Where to Begin? Synthesizing Initial Configurations for Cellular Automata

Andrew Wagner

May 2020

1 Introduction

Cellular automata are configurable dynamical systems that, in spite of their simple descriptions, tend to exhibit remarkably complex behavior. The choice of initial configuration plays a large role in realizing that behavior, and much of the work on cellular automata has concerned either characterizing automaton behavior under varying configurations [30] or searching for configurations that evolve in a particular way [20]. In this paper we present a flexible Alloy [9] specification for exploring cellular automata and demonstrate its suitability for formulating a variety of problems from the literature. We offer a collection of configuration synthesis problems on which this specification performs reasonably well in Alloy. We then identify limitations to this approach and evaluate related alternatives.

2 Background

Terminology varies widely across the cellular automata literature, so in Section 2.1 we ground the vocabulary used throughout. Section 2.2 offers a primer on one of Alloy's less popular features that is used heavily in the developed specification.

2.1 Cellular Automata

A *cell* is a finite state machine with a transition function possibly parameterized by the states of some other cells. This transition function is called the cell's *local step*, the set of cells by which it is parameterized is called the cell's *neighborhood*, and the relative positions of those neighboring cells is called the cell's *neighborhood index*[1] [29].

[1]The neighborhood index may also be viewed as a procedure that consumes a cell and produces its neighborhood.

A *cellular automaton* is a state machine that networks cells together in accordance with their neighborhood indices and lifts their local steps into a *global step* transition function. The term "state" is reserved for cell states, while the states of a cellular automaton are called its *configurations*. A configuration is total function mapping each cell to one of its states. A *trace* (sometimes called a history) of an automaton is a total function mapping each *time* to a configuration.

Finite cellular automata, which are the focus of this paper, are cellular automata in which there are a finite number of cells. The *boundary condition* of a finite automaton is a rule that populates the neighborhoods of the cells on the edge of the network. For cells arranged in an array, one popular boundary condition is the *periodic boundary*, under which the limits of the array wrap around.[2] Another common choice is the *null boundary*, under which the limits of the array are assigned to and remain in some special state, often called the *quiescent*[3] state [24].

Traditionally, cellular automata have the following properties:

1. **Behavioral homogeneity.** All cells share the same local step[4].
2. **Spatial homogeneity.** The network is regular. Most commonly, the automaton arranges the cells in an array or grid.
3. **Synchronicity.** The global step applies each cell's local step at every (discrete) moment in time.
4. **Determinism.** The local and global steps are fully deterministic.

Example 2.1. Conway's Game of Life is a cellular automaton with all of the traditional characteristics. Its cells are arranged on a two-dimensional grid and each shares the same binary state set, usually written as $\{0, 1\}$ or $\{\textsc{Dead}, \textsc{Alive}\}$. The neighborhood of a cell is defined by the *Moore* index, under which the eight cells surrounding it at radius 1 are neighbors. For a finite Life automaton with periodic boundary conditions, the neighborhood of a cell is computed modulo the width and height of the grid. The local step maps a cell to Alive if either it is Alive and exactly 2 or 3 of its neighbors are Alive, or it is Dead and exactly 3 of its neighbors are Alive. Otherwise, the cell is mapped to Dead.

Example 2.2. The elementary cellular automata (ECA) are a family of cellular automata with all of the traditional characteristics. Like those in Game of Life, the cells of an ECA share a binary state set, but they are instead arranged in a one-dimensional array. The neighborhood of a cell is the two cells surrounding it at radius 1. For a finite ECA with periodic boundary conditions, the neighborhood of a cell is computed modulo the width of the array. Every member of the ECA family has a distinct local step function.

[2] On a two-dimensional grid, the periodic boundary condition can be called a *toroidal* approximation.

[3] If an automaton has a 0 or Dead state, it is often used as the quiescent state.

[4] N.B. All cells then share the same state set, too.

2.2 Alloy

The Alloy Analyzer [9] is a lightweight modelling tool based on the bounded model finder Kodkod [27]. The language in which its specifications are written in is called Alloy. This paper assumes familiarity with both the tool and the language, though each code snippet is accompanied by an explanation and especially nuanced items are addressed in detail.

The code makes *extensive* use of one of Alloy's newer features: macros. Macros are defined using `let` syntax in the top level environment. They may consume arguments and are curried by default (i.e., they may be partially applied). They are applied just as functions and predicates are, which is either with square brackets (`f[x]`) or dot syntax (`x.f`). For macros, functions, or predicates of arity greater than 1, a mix of dot and bracket syntax may be used (`f[x, y] = x.f[y]`). Application is not to be confused with the two forms of relational join in Alloy, the right (`A.B`) and left (`B[A]`). In addition to being curried, a major advantage in using macros over functions and predicates is that they may consume functions, predicates, and formulas, none of which are first class in Alloy.

3 A Base Specification

We develop an Alloy specification in which the preceding examples are naturally expressible but one that is robust to abandoning most of the traditional properties. The one property that will remain fixed is spatial homogeneity: any expressible automaton arranges its cells in a two-dimensional grid (though one-dimensional automata like the ECA may be expressed in a grid of height 1, as we will see). The definition of the `grid` utility is given in the appendix.

`Time` is linearly ordered and bounded: there is a `first` and a `last` time; all times except the `last` have a unique successor under `next`; and all times except the `first` have a unique predecessor under `prev`. There are two cyclically ordered dimensions `X` and `Y` for width and height: each unit has a unique successor under `nextw` and a unique predecessor under `prevw`. Cells take on subclasses of `State`.

```
open util/ordering[Time]
open grid -- Provides sigs X and Y.
sig  Time {}
abstract sig State {}
```

For convenience, we alias the following types.

```
let Cell   = X->Y
let Config = Cell->State
let Trace  = Time->Config
```

Note that these types are more permissive than is generally desired. For example, there exist relations of type `Config` that are not functions, and therefore

not meaningful configurations. Appropriate structure is imposed in the body of the specification.

