



Verified Circuits in Conway's Game of Life

Downloaded from: <https://research.chalmers.se>, 2026-05-19 14:02 UTC


Citation for the original published paper (version of record):

Myreen, M., Carneiro, M. (2025). Verified Circuits in Conway's Game of Life. Leibniz International Proceedings in Informatics, LIPIcs, 352. <http://dx.doi.org/10.4230/LIPIcs.ITP.2025.25>

N.B. When citing this work, cite the original published paper.

GOL in GOL in HOL

Verified Circuits in Conway’s Game of Life

Magnus O. Myreen ✉ 

Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Mario Carneiro ✉ 

Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Abstract

Conway’s Game of Life (GOL) is a cellular automaton that has captured the interest of hobbyists and mathematicians alike for more than 50 years. The Game of Life is Turing complete, and people have been building increasingly sophisticated constructions within GOL, such as 8-bit displays, Turing machines, and even an implementation of GOL itself. In this paper, we report on a project to build an implementation of GOL within GOL, via logic circuits, fully formally verified within the HOL4 theorem prover. This required a combination of interactive tactic proving, symbolic simulation, and semi-automated forward proof to assemble the components into an infinite circuit which can calculate the next step of the simulation while respecting signal propagation delays. The result is a verified “GOL in GOL compiler” which takes an initial GOL state and returns a mega-cell version of it that can be passed to off-the-shelf GOL simulators, such as Golly. We believe these techniques are also applicable to other cellular automata, as well as for hardware verification which takes into account both the physical configuration of components and wire delays.

2012 ACM Subject Classification Theory of computation → Higher order logic; Theory of computation → Logic and verification; Theory of computation → Computability; Software and its engineering → Formal methods

Keywords and phrases Cellular automata, Higher-order logic, Interactive theorem proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.25

Related Version *Full Version*: <https://arxiv.org/abs/2504.00263>

Supplementary Material *Software*: <https://github.com/CakeML/game-of-life/tree/itp2025>
archived at `swh:1:dir:04ef2c240a8443efea077894f8e88cd3cba0e774`

Funding This work was supported by the Swedish Research Council, grant no. 2021-05165.

Acknowledgements A special thank you to Andreas Lööw for sowing the seeds for this work by pointing the first author to the blog posts by Nicholas Carlini [4].

1 Introduction

Conway’s Game of Life is a cellular automaton invented by John Conway and first publicly described in an interview with the Scientific American in 1970 [14]. It was said to immitate the rise and fall of societies. Since its initial publication, GOL has remained a curiosity among hobbyists and mathematicians for its combination of simplicity and surprisingly chaotic organic look as a model of computation. It has been demonstrated that one can create order in the chaos and build interesting constructions in GOL such as, e.g., simple computers, complete Turing machines, or even simulations of GOL in GOL [15, 19].

In this paper, we build infrastructure for formal reasoning about circuits in GOL and design and verify a circuit in GOL that can simulate GOL itself. However, before we describe our work, we provide the necessary background on GOL.



© Magnus O. Myreen and Mario Carneiro;

licensed under Creative Commons License CC-BY 4.0

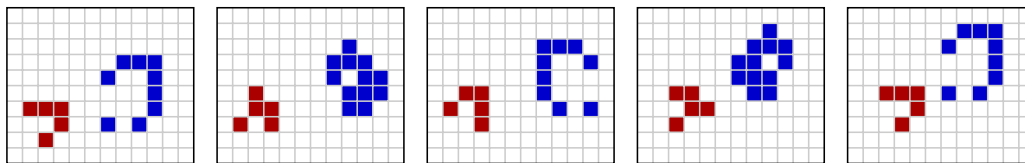
16th International Conference on Interactive Theorem Proving (ITP 2025).

Editors: Yannick Forster and Chantal Keller; Article No. 25; pp. 25:1–25:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Evolution of the glider (red, left) and the lightweight spaceship (LWSS) (blue, right).

1.1 A short introduction to Conway’s Game of Life

Conway’s Game of Life is a deterministic simulation that is performed on an unbounded two-dimensional grid of cells. Each cell can be either alive or dead. Time passes in discrete steps and, at each step, all cells simultaneously update to their next state. The state of a cell at location (i, j) at time $n + 1$ is determined by its state at time n and its immediate neighbours’ states at time n . Cell (i, j) is live at time $n + 1$, if and only if:

- cell (i, j) is live at time n and two or three of its neighbours are live at time n , or
- cell (i, j) is dead at time n and exactly three of its neighbours are live at time n .

