# ALCH: An Imperative Language for the CRN-TAM

Robyn R. Lutz (PI), Eric R. Henderson, James I. Lathrop, and Jack H. Lutz, Iowa State University

## 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. It has already been shown that the CRN-TAM is as powerful as the Abstract Tile-Assembly (aTAM) and that it is impossible to strongly construct the DST in the aTAM; therefore our construction demonstrates that the CRN-TAM is strictly more powerful than the aTAM. ALCH allows us to describe the DST construction in a convenient high-level form. We can therefore abstract away details of chemical species and reactions and reason at the level of algorithms.

## The CRN-TAM Model

In 2015 Schiefer 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 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 tile's removal can **remove** it from an assembly if it is bonded at strength $\tau$, the temperature.

## The ALCH Language

We have created ALCH, an imperative langage that compiles into CRN-TAM systems. 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 data types.

## ALCH: Basic Techniques

- **Sequential execution:** We control execution with line number species $\{X_0, X_1, \ldots, X_{n-1}\}$. In general, a reaction with $X_i$ as a reactant has $X_{i+1}$ as a product. (Flow control structures link up line number species in different ways.) Since there is exactly one line number species per execution thread present, reactions execute in a controlled sequence.
- **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. We can then "link" that variable to any statement that requires the return value.
- **Conditionals and loops:** We can control execution by adding boolean variables as catalysts to reactions that change the line number species. Since return values are hidden variable species, we can also use logical expressions to govern conditionals.

## ALCH: Branching and Multithreading

Sometimes we want to know which tiles can be added or removed. For example, we might want to know which tile we just popped off of a stack. ALCH contains syntax for "branch points", which diverge into multiple "branch paths" containing reversible tile addition/removal commands.