A trace is well-defined with respect to a global step if any configuration in the trace is followed by its image under that global step, excepting only the last configuration, which is followed by nothing. Traces with ill-defined configurations (i.e., non-functions) are not considered to be well-defined.

```
pred cfg_ok[cfg: Config] {
    all x: X, y: Y | one cfg[x][y]
}
let wf[global, trace] {
    -- well-defined configurations
    all t: Time | cfg_ok[trace[t]]
    -- uses global step
    all t: Time - last |
    let cfg = trace[t], cfg' = trace[t.next] |
        global[cfg, cfg']
}
```

With the arrangement of cellular automata fixed to a two-dimensional grid, an automaton is now completely specified by its global step. The global step of a traditional cellular automaton applies the same local rule to each cell at every moment in time.

```
let global_std[local, cfg, cfg'] =
    all x: X, y: Y |
        cfg'[x][y] = cfg.local[x, y]
```

A traditional $m \times n$ automaton with null boundary conditions can be simulated on an $(m+2) \times (n+2)$ non-homogeneous automaton with periodic boundary conditions by applying a constant-quiescent local step to the extreme rows and columns and the standard local step to the center.

```
let global_std_null[quiescent, local, cfg, cfg'] = {
    -- first and last rows quiescent
    (cfg + cfg')[first + last][Y] = quiescent
    -- first and last cols quiescent
    (cfg + cfg')[X][first + last] = quiescent
    -- center is standard
    all x: X - first - last, y: Y - first - last |
        cfg'[X][Y] = cfg.local[x, y]
}
```

Different from a conventional presentation of cellular automata, the global step passes the *entire* configuration to a cell's local step. In this way it is the job of the cell, not the automaton, to determine its neighbors' states. Were the cells arranged in a communicating network with some kind of message passing, this approach would needlessly inflate the number of connections on the network.

Here, there is no such communication overhead, and delegating the determination of the neighborhood to cells affords flexibility in defining, for example, non-homogeneous automata in which the neighborhood indices vary from cell to cell.

4 Conway's Game of Life

Each cell in Game of Life takes on one of two states.

```
one sig Dead, Alive extends State {}
```

The neighborhood of a cell in Game of Life is defined according to the MOORE neighborhood index, which maps a cell to the eight cells surrounding it at radius 1.

```
-- requires both dimens > 2
fun moore[x: X, y: Y]: set Cell {
    (x + x.prevw + x.nextw)
    -> (y + y.prevw + y.nextw)
    - x->y
}
```

`moore` uses `nextw` and `prevw` and therefore imposes periodic boundary conditions. Importantly, `moore` is only well-defined for two-dimensional automata when both dimensions have size greater than 2. When a dimension is of size 2 or a fewer, there exists a cell that is the neighbor of another in at least two distinct ways, but the multiplicity is collapsed when the set is returned. This is verifiable in Alloy: for a two-dimensional automaton, the `moore` neighborhood of any cell should have exactly 8 cells, but running with a dimension of 2 or fewer produces a counterexample.

```
-- no counterexample
check moore_ok {
    all x: X, y: Y | #moore[x, y] = 8
} for 1 Time, 3 X, 3 Y, 5 Int -- NB bitwidth
-- counterexample: (0, 1) has only 5 neighbors
check moore_bad {
    all x: X, y: Y | #moore[x, y] = 8
} for 1 Time, 2 X, 3 Y, 5 Int
```

The number of ALIVE cells in a set is called its *population*. In Life, a DEAD cell is *born* if its neighborhood has population 3, and an ALIVE cell *survives* if its neighborhood has population 2 or 3. All other cells die.

```
-- requires bitwidth >= 3
fun life_local[cfg: Config, x: X, y: Y]: State {
  let is_alive = (cfg[x][y] = Alive),
      neighbors = moore[x, y],
      pop = #{x': X, y': Y |
```

```
        x'->y' in neighbors and cfg[x'][y'] = Alive
      },
      born = !is_alive and pop = 3,
      survived = is_alive and pop in (2 + 3) |
        (born or survived)
        implies Alive
        else Dead
}
```

Note that `+` is set union, not arithmetic addition, so `(2 + 3)` is the set $\{2, 3\}$, not the number 5. A bitwidth of at least 3 is required for `life_local` to be well-defined.

Using the base specification, the local step may be lifted into a standard global step (or its null boundary variant), which in turn is used to specify well-defined traces.

```
let life_global = global_std[life_local]
pred life_trace[trace: Trace] {
    wf[life_global, trace]
}
```

Example 4.1. An arbitrary 5-trace of Life on a 5×5 grid.

```
run life_trace for 5 Time, 5 X, 5 Y, 3 Int
```

Example 4.2. Alloy can verify that Life is deterministic: traces starting from the same initial configuration are identical.

```
check life_deterministic {
    all trace, trace': Trace | {
        trace[first] = trace'[first]
        life_trace[trace]
        life_trace[trace']
    } implies trace = trace'
} for 5 Time, 3 X, 3 Y, 3 Int
```

4.1 Synthesizing Still Lifes

In the Game of Life, a *still life* is a configuration that is its own image under the global step. Typically, still lifes are restricted to nontrivial configurations with at least one ALIVE cell. A still life (and many other patterns) is measured by its population and the size of its bounding box, which takes as its width and height the distance between the horizontally and vertically extreme ALIVE cells, respectively.

Because Life is deterministic, as we verified earlier, there is only one trace from any initial configuration.[5] Synthesizing a still life then amounts to finding

[5]We did not, however, verify that Life is total: that there is a trace from any initial configuration. A discussion on totality can be found in Section 6.