The neighbours of a cell (i, j) are the eight cells that are adjacent to it, e.g., $(i, j + 1)$, or share a corner, e.g., $(i + 1, j + 1)$.

The rules of the game do not restrict the initial state, i.e., the state at time 0. The challenge is to find initial states that lead to interesting behaviour when the simulation is run. There are numerous GOL simulators¹ with which people can experiment with different initial configurations of the GOL grid. Anyone who has tried drawing a busy initial pattern in a GOL simulator will have observed that GOL quickly evolves into a chaotic mess that often looks like a digital depiction of the evolution of a bacteria culture.

Gliders and spaceships

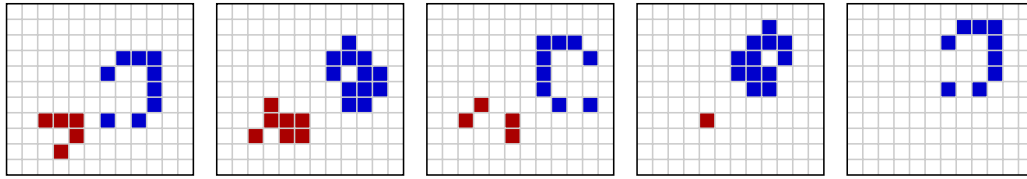
In this world of GOL chaos, there are however certain patterns that are well behaved and can be used in interesting ways. The simplest kind of pattern, called a “still life,” is pattern which stays unchanged on the next clock cycle. These patterns will simply remain static until something else interacts with them. There are also oscillators which go through a sequence of states before returning to the original state.

A slightly more interesting class of patterns are called “spaceships,” which are similar to oscillators but with a spatial shift. The most famous spaceship is called the *glider*. If left undisturbed, it will in four time steps transform itself into exactly its own original shape but shifted one step diagonally. Over time, gliders move across the grid diagonally, at a speed of $1/4$ cells per clock cycle. Another spaceship is the *lightweight spaceship* (LWSS), which move horizontally or vertically at a speed of $1/2$ cells per clock cycle. Figure 1 shows how the glider and LWSS move across the grid.

Useful behaviour through collisions

While gliders and spaceships are cute on their own, their real use comes into focus when we observe that they can carry signals and these signals can be processed via well behaved collisions of (streams of) gliders and spaceships. Collisions can have a variety of effects, including destroying one or both of the incoming spaceships, producing other spaceships,

¹ For example: <https://playgameoflife.com/>



■ **Figure 2** Collision between a glider and an LWSS which destroys the glider but leaves the LWSS intact.

maybe in different directions, and making a big mess of chaotic shapes (we will be deliberately avoiding this kind of collision). Figure 2 shows a collision between a glider and a LWSS which destroys the glider, but the LWSS is unimpeded.

Because gliders are so small, it is not so difficult for them to be spontaneously created, and a key component of our design (and indeed most GOL constructions) utilizes the *Gosper glider gun*², a pattern that produces a glider every 30 ticks. This is what enables us to have steady streams to collide in the first place.

To get things larger than gliders, one can use gliders as a fusion reaction component. Not all kinds of spaceships can be produced in this way, but the LWSS can be produced by three gliders colliding in just the right way, and we will be using this to produce LWSS streams.

If we view a spaceship stream as carrying a timed sequence of bits with a 1 where a ship is present and a 0 when one is absent, then a spaceship stream (a_i) which is intercepted by another stream (b_i) which cleanly destroys both can be seen as computing the logical operation $(a_i \wedge \neg b_i)$, because the output sequence of (a_i) survivors at position i is present only if ship a_i was present, and it was not knocked out by ship b_i . (It also computes $b_i \wedge \neg a_i$ for the (b_i) survivors, but usually we will only be interested in one output.)

