.

12 Nicholas Schiefer and Erik Winfree. Time complexity of computation and construction in the chemical reaction network-controlled tile assembly model. In *Proceedings of the 22nd International Conference on DNA Computing and Molecular Programming*, pages 165–182. Springer International Publishing, 2016. doi:10.1007/978-3-319-43994-5_11.

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

14 Marko Vasić, David Soloveichik, and Sarfraz Khurshid. CRN++: Molecular programming language. *Natural Computing*, pages 1–17, 2020. doi:10.1007/s11047-019-09775-1.

15 Erik Winfree. *Algorithmic self-assembly of DNA.* PhD thesis, California Institute of Technology, 1998. URL: https://resolver.caltech.edu/CaltechETD:etd-05192003-110022.

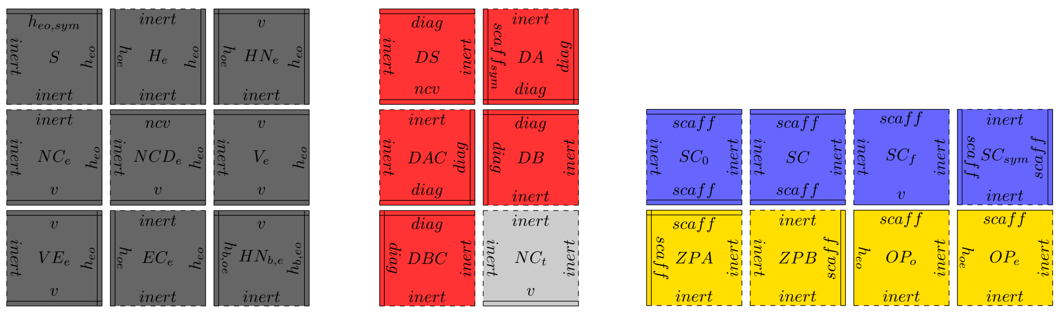
A Strict DST Construction: Details

We now present a more detailed look into our DST construction in ALCH. We begin with an overview of tile types and a brief description of their purpose; we then describe the DST construction algorithm in detail.

The DST is symmetric about the line $y = x$. We refer to the two symmetric halves as the lower symmetric triangle (LST) and the upper symmetric triangle (UST). We will focus on the LST construction algorithm, as it can be easily modified to construct the UST at the same time.

A.1 Tile Types

We distinguish three types of tiles. Structure tiles form the DST itself. Scaffold tiles form semi-permanent auxiliary scaffolds that enable us to build the DST, and probe tiles are added and removed quickly to probe the existing structure for useful information. One commonality between all three tile types is the *inert* bond label, which we use always at strength 0.



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

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

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

Figure 5 Tiles types used in the DST construction.

A.1.1 Structure Tiles

Structure tiles use several bond labels for the LST.

- v is a strength 2 vertical bond that joins structure tiles in completed regions of the DST.
- h_{eo} and h_{oe} are likewise structural horizontal bonds. We must disambiguate between even and odd columns; h_{eo} joins an even-column tile on the left with an odd-column tile on the right, and h_{oe} is the reverse.
- $h_{b,eo}$ and $h_{b,oe}$ are variants that mark the lowest ("base") row in the DST.
- ncv interfaces structure tiles with one type of scaffold tiles.

To avoid unwanted crosstalk between the symmetric halves, we duplicate the set of structure tiles into a symmetric group with bonds that are incompatible with the LST tiles. Likewise, we use separate bond labels and tile types to avoid crosstalk between even and odd columns. This produces four similar categories of structure tile: even LST, odd LST, even UST, and odd UST. We present a list of even LST tiles in Figure 5a. Note that most structure tiles have an h_{eo} bond of strength at least one on their eastern edges so that probe tiles can attach cooperatively.

Tile $\boxed{S}$ is the seed tile that we activate to form the southwest corner of the DST. Tile $\boxed{NCD_e}$ interfaces between the structure and the scaffold, and tile $\boxed{HN_{b,e}}$ is a variant tile type that occurs specifically on the lowest row. All the other structure tile types in Figure 5a fill in the DST structure in a straightforward way.