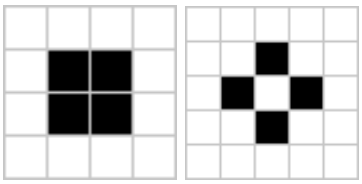

Figure 1: The still lifes with the smallest population (4) are the block (left) and the tub (right). Their bounding boxes are 2×2 and 3×3, respectively [15].

a witnessing 2-trace and taking its first configuration. A constraint on the population can be imposed on the first configuration in the trace, and a bounding box $m \times n$ can be imposed by running with dimensions $(m+2) \times (n+2)$ and a constraint that no border cell is ever ALIVE. Note that this constraint is subtly different from the null boundary condition. In the null boundary condition, border cells are always DEAD by *rule*: they are subjected to the constant-DEAD local step, which keeps them DEAD. Here, border cells are always DEAD by *configuration*: they are subjected to the normal Life local step, but are DEAD by virtue of their neighbors.

```
pred box_bounded[trace: Trace] {
    all t: Time | let cfg = trace[t] | {
        -- first and last cols are always Dead
        cfg[first + last][Y] = Dead
        -- first and last rows are always Dead
        cfg[X][first + last] = Dead
    }
}
let still_life[pop, trace] =
    some trace: Trace |
    let cfg = trace[first] | {
        #cfg.Alive = pop
        all t: Time | trace[t] = cfg
        life_trace[trace]
        box_bounded[trace]
    }
```

Example 4.3. The box is the unique still-life configuration with 4 ALIVE cells and a bounding box of 2×2 [15].

```
run block {
    still_life[4]
} for 2 Time, 4 X, 4 Y, 4 Int
```

Alloy does very well at synthesizing still-lifes, even dense ones with large dimensions (Table 1). The bottleneck in exploring still-lifes in this fashion is not the solver, but the choice of initial population and bounding box. That is, if

cmd	x, y, t	v	pv	c	avg (ms)	stdev (ms)
os 2	5, 5, 3	17940	750	54130	76	8
os 3	11, 8, 4	92470	3520	282850	17710	454
os 4	9, 9, 5	111775	4050	420335	68070	1609
os 5	14, 14, 6	67016	2352	252996	TO	TO
sl 4	4, 4, 2	6300	320	18920	46	8
sl 5	5, 5, 2	9985	500	30540	70	14
sl 10	8, 8, 2	24855	1280	92165	186	32
sl 15	11, 7, 2	30035	1540	111870	191	47
sl 20	9, 9, 2	31700	1620	118000	175	6
sl 25	15, 15, 2	86910	4500	324585	1193	108
sl 30	11, 9, 2	38545	1980	143760	361	33
ss 1, 1, 4	5, 5, 5	32885	1250	126615	365	68
ss 2, 0, 4	7, 6, 5	55115	2100	212575	13713	921
ss 0, 1, 3	18, 7, 6	124370	5040	478010	124751	1898
ss 0, 1, 4	23, 12, 5	71819	2760	277435	TO	TO

Table 1: Game of Life results. In the first column, the `cmd` (command) prefix is either `sl` (still-life) with size, `os` (oscillator) with period, or `ss` (spaceship) with delta-x, delta-y, and inverse speed. In the second column, the `x`, `y`, and `t` are the width, height, and trace length, respectively. In the third column, the `v`, `pv`, and `c` are the variables, primary variables, and columns, respectively. Average time (5 trials) and standard deviation are given in the leftmost columns. All tests run on a Quad-Core Intel Core i7 at 2.6 GHz with 16 GB of RAM.

one knows those parameters a priori, Alloy can find it quickly. However, finding appropriate values for those parameters is non-trivial and, for now, requires some guiding external resource (e.g., [15]).

4.2 Synthesizing Oscillators

An *oscillator* is a configuration on which Life is periodic; that is, the configuration repeats after a some number of steps. In finite Life, any configuration is periodic since the space of possible configurations is finite. More specifically, then, the interest is in configurations with periods that are small relative to some reasonable trace length. With determinism, synthesizing a period-ρ oscillator amounts to synthesizing a witnessing $(\rho + 1)$-trace. The bounding box of an oscillator is taken to be that of its largest configuration.

```
let oscillator[per] =
    let tper = {t: Time | #t.prevs = per} |
    some trace: Trace | let cfg = trace[first] {
        -- repeats at tper
        trace[tper] = cfg
        -- but does not repeat earlier
```

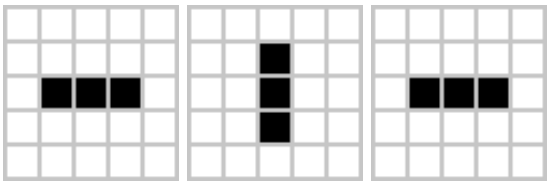

Figure 2: The smallest period-2 oscillator is the blinker [13].

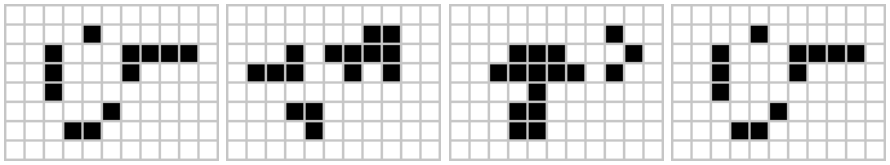

Figure 3: The smallest period-3 oscillator is the caterer [13].

```
        all t': first.nexts & tper.prevs |
            trace[t'] ≠ cfg
        life_trace[trace]
        box_bounded[trace]
    }
```

Example 4.4. The blinker is the unique period-2 oscillator with bounding box 3×3 (up to symmetry) [13].

```
run blinker {
    oscillator[2]
} for 3 Time, 5 X, 5 Y, 4 Int
```

Example 4.5. The caterer is the unique period-3 oscillator with bounding box 9×6 (up to symmetry) [13].

```
run caterer {
    oscillator[3]
} for 4 Time, 11 X, 8 Y, 4 Int
```

Alloy can find the smallest oscillators of periods up to 4 in reasonable time (Table 1), but it times out ($>$ 15 min) when searching for period-5 oscillators. While this is surely due in part to the longer trace length, higher-period oscillators tend to require larger bounding boxes. For example, the smallest period 5 oscillator (the pseudo-barberpole) has a 12×12 bounding box [13].

4.3 Synthesizing Spaceships

A *spaceship* is a configuration that translates over time. The *speed* of a spaceship is taken to be the distance of the translation divided by the number of generations required to travel that distance [14].