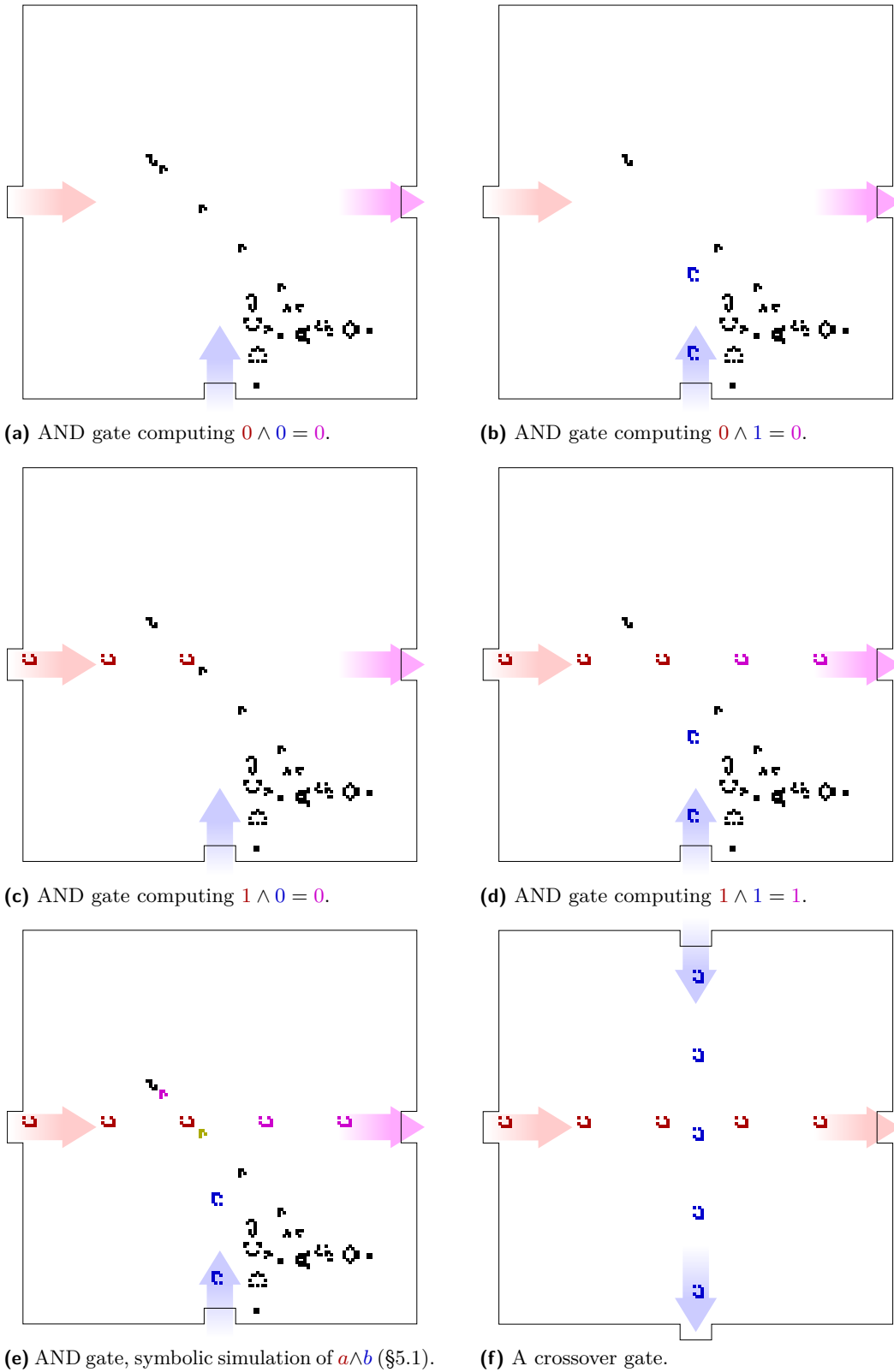
Empty space gives us the constant sequence 0, and the glider gun or its variations give us the sequence 1, so we can already see how we can get a NOT gate as $1 \wedge \neg a$, an AND gate using $a \wedge \neg(1 \wedge \neg b)$, and so we seem to have a complete system of logic gates already. The devil is in the details, but this basic intuition is broadly correct and the rest is engineering.

Nicholas Carlini [4] and others (e.g., [19]) have demonstrated that one can build digital gates based on such collisions. In our work, we use Carlini’s gates and his circuit conventions as a starting point. Figure 3 shows how one of his AND gates works. This particular gate takes in streams of LWSSs travelling *east* and *north*, and essentially computes AND using the formula $a \wedge \neg(\neg b \wedge 1)$ as we described.