A.1.2 Scaffold Tiles

We use two types of scaffolds. The vertical scaffold extends along the eastern face where the next column is to be added; we extend and retract it to expose structural tile addition sites. The diagonal scaffold extends along parts of the southwest-to-northeast diagonal and provides an attachment point for the vertical scaffold. We require two additional bond labels: $scaff$ for the vertical scaffold and $diag$ for the diagonal scaffold.

See Figure 5b for a list of diagonal scaffold tiles. Tile $\boxed{DS}$ interfaces with the structure tiles and begins the diagonal scaffold; tiles $\boxed{DA}$ and $\boxed{DB}$ form the body of the diagonal. Since $\boxed{DA}$ and $\boxed{DB}$ contain bond sites to begin the vertical scaffolds, when we finish with a column we must replace them with the capped variants $\boxed{DAC}$ and $\boxed{DBC}$ so future vertical scaffolds don't spuriously bond there. We use $\boxed{NC_t}$ as a temporary variant of $\boxed{NC_e}$ that is useful for cleaning up the scaffold.

We present a list of vertical scaffold tiles in Figure 5c. Tiles $\boxed{SC}$ and $\boxed{SC_{sym}}$ form the body of the vertical and symmetric horizontal scaffolds. We use $\boxed{SC_0}$ and $\boxed{SC_f}$ at the beginning and end of the LST vertical scaffold so that we can identify when we are done adding and removing it; since we know this information from the LST, we don't require corresponding symmetric tile species.

A.1.3 Probe Tiles

When constructing a new column, we use a probe mechanism to determine whether specific rows in the last constructed column contain structure tiles; this allows us to use XOR to reconstruct the DST with constant information stored in chemical species counts. We have separate probe mechanisms to detect "zeros" (empty positions) and "ones" (filled positions). See Figure 5c for a list of probe tiles.

A.2 Initialization

We now begin our discussion of the DST construction algorithm. First, we prepare the structure shown in Figure 3a, using a straightforward series of tile additions that do not result in ambiguity. We also initialize to odd the flag that tracks whether we are in an even or an odd column.

We face two challenges when constructing the rest of the triangle:

- We must add each tile in the correct location, instead of any of a potentially unbounded number of incorrect locations.
- At each position, we must determine which tile to add, if any; i.e., we must know whether to add nothing, $\boxed{EC_e}$, $\boxed{HN_e}$, etc.

We solve the first problem by tracking the tile positions around the tile position in question. To solve the second problem, we extend a scaffold of tiles to occlude all unintended bond sites.

To begin, we add the diagonal scaffold tile $\boxed{DS}$ above $\boxed{NCD_e}$; this will be the start of our occluding scaffold. Immediately after $\boxed{DS}$ is added, we enter a loop construct in our algorithm. We will refer to this loop as the outer loop; each outer loop iteration constructs another column of the LST.

A.3 Outer Loop

A.3.1 Initialization: building the scaffold

Inside the loop, we must first build out the scaffold. We add $\boxed{DA}$ and $\boxed{DB}$ to $\boxed{DS}$, and we extend the base row with $\boxed{H_{b,o}}$ (or $\boxed{HN_{b,e}}$ in an even row). Since we have a tile set specifically for constructing the base layer, we don't need to worry about adding $\boxed{H_{b,o}}$ in the wrong row.

Now, we construct the vertical scaffold down from $\boxed{DB}$ to produce a structure like the one shown in Fig 3b. We add $\boxed{SC_0}$ first so that when we remove it again we will know we have reached the top; as discussed below, $\boxed{SC_f}$ is a mechanism to detect the bottom row.

We then add $\boxed{SC}$ until we reach the bottom row. Since we have added $\boxed{H_{b,o}}$ extending out, we cannot add $\boxed{SC}$ at row 0 or lower. We must detect when we reach the bottom, however, so we can stop attempting to add $\boxed{SC}$ and continue with the rest of the program.