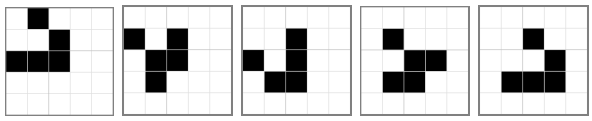

Figure 4: The glider [14].

Because spaceships by definition travel, spaceship synthesis cannot use the box bounded technique from still life and oscillator synthesis, lest it should "crash" into the borders.[6] A first attempt at synthesizing spaceships instead exploits periodic boundary conditions: a spaceship will travel to the border, wrap around, and eventually travel back to where it started.

Example 4.6. The glider (Figure 4) is the smallest spaceship, with a population of 5 and a 3×3 bounding box. It travels diagonally at a speed of $\frac{1}{4}$; that is, it takes 4 generations for every cell to translate by $(\pm 1, \pm 1)$. The glider can be synthesized like an oscillator without box bounding. For example, on a 5×5 grid, it takes 20 steps for the glider to return to its initial position.

```
run glider {
    let tmoved = {t : Time | #t.prevs = 20} |
    some trace: Trace |
    let cfg = trace[first] {
        -- repeats at tmoved
        trace[tmoved] = cfg
        -- but does not repeat earlier
        all t': first.nexts & tmoved.prevs |
            trace[t'] ≠ cfg
        life_trace[trace]
        -- box bound initial config, only
        cfg[first + last][Y] = Dead
        cfg[X][first + last] = Dead
    }
} for 21 Time, 5 X, 5 Y, 6 Int
```

While this technique is reasonable for glider synthesis (consistently finishing within a few seconds), it does not scale well to larger spaceships, which require long trace lengths. For example, the next largest spaceship, the lightweight spaceship (Figure 5), requires a 29-trace to synthesize in this fashion and consistently times out (> 15 min) on a query analogous to Example 4.6.

Instead, using the `grid` utility, spaceship travel can be expressed as a translation, directly.

```
let spaceship[dx, dy, v] =
    some trace: Trace |
```

[6] However, one can and should enforce the bounding box in the initial configuration alone.

Figure 5: The lightweight spaceship [14].

```
  let cfg = trace[first],
      tv = {t: Time | #t.prevs = v},
      cfg' = trace[tv] | {
    life_trace[trace]
    cfg' = translate[cfg, dx, dy]
    some cfg.Alive -- nontrivial
      -- box bound initial config, only
    cfg[first + last][Y] = Dead
    cfg[X][first + last] = Dead
  }
```

Example 4.7. A faster technique for synthesizing a glider. This method significantly reduces the number of variables (×7), the number of clauses (×8), and the total solve time (×10).

```
run glider' {
    spaceship[1, 1, 4]
} for 5 Time, 5 X, 5 Y, 5 Int
```

Example 4.8. The lightweight (Figure 5) spaceship is the next largest spaceship. It travels orthogonally (i.e., in one of the cardinal directions) at a speed of 2 cells every 4 generations.

```
run lwss {
    spaceship[2, 0, 4]
} for 5 Time, 7 X, 6 Y, 5 Int
```

4.4 Variants of Life

The Game of Life is part of a broader family of automata often called the "Life-like" automata [11]. Local steps in this family are conventionally written `B[]S[]` where each hole is filled with a set of integers indicating those neighborhood populations on which cells are **B***orn* and **S***urvive*, respectively. The rule for the Game of Life is written `B2S23`, for example. The formulation for `life_local` can then be generalized as follows.

```
let lifelike_local[B, S, cfg, x, y] {
  let is_alive = (cfg[x][y] = Alive),
      neighbors = moore[x, y],
      live_count = #{x': X, y': Y |
```

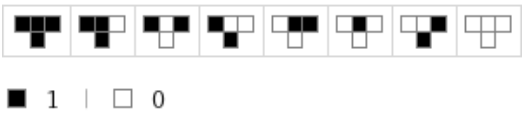

Figure 6: Derivation for Rule 210 (11010010 in binary) [31].

```
        x'->y' in neighbors and cfg[x'][y'] = Alive
      },
      born = !is_alive and live_count in B,
      survived = is_alive and live_count in S |
        (born or survived)
        implies Alive
        else Dead
}
```

Example 4.9. The Game of Life, written in terms of `lifelike_local`.

```
fun life_local[cfg: Config, x: X, y: Y]: State {
    lifelike_local[3, 2 + 3]
}
```

Example 4.10. The Highlife variant, famous for its replicator pattern [12].

```
fun highlife_local[cfg: Config, x: X, y: Y]: State {
    lifelike_local[3 + 6, 2 + 3]
}
```

5 Elementary Cellular Automata

Elementary cellular automata (ECA), like Life-like automata, use a binary state set. Unlike Life-like automata, ECA are one-dimensional, though we will represent them here as height-1 two-dimensional automata.

As mentioned in Example 2.2, there are 256 distinct ECA. The *Wolfram Code* of an ECA is an integer $n \in [0, 256)$ that fully specifies the automaton's behavior [30]. With the 8 state triples sorted in descending order, the 8 digits of the binary expansion of n indicate the image of each under the local step.[7] Figure 6 gives an example derivation for Rule 210.

The Wolfram Code is the most popular encoding of ECA, but Alloy is not well-suited for converting decimal integers to binary. An alternative representation of an ECA considers only its *active* transitions; that is, those state triples on which the local step actually changes the state of the center cell [4]. The 8 active transitions are labeled A through H.