An additional aspect of Carlini’s design is that the streams have period 60 instead of period 30. In Figure 3 one can see a second glider gun (left, near the b input) which takes out every other glider produced from the first gun (near the bottom right corner) so that the 1 stream has period 60. The reason for this convention is so that LWSS streams can pass through each other without collision (Figure 3f), which gives us a “crossover gate”, a key tool for building logic circuits in the plane.

In this paper, we describe how one can formally verify circuits in GOL, and in particular, how we have designed and verified a GOL circuit that implements GOL itself. The work we presented here has been carried out in the HOL4 theorem prover [18]. The final product is a verified “GOL in GOL compiler” that, given a GOL configuration, produces a tiling of logic gates implemented in GOL such that the overall behaviour of the circuit is a simulation of GOL starting from the given GOL configuration.

² Named after Bill Gosper who discovered it in 1970 [15].



■ **Figure 3** Illustration of some gates and their data flow behavior. Color key: $a, b, \neg b, a \wedge b$.

2 Mega-Cell and Approach

The technical work for this project had as its goal to build a verified mega-cell as shown in Figure 4. The purpose of this mega-cell is to implement one GOL cell in a space completely tiled with copies of the mega-cell. As the entire grid is tiled with mega-cells performing similar computations, this verified circuit built inside GOL simulates GOL itself. Each of the gates and wires in the diagram are implemented by gates similar to Carlini’s gates.

The circuit in Figure 4 works as follows: at the heart of each mega-cell is a latch, i.e., the little loop highlighted in yellow. This latch holds the state of the GOL cell this mega-cell is simulating. The output of this latch is lead through wires to all neighbouring cells, as can be seen by following the wires from the latch. Because all neighbouring cells have the same circuit wiring, whenever it gives away its value to a neighbor in one of the outgoing arrows from the cell, there is a corresponding input from the opposite side where the neighbor shares its value to this cell, and ultimately the 8 incoming arrows provide the values of all 8 surrounding cells. These inputs are then lead through half-adders (H-A) to sum up the number of neighbours that are alive at the moment.

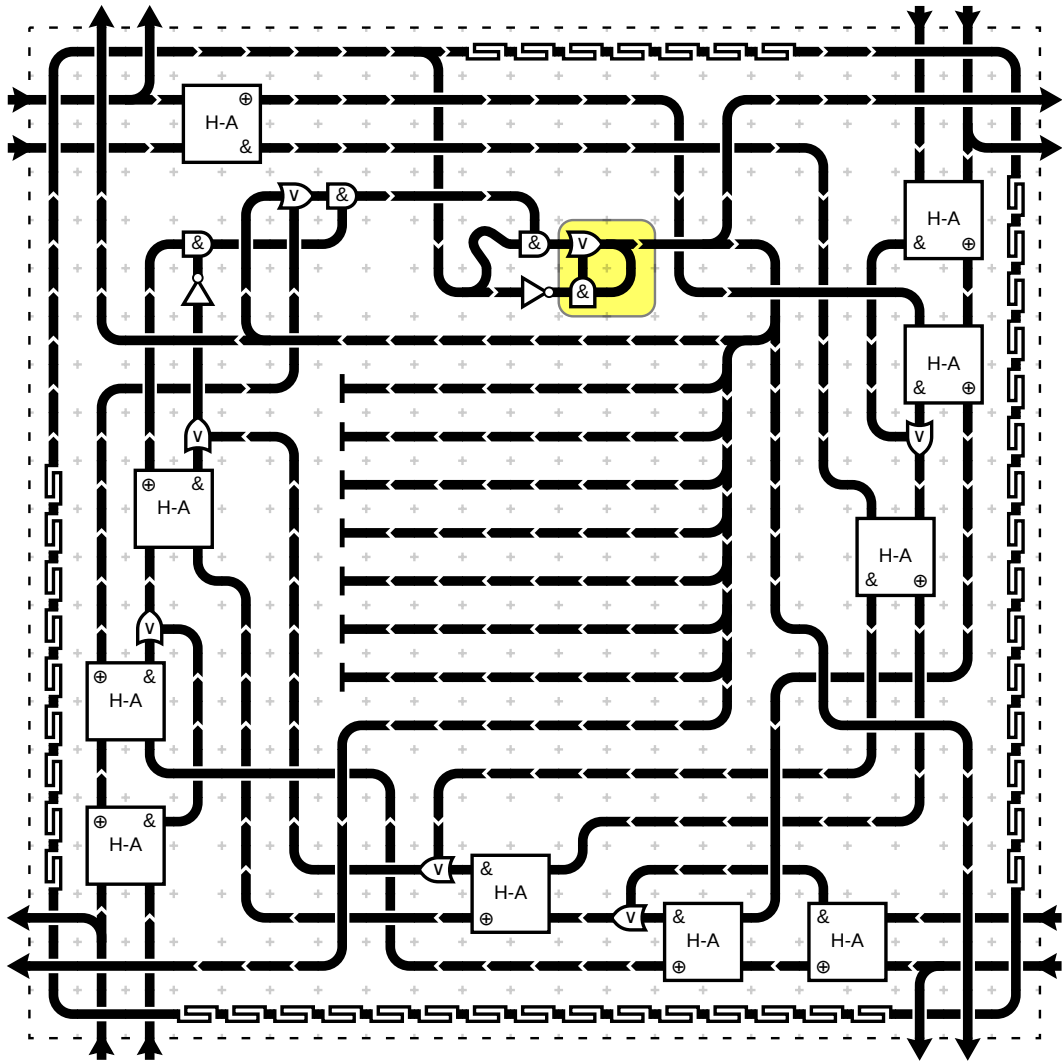
While this computation is being performed, there is also a train of LWSSs cycling slowly on the big loop around the perimeter of the mega-cell. This train of LWSSs is the clock, and it is timed so that just as the value is finished computing, the clock pulse reaches the latch input, and the latch updates its state to the next value this cell is supposed to represent. The twisty wires make the clock signal go slowly enough to not tick the latch forward before the next value is ready.