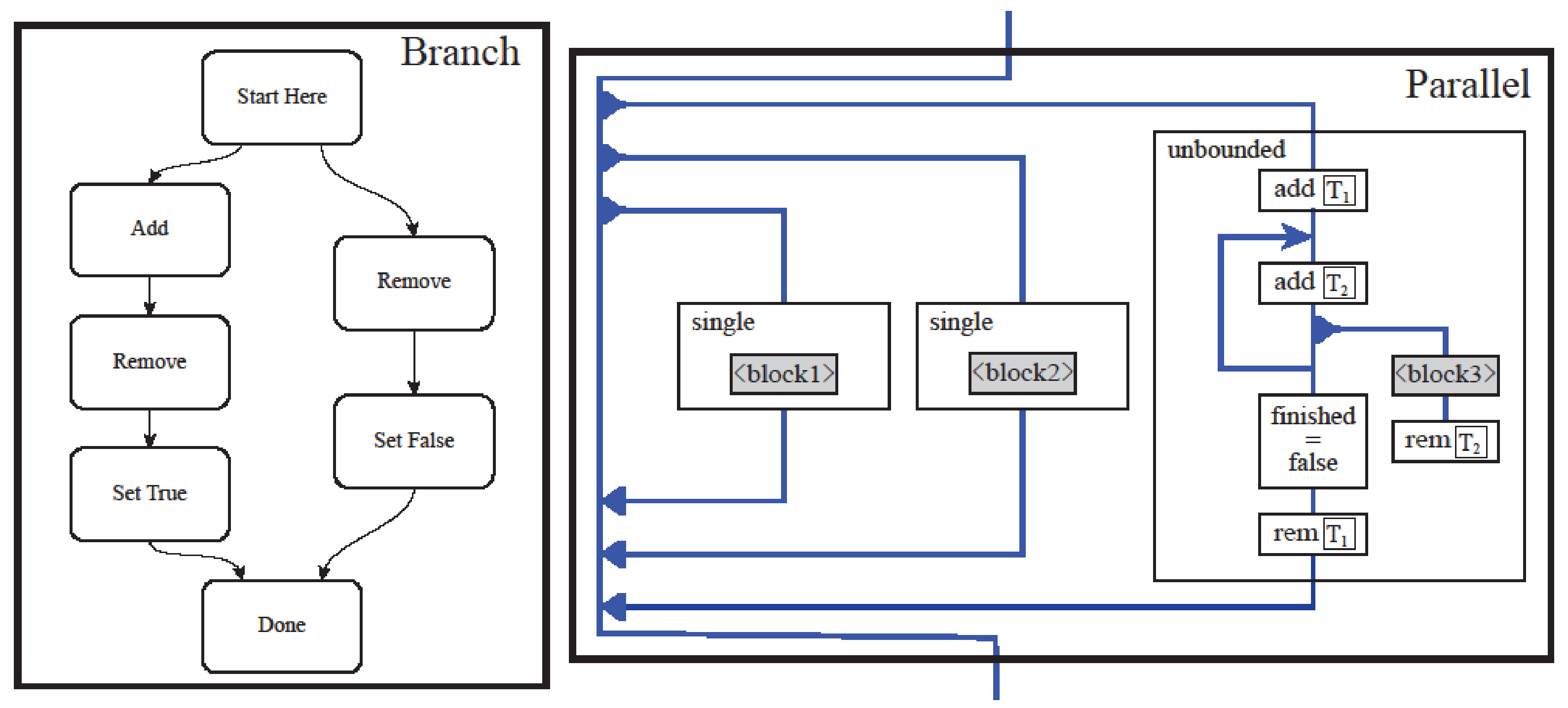


Fig. 1: Execution flow through a branch (left) and a parallel region (right) with two single blocks and one unbounded block. In the unbounded block, ALCH adds and removes tiles to track how many threads are running.

Users can also define multithreaded regions containing code blocks that execute simultaneously, exploiting chemical parallelism. Each block can be "single", where ALCH spawns one thread, or "unbounded", where ALCH continues to spawn threads until receiving the signal to stop. ALCH tracks how many unbounded threads it spawns and blocks until all are cleaned up, so parallel regions are fully modular.

## Strict Assembly of the Discrete Sierpinski Triangle

Our construction of the discrete Sierpinski triangle (DST) measures the presence or absence of tiles in a local $3 \times 3$ matrix, stored by chemical signals in solution. We use the **XOR** characterization of the DST to determine where tiles should be placed.