[7] Where 0 is Dead and 1 is Alive.

```
abstract sig Active {
  transition: State->State->State->State
}
one sig A extends Active {} {
    transition = Dead->Dead->Dead->Alive
}
one sig B extends Active {} {
    transition = Dead->Dead->Alive->Alive
}
one sig C extends Active {} {
    transition = Alive->Dead->Dead->Alive
}
one sig D extends Active {} {
    transition = Alive->Dead->Alive->Alive
}
one sig E extends Active {} {
    transition = Dead->Alive->Dead->Dead
}
one sig F extends Active {} {
    transition = Dead->Alive->Alive->Dead
}
one sig G extends Active {} {
    transition = Alive->Alive->Dead->Dead
}
one sig H extends Active {} {
    transition = Alive->Alive->Alive->Dead
}
```

Specifying height 1 in the run statement is sufficient to realize a one-dimensional neighborhood, but to afford flexibility in, for example, running a one-dimensional and two-dimensional automaton in parallel, the ECA local step will consider only the first row.

```
let otherwise[s, x] = some s implies s else x
let eca_local[active, cfg, x, y] {
    let cfg1d = cfg.flip12[first],
        l = cfg1d[x.prevw],
        c = cfg1d[x],
        r = cfg1d[x.nextw],
        change = active.transition[l][c][r] |
            change.otherwise[c]
}
```

`active.transition` collects all transitions in the active set. Joining with any active triple then yields the state to which it changes, and joining with any inactive triple then yields the empty set. `change.otherwise[c]` will return `change` in the former case but default to `c` in the latter. The function

`flip12` is provided by `util/ternary` and it flips the first two columns of a three-column relation like `cfg : X->Y->State`. Lifting the local step into a global step and its corresponding trace predicate is straightforward.

```
let eca_global[active] =
    global_std[eca_local[active]]
let eca_trace[active, trace] {
    wf[eca_global[active], trace]
}
```

Example 5.1. Rule 210 in terms of active transitions.

```
let r210 = (B + C + E + F)
pred r210_trace[trace: Trace] {
    eca_trace[r210, trace]
}
run r210 for 5 Time, 10 X, 1 Y, 4 Int
```

Example 5.2. All ECA are deterministic.

```
check eca_deterministic {
  all active: set Active, trace, trace': Trace | {
    eca_trace[active, trace]
    eca_trace[active, trace']
    trace[first] = trace'[first]
  } implies trace = trace'
} for 5 Time, 10 X, 1 Y, 4 Int
```

There is a subtle but important difference between this check and the deterministic check for Game of Life (Example 4.2). Here, we quantify not only over configurations, but also over automata (via the sets of active transitions that determine their local steps).

6 Introducing Non-determinism

So far, all the automata presented have been deterministic: there is at most one trace from any configuration. We verified this property for the Game of Life in Example 4.2 and for the ECA in Example 5.2. We did not, however, verify that these automata are total: that there actually *is* a trace from any configuration. This is property is *expressible* in Alloy syntax.

```
check life_total {
    all cfg: Config |
        cfg_ok[cfg]
        implies some trace: Trace | {
            life_trace[trace]
            trace[first] = cfg
        }
} for 2 Time, 5 X, 5 Y, 4 Int
```

However, Alloy refuses to verify this property, failing with a message about higher-order quantification. `life_total` desugars into a run statement of the following form, where the result is expected to be unsatisfiable. That is, it searches for a counterexample.

```
run life_total_cex {
    some cfg: Config | {
        cfg_ok[cfg]
        all trace: Trace |
            life_trace[trace]
            implies trace[first] ≠ cfg
    }
} for 2 Time, 5 X, 5 Y, 4 Int expect 0
```

Recall that `Trace` is a type alias for the relational type `Time->X->Y->State`. In Alloy, universal quantification over relations, as in `all trace : Trace`, is rejected syntactically.

"Exists-for all" statements like `life_total_cex` are, however, amenable to a technique called Counter-Example Guided Inductive Synthesis (CEGIS) [25]. CEGIS proceeds by instantiating the `some` with a candidate and then searching for a counterexample to the `all` property. If no counterexample is found, then the candidate witnesses the claim. If a counterexample is found, CEGIS performs a learning step that reduces that set of viable candidates and iterates. This technique is implemented and available for Alloy specifications via the Alloy* Analyzer [18], which succeeds in verifying the totality claim for all the preceding automata.

In a non-deterministic automaton, a configuration may have more than one well-formed trace. Therefore, synthesizing a configuration that realizes a particular property on all its traces is, like `life_total_cex`, an "exist-for all" claim. Since Alloy rejects formulas of this shape, studying non-deterministic automata will require moving to Alloy*. Synthesizing a configuration with property `P` of all its traces can be expressed using the following pattern (where `trace_ok` defines well-formed traces, like `life_trace` or `eca_trace`).

```
let synth[trace_ok, P] =
    some cfg: Config when cfg_ok[cfg] |
    all trace: Trace
        when {
            trace_ok[trace]
            trace[first] = cfg
        } | P[trace]
```

This makes use of Alloy*'s one syntactic addition to the specification language: the `when` clause. For *guarded* quantification statements of the form `all x : D | P[x] implies Q` or `some x: D | P[x] and Q`, rewriting `P[x]` with a `when` clause following the domain `D` helps guide CEGIS. It changes neither the semantics nor the satisfiability of the claim.

size	synth avg (ms)	synth std (ms)	alloy* avg (ms)	alloy* std (ms)
3*	309	210	275	533
4	262	170	92	23
5	698	409	7199	966
6	633	329	206	7
7	1617	466	TO	TO
8	1494	550	TO	TO
10	10969	528	TO	TO
16	48183	609	TO	TO
20	248000	3422	TO	TO
24	711067	95083	TO	TO
26	TO	TO	TO	TO

Table 2: Self-healing configuration synthesis results. `synth` uses algorithm [19]. All boards are square, and `size` indicates side length. Note that there are no self-healing configurations on 3×3, so times are for `UNSAT`. Averages and standard deviations are over 5 trials. All tests run on a Quad-Core Intel Core i7 at 2.6 GHz with 16 GB of RAM.

6.1 Blocking out the noise

Cellular automata are notoriously sensitive to non-deterministic perturbations to their natural evolution, commonly called "noise" [17]. Fault-tolerant cellular automata [5] are highly specialized automata designed to reduce the impact of such noise, but many cellular automata are *ergodic*, meaning that they "forget" all information about their initial configuration when subjected to even trace amounts of noise.

One kind of pattern sometimes discussed in the Game of Life community is the "self-healing" [22] or "self-repairing" [7] object. In such an object, "killing" any one of its members does not result in permanent damage because the original object is immediately restored. Such patterns can then be seen as a basic form of resistance to noise.