The mega-cell is laid out on a grid with components which are 1×1 or 2×2 tiles large, and each tile in this grid is 150×150 GOL cells (Figure 3 shows a some of these tiles). The mega-cell is 21×21 tiles large, meaning that the full construction is 3150×3150 GOL cells in size. Proving the correctness of such a large object requires careful design of abstraction layers in order to be manageable.

Our approach consists of several layers of abstraction. The work is organised into the following high-level steps:

1. We formalise the rules of GOL, define the notion of a GOL pattern’s area of influence, and show that two patterns evolve independently if their areas of influence are disjoint.
2. In order to locally prove properties of patterns that communicate with adjacent patterns, we define GOL-IO, an alternative semantics for GOL that allows for input and output. We prove composition and input-output internalisation theorems for these.
3. We then formalise the what it means for a pattern to implement a gate, such as Carlini’s gates. This involves defining exactly how streams of inputs and result in streams of outputs. Again, we prove composition theorems and input-output internalisation.
4. Next, we define the notion of signals that only carry meaningful values some of the time. This is important since variations in wire delay cause signals to “jitter” for a while before settling down to the correct value.
5. We then compose all of the gates in our mega-cell together to obtain a proof that it calculates a specific logic formula, and does so within the time budget provided by the clock signal.
6. Finally, we prove that the formula that is computed is the GOL step function, and that therefore a complete tiling of the space with mega-cells results in the desired GOL simulation.

While many theorems were proved using traditional goal-oriented tactic proofs, some of the heavy lifting was done by automation, including in-logic computation.



■ **Figure 4** A high-level circuit diagram representation of the mega-cell in our construction. There are AND (&), OR (v) and NOT (>) gates, half-adders (H-A), and all of it is acyclic except for the latch, highlighted in yellow, and the clock, which is a wire cycle forming the outer border of the cell. The clock uses slow wires (visualized as switchbacks) in order to ensure that the main logic can complete in time for the next clock cycle.

1. We proved high-level specifications for individual gates using symbolic simulation of the GOL rules.
2. We also automated the composition of all of the gate specifications that make up a mega-cell. This automation computes how and when each wire will have a specific value. Both of these, but particularly the former, made significant use of HOL4's recently added feature for fast kernel computation [1].

3 Formally reasoning about GOL

This section describes how we formalise the rules of GOL and our approach to modular verification of patterns in GOL.