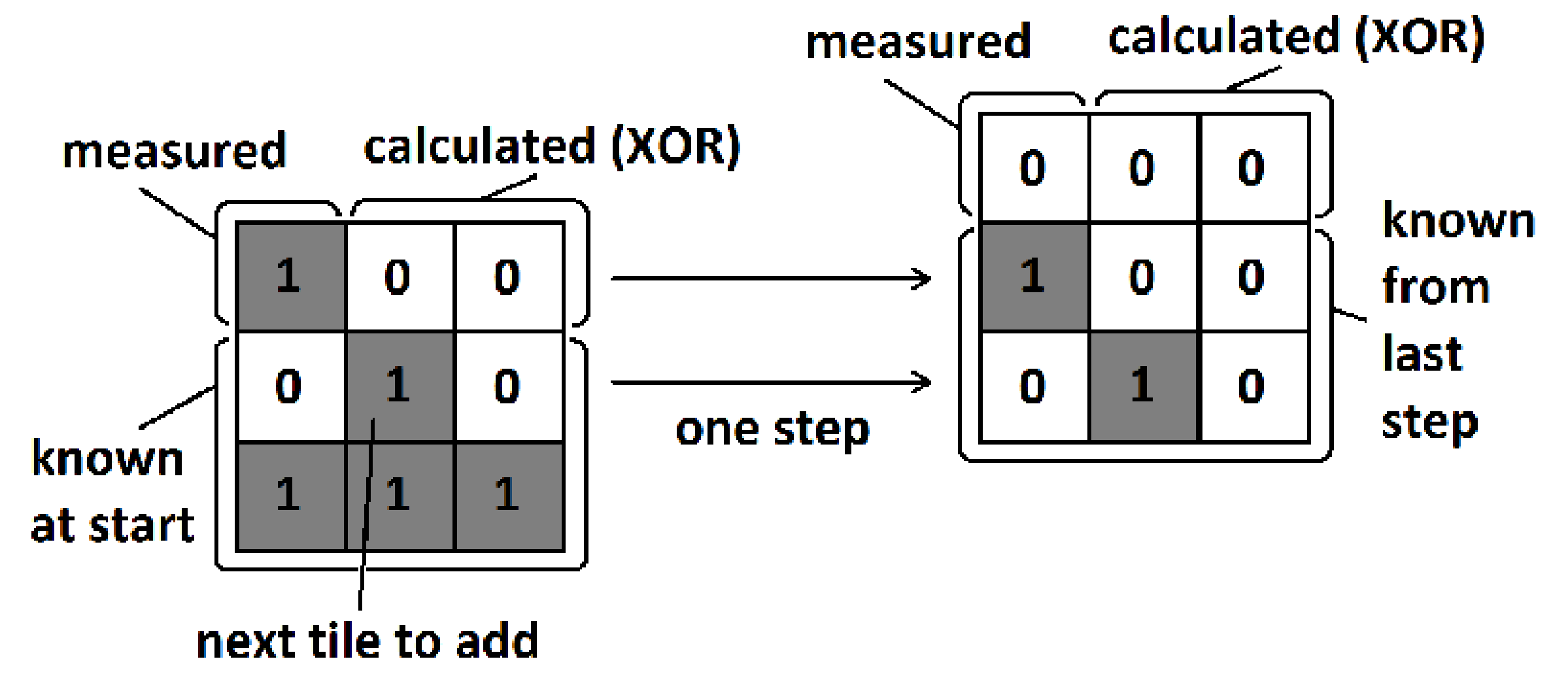
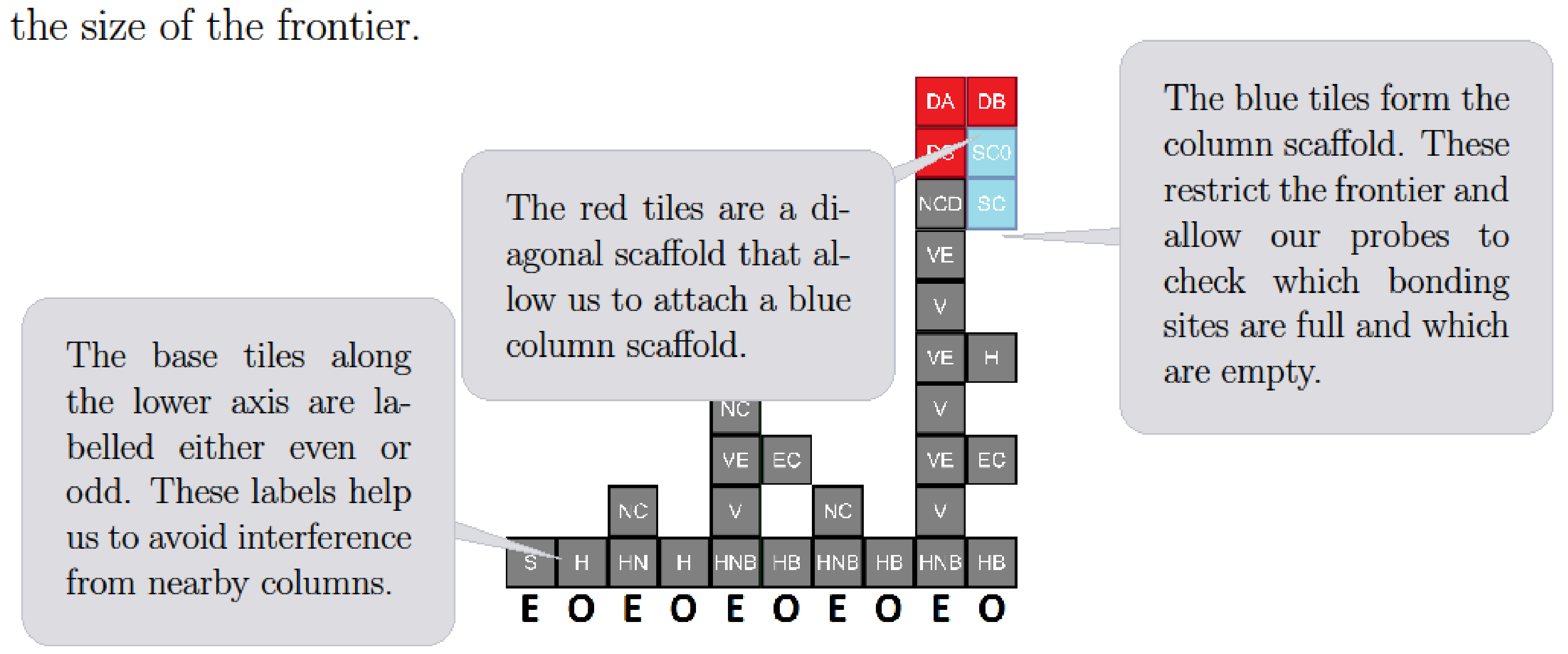