For example, the block (Figure 1) is considered to be self-healing: killing any one of its members produces an "L" pattern that in turn evolves back in to the original block. "Forests" of sufficiently dispersed blocks will always be self-healing as well. With periodic boundary conditions, certain tilings of the board can also be self-healing (Figure 7).

If for any damage made to a configuration, the configuration is repaired, then the configuration is self-healing.[8] Here, we take damage to mean killing off a live cell.

```
pred damaged[cfg, cfg': Config] {
    some x: X, y: Y | {
        -- target a live cell
```

[8]Here, we restrict attention to the Game of Life. However, `self_healing` can be abstracted over a `trace_ok` predicate as a macro.

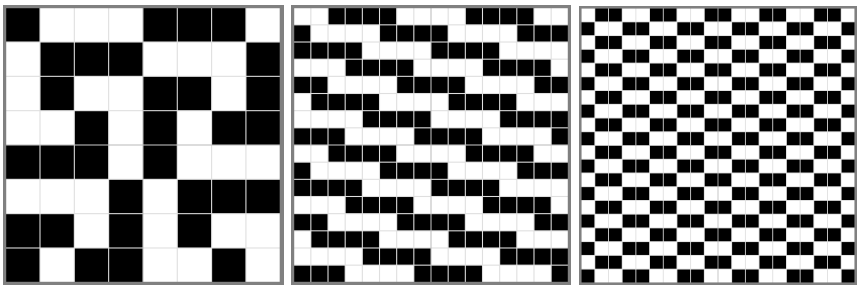

Figure 7: Self-healing configurations that exploit periodic boundary conditions, producing a tiling effect. All were found using the configuration synthesis tool [19] and rendered using Golly [28].

```
            cfg[x][y] = Alive
            -- and kill it
            cfg'[x][y] = Dead
            -- leave all others unchanged
            all x': X - x, y': Y - y |
                cfg'[x][y] = cfg[x][y]
        }
    }
    pred self_healing[cfg: Config] {
        all trace: Trace when {
            life_trace[trace]
            damaged[cfg, trace[first]]
        } | trace[first.next] = cfg -- repair
        cfg_ok[cfg]
        some cfg.Alive -- nontrivial
    }
```

Alloy* does not, however, scale well on this problem (Table 2). It can find self-healing configurations for boards of 5×5 or smaller in only a few seconds, but it consistently times out (> 15 min) on larger dimensions. To make explicit the scale of the search problem, there are $2^{m \times n}$ ways to fill a board of size $m \times n$ with binary states, which is the size of the candidate space. Verification of a candidate configuration is a search subproblem in which the goal is to find a counterexample trace on which an ALIVE cell is killed but the configuration cannot repair itself. For a candidate configuration with k ALIVE cells, there are k traces (one for each potential target), which is the size of the search space for the subproblem.

6.2 Configuration Synthesis: A Promising Alternative

Nelson et. al. introduced in [19] a CEGIS-like algorithm specialized for synthesizing configurations of systems that may modify the configuration online.

Configurations are synthesized with respect to a set of safety properties that must hold at every time on any trace. Like standard CEGIS, the algorithm first finds a candidate configuration and then attempts to find a counterexample, which in this case is a trace on which the system fails a safety property. The algorithm halts successfully if no such counterexample is found.

When a counterexample is found, the algorithm departs from standard CEGIS by exploiting the trace structure of the problem. It first determines a *proximate cause*, a property that only holds of counterexample traces. It then determines a sequence of *enabling causes* with which it can learn a *root cause*, a property of initial configurations that entails the proximate cause. These new steps are designed to exclude more configurations (candidates) on each iteration than would standard CEGIS.

With an experimental variant of this algorithm, preliminary results (Table 2) indicate that it is better suited for the self-healing configuration synthesis task presented in the preceding section. Whereas Alloy* cannot find self-healing configurations at size 7×7 or higher, the configuration synthesis tool can find such configurations at sizes as large as 24×24 within the 15 min time limit. One caveat is that the experimental tool does not consume Alloy(*) specifications, so there is a substantial difference in problem formulation. Still, the results are promising and we expect that this task will serve as a useful test-bed for evaluating the performance of the tool.

7 Related and Future Work

7.1 Search and Synthesis in the Game of Life

"Glider synthesis" is a technique for finding interesting patterns in the "debris" of colliding gliders [20]. Another search technique called "Soup search" starts with a random configuration (a "soup") and collects interesting patterns found during its evolution [6, 10]. The GridWalker algorithm [2] is a SAT-based algorithm specialized for finding interesting initial patterns in the Game of Life. Its search is designed to maximally exploit structure in the Life formulation, but it is not readily generalizable to other cellular automata.

7.2 Non-determinism in Cellular Automata

There are variety of ways to introduce non-determinism to cellular automata beyond the "noise" interpretation given in Section 6.

Probabilistic cellular automata [16] have local steps that are weighted under some probability distribution. They are popular not only in mathematics and computer science, but in natural sciences as well, as they are useful for simulating, for example, disease spread [8]. Behavior of these automata are usually analyzed in *expectation*, which speaks to the *proportion* of traces realizing some property. Neither Alloy* nor the configuration synthesis tool are well-suited for this type of analysis.