Whenever we attempt to add $\boxed{SC}$, we also attempt to add $\boxed{SC_f}$ in parallel using the branch structure. Recall that $\boxed{H_{b,o}}$ always has a bond site on its north edge; since $\boxed{SC_f}$ has single bonds on its north and south edges and must bond at strength 2, it can only bond in row 1 between $\boxed{H_{b,o}}$ to the south and $\boxed{SC}$ to the north.

It may be that $\boxed{SC}$ bonds in row 1 instead of $\boxed{SC_f}$; we always add $\boxed{SC}$ reversibly so that if this happens the program can proceed (and can *only* proceed) by removing $\boxed{SC}$. In this way we have as many chances as we need to add $\boxed{SC_f}$ and continue with the program. We attempt to add $\boxed{SC}$ in one branch and $\boxed{SC_f}$ in another; the return value tells us whether we have finished adding the scaffold.

A.3.2 Guaranteeing correct added tile position

We can now remove the vertical scaffold row by row, exposing only one tile addition site at a time. There are two types of addition sites: north and east edges of preexisting tiles. We claim that when we add a new structure tile, at most one potential bond site is exposed, so the tile is added unambiguously.

Recall that we have separate bond types for even and odd columns; an odd-column structure tile cannot bond to the east side of another odd-column structure tile, and likewise with even columns. If we are building column i, then, we don't need to worry about unintended bonding in column $i + 1$. In column i itself, the region above our intended bond site is covered by vertical scaffold tiles and is therefore not a concern. In the region below our intended bond site, all viable bond sites have already been taken up. We can therefore guarantee that we can always add the next DST structure tile unambiguously.

A.3.3 Choosing the correct tile

Now that we can guarantee that tiles are added at the correct position, we must determine which tile to add and whether or not to add one at all.

We store a 3×3 "window" of boolean flags around the tile position where we will potentially add a tile, as shown in Fig. 4b. Each flag is true or false based on whether the corresponding position in the DST is full or empty. Note that if we possess this information, it is easy to determine whether we must add the center tile, and, if so, which tile we must add.

If we are constructing column i, we have added the first tile $\boxed{H_{b,o}}$ or $\boxed{HN_{b,e}}$ at position $(i, 0)$. We will therefore initialize the 3×3 grid centered on $(i, 1)$, which is the next potential tile position to fill. The bottom row is always filled in all three positions by the base row, so we can initialize the lower three flags to true. The second row in the DST always alternates between full and empty, so we need to set either the center flag or the center-left and center-right flags to true depending on whether we are building an odd or an even column. Since we track this information, we can easily initialize the middle row.

We do not immediately have enough information to initialize the upper tiles. Recall, however, that the DST can be characterized as a cellular automaton based on the XOR relation $\oplus$:

$$DST[x, y] \leftrightarrow DST[x-1, y] \oplus DST[x, y-1]. \tag{41}$$

Therefore if we could somehow measure the upper-left tile, we could calculate the upper-center and upper-right tiles.

We can measure the upper-left tile $(i-1,2)$ using the branch construct. We require two series of tile additions: one that is only possible if $(i-1,2)$ is empty, and one that is only possible if it contains a tile.

If $(i-1,2)$ is empty, we can add a "zero probe" tile into that location; we therefore attempt to add such a tile, first building $\boxed{ZPA}$ down from the scaffold and then attempting to build $\boxed{ZPB}$ at $(i-1,2)$. See Fig. 4a for an illustration.

Recall that all structural tiles have an east bond site of strength at least one. We therefore attempt in parallel to add a "one probe" $\boxed{OP}$ with strength-one north and west bond sites. If there is a structural tile in $(i-1,2)$, then the one probe can bond cooperatively with it and the vertical scaffold, as shown in Fig. 4a.

We perform these attempts in parallel using the branch construct. It is possible that $\boxed{ZPA}$ will bond when $(i-1,2)$ is full. Since we add $\boxed{ZPA}$ reversibly, this is not a problem; the program can only proceed by removing $\boxed{ZPA}$, and $\boxed{OP}$ then has another chance to bond. It is clear, then, that only the correct branch can fully complete. When it does, the branch statement returns the correct value of $(i-1,2)$.