3.1 Rules of GOL

We define a GOL state as a set $S \subseteq \mathbb{Z}^2$, where $(i, j) \in S$ means that (i, j) is alive in state S . As the definition of GOL's next-state function depends on counting the number of live cells neighboring a cell, we use the `live_adj` function to count the number of live neighbours:

$$\begin{aligned} \text{adj } i j &\stackrel{\text{def}}{=} \{(i', j') \mid \max(|i' - i|, |j' - j|) = 1\} \\ \text{live_adj } S \ i \ j &\stackrel{\text{def}}{=} \text{card } (S \cap \text{adj } i j) \end{aligned}$$

Now, given a state S , (i, j) will be alive in the next state if its number of live neighbors is 2 or 3 if (i, j) is live, or exactly 3 if (i, j) is dead.

$$\text{step } S \stackrel{\text{def}}{=} \{(i, j) \mid \text{if } (i, j) \in S \text{ then live_adj } S \ i \ j \in \{2, 3\} \text{ else live_adj } S \ i \ j = 3\}$$

3.2 Area of influence and compositionality

In order to enable modular reasoning about patterns in GOL, we need some notion of non-interference. The intuition we follow is that two patterns in GOL will not interfere with one another as long as they are sufficiently far from each other.

We formalise this intuition by defining a function `infl` which computes the *area of influence* of a GOL state. Location (i, j) is in the area of influence of the patterns in GOL state S if it is at most one step away from a live cell in S .

$$\text{infl } S \stackrel{\text{def}}{=} \{(i', j') \mid \exists i j. (i, j) \in S \wedge \max(|i' - i|, |j' - j|) \leq 1\}$$

Using `infl` we can capture the intuition that, if two patterns s and t are sufficiently far from each other, then they evolve independently of each other in the next step. "Sufficiently far" can be asserted by simply requiring that their areas of influence are disjoint.

$$\vdash \text{infl } S \cap \text{infl } S' = \emptyset \Rightarrow \text{step } (S \cup S') = \text{step } S \cup \text{step } S'$$

Note that, while $\text{infl } S \cap \text{infl } S' = \emptyset$ is a sufficient condition, it is not the weakest condition one can have on that theorem. We chose to use this simple condition because it makes proofs easier than more fine-grained conditions.

3.3 GOL-IO

The patterns in GOL that we set out to verify communicate by sending spaceship patterns to one another. For example, the AND gates shown in Figure 3 receive input from the left and the bottom, and produce output to the right. The input and output consist of LWSSs that they receive or hand over to surrounding patterns in GOL.

In order to verify each gate in isolation, we must therefore sever the links between the gates and replace them by an interface. Because every gate will only interact with the interfaces of its neighbors, the precise details of evolution inside the gate will not matter. The way we express this is through a modified step relation called `io_step`:

$$\begin{aligned} \text{io_step } c \ S_1 \ S_3 &\stackrel{\text{def}}{=} \\ &\exists S_2. \\ &\text{infl } S_1 \subseteq c.\text{area} \wedge \text{step } S_1 = S_2 \wedge \\ &S_2 \cap c.\text{assert_area} = c.\text{assert_content} \wedge \\ &S_3 = c.\text{insertions} \cup (S_2 - c.\text{deletions}) \end{aligned}$$

This relation is functional, but unlike `step` it is not total. It is parameterized by a “modifier” `c` which does several things at once:

- `c.area` provides “guard rails” for the simulation. The initial state must stay within `c.area` and must not touch the interior border.
- `c.assert_area` and `c.assert_content` allow the modifier to assert that a particular pattern appears in the simulation on this step, without otherwise modifying the behavior.
- `c.insertions` and `c.deletions` actually change the state.
 - Inputs can be placed on the board at any time using `c.insertions`.
 - Outputs are cleanly zapped from the state using `c.deletions`.

For most steps, `c.insertions`, `c.deletions`, `c.assert_area` and `c.assert_content` are all empty. However, at time points when an input is supposed to arrive, `c.insertions` will contain an LWSS at an input port. Similarly, output is handled by a combination of `c.deletions`, `c.assert_area` and `c.assert_content` — `c.assert_area` and `c.assert_content` ensure that the expected output was produced, and `c.deletions` removes the output from our local simulation.

An important feature of `io_step` is that matching inputs and outputs cancel out. That is, if `c.assert_content = c.insertions` and `c.assert_area = c.deletions`, then `io_step c S1 S3` implies `step S1 = S3`, i.e., that the insertions and deletions have no effect. This will be relevant later, for the composition theorem.