Flexible cellular automata [29] have cells that randomly draw a local step function from a fixed set at each time step. Such an automaton is expressible in the Alloy specification presented in this paper, but because functions are not first class, we cannot directly express a set of local steps. One option is define a specialized global step for a particular local step combination.

```
let global_shift_eca[cfg, cfg'] =
    all x: X, y: Y |
    let w170 = eca_local[r170, cfg, x, y],
        w204 = eca_local[r204, cfg, x, y],
        w240 = eca_local[r240, cfg, x, y] |
            cfg'[x][y] in (w170 + w204 + w240)
```

Another option is to defunctionalize [23] the local steps.

```
abstract sig Tag {}
one sig R170, R204, R240 extends Tag {}
fun apply[tag: Tag, cfg: Config, x: X, y: Y]: State {
    -- dispatch on the tag
    (tag = R170)
    implies eca_local[r170, cfg, x, y]
    else (tag = R204)
    implies eca_local[r204, cfg, x, y]
    else (tag = R240)
    implies eca_local[r240, cfg, x, y]
    -- danger zone
    else none
}
let global_flex[tags, cfg, cfg'] =
    all x: X, y: Y |
    some tag: tags |
        cfg'[x][y] = apply[tag, cfg, x, y]
```

Asynchronous cellular automata [4] do not step all of their cells at every moment in time. One form of asynchronism selects at random a subset of cells to step, leaving all others unchanged. This is readily expressible as a new global step.

```
let global_async[count, local, cfg, cfg'] =
    some updating: set Cell when #updating = count |
        all x: X, y: Y |
            x->y in updating
            implies cfg'[x][y] = cfg.local[x, y]
            else cfg'[x][y] = cfg[x][y]
```

Schrödinger's Game of Life [22] is a Life variant in which cells, in addition to being ALIVE or DEAD, can be maybe-ALIVE/maybe-DEAD, abbreviated as CAT. In the local step, if there are k CATs in the neighborhood (including the focused cell), then there are 2^k cases that must be accounted for. By considering

the maximum and minimum ALIVE population among those possibilities, a deterministic, closed-form can automaton be derived. The translation resembles the powerset transformation of NFAs to DFAs.

Here, we present an equivalent view of this game and its formulation in our framework. Take an initial configuration `cfg` that may or may not contain CATs. A trace is well-defined on this configuration if its first configuration `cfg ↪ '` is just like `cfg` except that all and only the CAT cells in `cfg` are replaced by non-CAT cells.

```
one sig Cat extends State {}
pred uncat[cfg, cfg': Config] {
    all x: X, y: Y | {
        cfg[x][y] ≠ Cat
        -- non-Cats are preserved
        implies cfg'[x][y] = cfg[x][y]
        -- Cats step to some non-Cat
        else    cfg'[x][y] in State - Cat
        -- well-defined
        cfg_ok[cfg] and cfg_ok[cfg']
    }
}
```

Notice that this definition does not depend on a particular state set; only CAT is distinguished. Thus, this formulation is not restricted to binary automata.

Traces no longer have any branching points, as all the non-determinism is shifted to the initialization step. That is to say, two traces with a common prefix must in fact be the same trace.

```
pred schrod_life_trace[cfg: Config, trace: Trace] {
    some cfg': Config | {
        uncat[cfg, cfg']
        trace[first] = cfg'
        life_trace[trace] -- or any trace_ok
    }
}
```

In the original presentation, uncertainty at a cell during the evolution of the automaton is collapsed by labelling it as CAT. That is, a cell `x->y` at time `t` is CAT if there exist a pair of traces on which `x->y` has different states at `t`. Otherwise, `x->y` at `t` has the unique state upon which all traces agree. We can phrase this reduction as a synthesis query in Alloy*, though it is unable to solve it.

```
pred collapses[collapsed, trace: Trace] {
  all t: Time, x: X, y: Y |
      collapsed[t][x][y] = trace[t][x][y]
      or collapsed[t][x][y] = Cat
}
```

```
let synth_collapse[trace_ok] =
    some cfg: Config when cfg_ok[cfg] |
    some collapsed: Trace |
    -- collapsed agrees with good traces
    all trace: Trace
        when schrod_life_trace[cfg, trace] |
            collapses[collapsed, trace]
    -- Cats require witnesses of uncertainty
    all t: Time, x: X, y: Y |
        collapsed[t][x][y] = Cat
        implies some trace, trace': Trace when {
                    trace_ok[cfg, trace]
                    trace_ok[cfg, trace']
        } | trace[t][x][y] ≠ trace'[t][x][y]
```

Though this formulation does not yet perform well, it is attractive because it is agnostic to the specifics of the automaton; any automaton can be lifted into a Schrödenger variant.

7.3 Program Synthesis

This paper has focused on synthesizing *configurations* of automata, not automata themselves. A subtle exception was pointed out in Example 5.2, where in verifying determinism for all sets of active transitions, we were effectively quantifying over automata. The desugaring of the assertion is suggestive of another application of synthesis to the study of cellular automata.

```
run eca_deterministic_cex {
  some active: set Active, trace, trace': Trace | {
    eca_trace[active, trace]
    eca_trace[active, trace']
    trace[first] = trace'[first]
    trace ≠ trace'
  }
} for 5 Time, 10 X, 1 Y, 4 Int expect 0
```

This query is expected to be unsatisfiable, but it otherwise has all the makings of synthesizing a *rule set* for cellular automata. There is interest in, for example, finding rules that preserve mass or density of all initial configurations [1], or that erase abnormalities from any initial configuration [3]. Because one must consider all initial configurations, there are again many traces to consider, like in the non-deterministic case. We suspect that cellular automaton description languages like ALPACA—A Language for the Pithy Articulation of Cellular Automata—would be amenable to modern program synthesis techniques in a framework such as Rosette [21, 26].

8 Conclusion

In this paper we have presented a general approach for modelling cellular automata in Alloy (Section 3) and applied it to a number of traditional (Sections 4, 5) and non-traditional (Sections 6, 7.2) automata. We have demonstrated that Alloy is well-suited for synthesizing still-lifes (Section 4.1) and can manage to synthesize small oscillators (Section 4.2) and spaceships (Section 4.3). Where Alloy has limitations (notably, in the space of non-deterministic automata), we have applied related tools that are more specialized to synthesis tasks (Section 6). We found a configuration synthesis algorithm (Section 6.2) to be particularly effective at synthesizing self-healing configurations, which can repair themselves in the presence of limited non-deterministic noise. As an avenue of future work, we hope to broaden the application of synthesis to cellular automata by targeting not only initial configurations, but rule sets as well (Section 7.3).