Once we know $(i-1,2)$, we can calculate the upper-center tile value in our 3×3 window using the XOR characterization of the DST. We can then similarly calculate the upper-right tile; that completes the grid, and we can add the appropriate tile into the exposed bond site or skip it if no tile is required.

We must then adjust the grid so it is centered on $(i,2)$ instead of $(i,1)$. Note that the lower two rows of the new grid must be the same as the upper two rows of the old grid, which we have already calculated and stored. We therefore need to calculate only the top row. We can measure $(i-1,3)$ in the same way we measured $(i-1,2)$; this allows us to calculate the new top row the same way we calculated the old top row. We can then continue adding or skipping tiles and sliding the grid upwards iteratively as we construct the column; one sliding iteration is shown in Fig. 4b.

At some point we will attempt to remove $\boxed{SC}$ and instead remove $\boxed{SC_0}$; we detect this with the branch construct and terminate the column loop. This also signals the end of the outer loop. We remove $\boxed{DB}$ and replace it with $\boxed{DBC}$ so that new $\boxed{SC}$ and $\boxed{SC_0}$ tiles can't bond there and restart the outer loop.

A.4 Cleaning Up the Diagonal

As we build additional columns, we extend the diagonal scaffold along the diagonal of the DST; since it is not part of the DST, we must periodically remove it to "clean up" our construction.

At every horizontal coordinate that is a power of 2, the DST contains a column of filled cells that extends all the way from the baseline to the diagonal, as shown in Fig 3c. We can detect this in our program by inspecting the state of the 3×3 grid when we remove $\boxed{SC_0}$; if we have just completed a column of filled cells, we clean up the diagonal in the region to the left of the completed column.

We began the diagonal scaffold with a special tile $\boxed{DS}$, just as we begin the vertical scaffolds with $\boxed{SC_0}$. We can therefore remove $\boxed{DA}$ and $\boxed{DBC}$ repeatedly until we can instead remove $\boxed{DS}$ with the branch construct. When we remove $\boxed{DS}$, we have cleaned up the scaffold.

Recall that there is a specific tile $\boxed{NCD_e}$ with a north bond site that allows the diagonal scaffold to connect. On the old column that supported the diagonal scaffold, we must replace $\boxed{NCD_e}$ with $\boxed{NC_e}$; otherwise when we attempt to add the diagonal scaffold, it might bond on the old column instead of the new one. We must also ensure that $\boxed{NCD_e}$ is at the top of the new column. To facilitate this swap without risking an unintended tile placement, we place a new tile species $\boxed{NC_t}$ as a temporary cap on the new column. At the end of this process, all tile positions to the left of the new column correctly match the LST, with no excess scaffold. Since we repeat this process iteratively at farther and farther positions, we are strongly constructing the LST.

A.5 Constructing the Upper Symmetric Triangle

We have shown a construction of the lower symmetric triangle (LST) of the DST. We can construct the upper symmetric triangle (UST) at the same time using a very similar mechanism. There is no need to calculate the 3×3 grid for the UST, as we already know its symmetric version for the LST.

We duplicate the tileset that we used to construct the LST so that there is no tile placement ambiguity between symmetric halves. With a few exceptions, discussed below, whenever we add or remove a vertical scaffold or structure tile in the LST, we also add or remove the symmetric tile in the UST. Also, since the horizontal scaffold bonds onto $\boxed{DA}$, we must replace $\boxed{DA}$ with $\boxed{DAC}$ at the end of the outer loop.

The diagonal scaffold is not entirely symmetric across the diagonal axis, so we must make several adjustments. First, since the diagonal scaffold occupies the spaces where $\boxed{SC_0}$ would go in the UST, we do not add a symmetric $\boxed{SC_0}$; we attach the horizontal scaffold directly onto the diagonal scaffold. We rely on the $\boxed{SC_0}$ tile in the LST to inform horizontal scaffold removal. Second, every power-of-two row in the UST intersects with the diagonal at one grid point; we fill in that grid point manually every time we clean up a section of the diagonal scaffold.

With these modifications, our ALCH program strongly constructs the full DST in the CRN-TAM.