Fig. 2: This figure shows how measured values are used to determine where tiles are placed.

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


The red tiles are a diagonal scaffold that allow us to attach a blue column scaffold.

The blue tiles form the column scaffold. These restrict the frontier and allow our probes to check which bonding sites are full and which are empty.

The base tiles along the lower axis are labelled either even or odd. These labels help us to avoid interference from nearby columns.

We can **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.

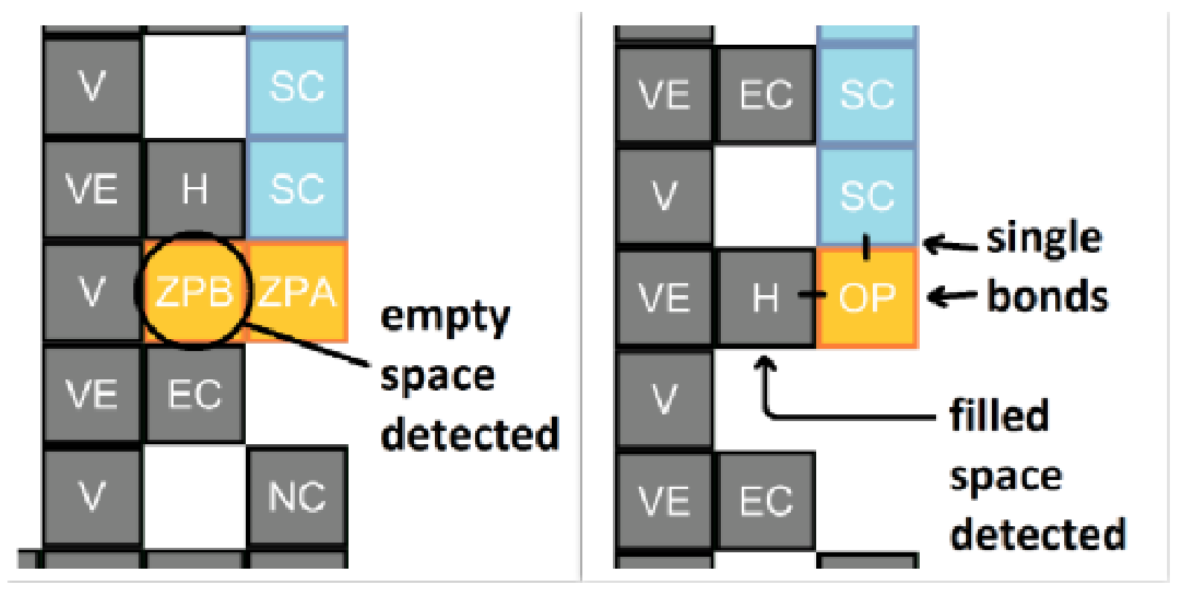


Fig. 4: The left image shows the detection of an empty cell and the right shows the detection of full cell.

Our occlusion and measurement techniques are very general. With several modifications, we have adapted our algorithm to construct the Sierpinski carpet fractal as well.

2021 NSF Cyber-Physical Systems Principal Investigators' Meeting
June 2-4, 2021

Award ID#: 1545028