Appendix: The `grid` utility

`util/ordering` imposes an ordering on an exact-bounded sig via the partial functions `next` and `prev`. However, `prev` is undefined at `first` and `next` is undefined at `last`. We lift `ordering` into `cyclic` by adding relations `nextw` and `prevw` that behave just like `nextw` and `prevw` on all elements except `first` and `last`, on which they wrap around. We also add a `shift` function that performs modular arithmetic on an element using the element domain size as the modulus.

```
module cyclic[exactly elem]
open util/ordering[elem]

let orElse[s, x] = some s implies s else x

fun nextw[e: elem]: elem {
  e.next.orElse[first]
}

fun prevw[e: elem]: elem {
  e.prev.orElse[last]
}

fun shift[e: elem, delta: Int]: elem {
    let mod = #elem,
        pos = #e.prevs,
        -- reduce first to min risk of overflow
        sdelta = rem[delta, mod],
        -- unsign
        udelta = sdelta ≥ 0
                 implies sdelta
                 else plus[sdelta, mod],
        res = rem[plus[pos, udelta], mod] |
            { e' : elem | #e'.prevs = res }
}
```

The `grid` utility opens `cyclic` on two fresh dimension sigs `X` and `Y`. Keeping the dimensions independent from one another allows arbitrary bounding boxes to be conveniently specified in the run statement (examples in Sections 4.1, 4.2, 4.3). The function `translate` maps `shift` over the `X` and `Y` columns of an `X->Y->univ` relation for specified deltas.

```
open cyclic[X] as ox
open cyclic[Y] as oy

sig X {}
sig Y {}
```

```
fun translate[rel: X->Y->univ, dx: Int, dy: Int]
    : X->Y->univ {
        {
            x: X, y: Y, z: univ |
            let x' = x.shift[negate[dx]],
                y' = y.shift[negate[dy] |
                    z in rel[x'][y']
        }
}
```

References

[1] Boccara, N., and Fukś, H. Number-conserving cellular automaton rules. *Fundamenta Informaticae 52*, 1-3 (2002), 1–13.

[2] Bontes, J. *Searching for patterns in Conway's Game of Life*. PhD thesis, Faculty of Science, 2019.

[3] de Sá, P. G., and Maes, C. The Gacs-Kurdyumov-Levin automaton revisited. *Journal of Statistical Physics 67*, 3-4 (1992), 507–522.

[4] Fates, N., Regnault, D., Schabanel, N., and Thierry, É. Asynchronous behavior of double-quiescent elementary cellular automata. In *Latin American Symposium on Theoretical Informatics* (2006), Springer, pp. 455–466.

[5] Gács, P. Reliable computation with cellular automata. *Journal of Computer and System Sciences 32*, 1 (1986), 15–78.

[6] Goucher, A. P. agpsearch. *Catagolue*. `https://catagolue.appspot.com/syntheses`.

[7] Hutton, T. Self-repairing pattern (discussion thread). `https://www.conwaylife.com/forums/viewtopic.php?f=2&t=979`, 2012.

[8] Ilnytskyi, J., Kozitsky, Y., Ilnytskyi, H., and Haiduchok, O. Stationary states and spatial patterning in ansisepidemiology model with implicit mobility. *Physica A: Statistical Mechanics and its Applications 461* (Nov 2016), 36–45.

[9] Jackson, D. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[10] Johnston, N. The Online Life-Like CA Soup Search (discussion thread). `https://www.conwaylife.com/forums/viewtopic.php?f=7&t=131&p=570`.

[11] LifeWiki. List of Life-like cellular auotmata. `https://www.conwaylife.com/wiki/List_of_Life-like_cellular_automata`.

[12] LifeWiki. OCA:HighLife. `https://www.conwaylife.com/wiki/OCA:HighLife`.

[13] LifeWiki. Oscillator. `https://www.conwaylife.com/wiki/Oscillator`.

[14] LifeWiki. Spaceship. `https://www.conwaylife.com/wiki/Spaceship`.

[15] LifeWiki. Still life. `https://www.conwaylife.com/wiki/Still_life`.

[16] Mairesse, J., and Marcovici, I. Around probabilistic cellular automata. *Theoretical Computer Science 559* (2014), 42–72.

[17] Marcovici, I., Sablik, M., and Taati, S. Ergodicity of some classes of cellular automata subject to noise. *Electronic Journal of Probability 24*, 0 (2019).

[18] Milicevic, A., Near, J. P., Kang, E., and Jackson, D. Alloy*: A general-purpose higher-order relationalconstraint solver. In *International Conference on Software Engineering* (2015).

[19] Nelson, T., Danas, N., Giannakopoulos, T., and Krishnamurthi, S. Synthesizing mutable configurations: Setting up systems for success. In *Workshop on Software Engineering for Infrastructure and Configuration Code* (2019).

[20] Niemiec, M. D. Synthesis of complex life objects from gliders. *New Constructions in Cellular Automata* (2003), 55.

[21] Pressey, C. ALPACA. *Cat's Eye Technologies* (1998). `https://catseye.tc/distribution/ALPACA_distribution`.

[22] Pressey, C. Schrödinger's game of life. *Cat's Eye Technologies* (2015). `https://catseye.tc/distribution/Schr%C3%B6dinger's_Game_of_Life_distribution`.

[23] Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2* (1972), pp. 717–740.

[24] Sarkar, P. A brief history of cellular automata. *ACM Comput. Surv. 32*, 1 (Mar. 2000), 80–107.

[25] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A., and Saraswat, V. A. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006* (2006), pp. 404–415.

[26] Torlak, E., and Bodik, R. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (2013), pp. 135–152.

[27] Torlak, E., and Jackson, D. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2007), Springer, pp. 632–647.

[28] Trevorrow, A., Rokicki, T., et al. Golly: open source, cross-platform application for exploring conways game of life and other cellular automata. `http://golly.sourceforge.net`, 2009.

[29] White, P. E., and Butler, J. T. Synthesis of one-dimensional binary scope-2 flexible cellular systems from initial final configuration pairs. *Information and Control 46*, 3 (1980), 241–256.

[30] Wolfram, S. Statistical mechanics of cellular automata. *Reviews of Modern Physics* (1983).

[31] Wolfram Alpha LLC, 2018. WolframAlpha["rule 210"]. `https://www.wolframalpha.com/input/?i=rule+210`.