PRESENTERS
Sonia Moreno, B.S. 2019
Hugh D. Potter, Ph.D. student
Narun K. Raman, B.S. student
Matthew R. Riley, M.S. student

ALCH: An Imperative Language for the CRN-TAM

Titus H. Klinge[1], James I. Lathrop[2], Sonia Moreno[3], Hugh D. Potter[2], Narun K. Raman[3], Matthew R. Riley[2]

Drake University[1], Iowa State University[2], Carleton College[3]

Overview

We introduce ALCH, an imperative language for describing programs in the CRN-controlled tile assembly model (CRN-TAM), as well as an ALCH compiler and simulator. ALCH supports many of the features of the C programming language and contains a nondeterministic "branching" structure that allows us to query assemblies as they are built.

We also present a strict construction of the discrete Sierpinski triangle (DST) in the CRN-TAM; this is known to be impossible in the aTAM. ALCH allows us to describe the DST construction at a high-level, thus allowing us to reason at the level of algorithms rather than signals and reactions.

The CRN-TAM Model

In 2015 Shiefer and Winfree introduced the chemical reaction network-controlled tile self-assembly model, or CRN-TAM, to investigate interactions between non-local chemical signals and self-assembly systems.

A CRN-TAM program is defined by a finite sets of chemical signals, tiles, and reactions. These reactions can act upon both signals and tiles.

- When a tile **attaches** to the assembly it releases its removal signal into the solution.
- A tiles removal signal can **remove** it from an assembly, if it is bonded at τ strength.

The ALCH Language

ALCH is similar to a subset of the C programming language, and supports the following operations:

- global Boolean variables with assignment and logical operators
- while loops and conditional evaluation
- tile addition/removal and assembly activation/deactivation
- nondeterministic "branching" to query assemblies (see below)

ALCH does not support function calls; the call stack would require unbounded information storage. We also have not implemented numeric or compound datatypes.

Code Sample

```
if(resetDiag)// Clean up diagonal.
{
  resetDiag = false;
  remove DBA;
  remove DAC;
  while(// Loop until we can remove DS.
    branch
    {
      true()
      {
        remove DBC;
      }
      false()
        remove DS;
      }
    }
  )
  {
    remove DAC;
  }
//...
```

Basic Techniques

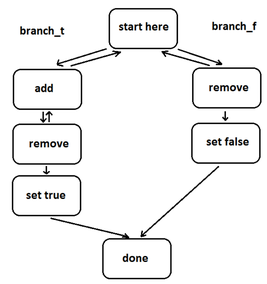
- **Sequential execution:** We control execution with "line number species" $\{X_0, X_1, \ldots, X_{n-1}\}$. A reaction with X_i as a reactant and X_j as a product moves execution from line i to line j.
- **Boolean variables:** We use a dual-rail system, where we represent a variable `a` with two species $(a, \overline{a})$.
- **Returning values:** For operators like `&&` (logical AND), our compiler creates a hidden variable to contain the return value.
- **Conditionals and loops:** We can control execution by adding Boolean variables as catalysts to reactions that change the line number species.

Compiled Example: Conditional

`if(a)`	$X_0 + a \to X_1 + a$
`{`	$X_1 \to X_2 + \boxed{t}$
`  add t;`	$X_2 + t^* \to X_5$
`}`	
`else`	$X_0 + \overline{a} \to X_3 + \overline{a}$
`{`	$X_3 \to X_4 + t^*$
`  remove t;`	$X_4 + \boxed{t} \to X_5$
`}`	

Branching and Assembly Queries

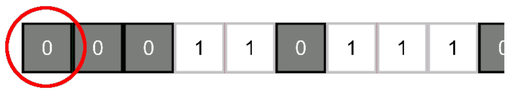
Sometimes we want to know which tiles can be added or removed. The branch construct nondeterministically chooses a "branch path" of **reversible** add/remove commands. If a branch can't complete, execution random walks back to the branch point.



`branch` returns true or false based on which branch path completed.

Compiled Example: Branching

This branching example probes the top of a stack by attempting to remove both a zero and a one tile.

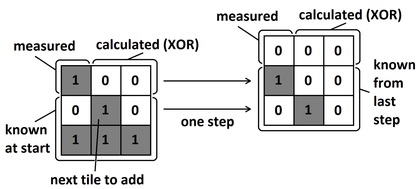


`R = branch`	
`{`	
`  false()`	$X_0 \leftrightarrow X_{1f} + 0^*$
`  {`	$X_{1f} + \boxed{0} \leftrightarrow X_{2f}$
`    remove zeroTile;`	$X_{2f} + R \to X_3 + \overline{R}$
`  }`	$X_{2f} + \overline{R} \to X_3 + \overline{R}$
`  true()`	$X_0 \leftrightarrow X_{1t} + 1^*$
`  {`	$X_{1t} + \boxed{1} \leftrightarrow X_{2t}$
`    remove oneTile;`	$X_{2t} + R \to X_3 + R$
`  }`	$X_{2t} + \overline{R} \to X_3 + R$
`};`	

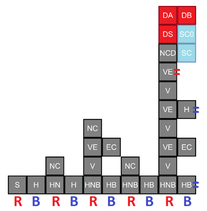
Only the zero branch can successfully complete; when it does, ALCH will set `R` to false to store which branch completed (i.e., which tile popped off of the stack).

Strict Self-Assembly of the DST

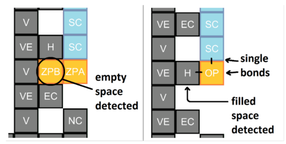
Our construction of the discrete Sierpinski triangle (DST) measures the presence or absence of tiles in a local 3×3 matrix. These values are represented by dual-railed Boolean values and stored as chemical signals in solution. Like the weak construction we use the **XOR** operation to complete the matrix. The matrix is used to determine where tiles should be placed, with true corresponding to a filled cell and false to an empty cell.



We are able to temporarily attach **scaffolding** tiles. These tiles aid in construction of the DST in by allowing us to access specific locations in the assembly and by restricting the size of the frontier.



We are also able to **probe** the assembly to determine where tiles should be placed. We accomplish this with ALCH's branching mechanism. If the probe detects a tile it returns a one to the matrix; if it detects the absence of a tile a zero is returned.



The CRN-TAM is known to be Turing complete, so we could have calculated which tile to add using a full chemical computer. The probe approach allows us to avoid the full machinery of Turing completeness and rely instead on measuring and interpreting local information.

Research supported in part by National Science Foundation grants 1545028 and 1900716..