3.4 GOL-IO runs

To reason about runs consisting of many GOL-IO steps, we define `io_steps k` which performs `k`-steps of `io_step`. Since `io_step` requires a modifier `c`, `io_steps` requires a sequence of modifiers `c : ℕ → modifier`.

$$\begin{aligned} \text{io_steps } 0 \ c \ n \ S_1 \ S_2 &\stackrel{\text{def}}{=} S_1 = S_2 \\ \text{io_steps } (\text{Suc } k) \ c \ n \ S_1 \ S_3 &\stackrel{\text{def}}{=} \exists S_2. \text{io_step } (c \ n) \ S_1 \ S_2 \wedge \text{io_steps } k \ c \ (n + 1) \ S_2 \ S_3 \\ \text{run } c \ S &\stackrel{\text{def}}{=} \forall k. \exists S'. \text{io_steps } k \ c \ 0 \ S \ S' \end{aligned}$$

The `run c S` function asserts that the execution starting from state `S` is able to run indefinitely using and respecting the modifiers in `c`. Note that the assertions in `c` can be used to assert that desired values appear at points of interest in the simulation. It is through these assertions we record the behaviour of the verified circuits we build.

4 Verified Circuits in GOL

This section describes how we write specifications for logic gates built within GOL, the key theorems for working with them, and the abstractions we build to reach the level where we can construct the verified circuit that implements GOL itself.

4.1 GOL-IO runs for circuit components

We state our circuit specifications in terms of `circuit_run` which is defined in terms of `run` from the previous section. The meaning of the parameters, `area`, `ins`, `outs`, `init`, and helper functions, `circ_mod` and `circ_mod_wf`, will be explained later.

$$\text{circuit_run } \textit{area } \textit{ins } \textit{outs } \textit{init} \stackrel{\text{def}}{=} \\ \text{run } (\text{circ_mod } \textit{area } \textit{ins } \textit{outs}) \textit{init} \wedge \\ \text{circ_mod_wf } \textit{area } \textit{ins } \textit{outs}$$

To get a sense of what circuit specifications look like using an example, consider the AND gate from Figure 3. We can prove the following specification theorem:

$$\vdash \text{circuit_run } \{(0, 0)\} \{((-1, 0), \text{E}, a), ((0, 1), \text{N}, b)\} \{((1, 0), \text{E}, a^{[5]} \wedge b^{[5]})\} \text{and_gate_pattern}$$

The coordinates and delays here are in a higher level coordinate system, where 1 unit corresponds to 75 GOL cells (or 1/2 of a tile) and 1 tick of delay corresponds to 60 GOL steps. The components of the specification are as follows:

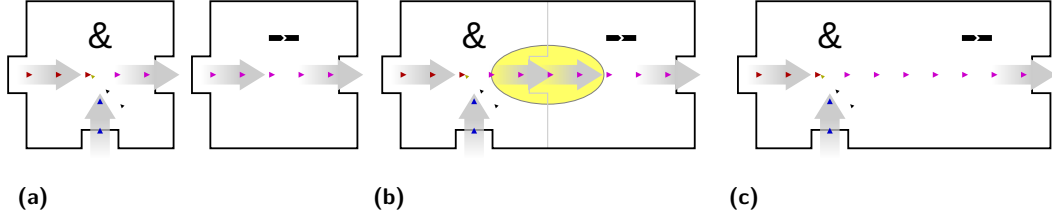
- $\textit{area} := \{(0, 0)\}$ asserts that this circuit uses one full tile centered at position (0, 0). (Because the coordinate system here is in half-tile units, an adjacent gate would be at (2, 0). Gates are always placed at double-even coordinates.)
- $\textit{ins} := \{((-1, 0), \text{E}, a), ((0, 1), \text{N}, b)\}$ states that this gate has two inputs. The first one is at position (-1, 0) (the left edge of the tile), moving east (E) into the tile, and carrying some signal (a_t). Note that signals are functions from natural numbers to booleans, where a_t is the value that arrives into this circuit on tick t . The second input comes from (0, 1), moves north (N), and is carrying signal (b_t).
- $\textit{outs} := \{((1, 0), \text{E}, a^{[5]} \wedge b^{[5]})\}$ states that there is one output stream appearing at location (1, 0), moving east out of the tile and carrying signal $a^{[5]} \wedge b^{[5]}$, where $(a^{[n]})_t \stackrel{\text{def}}{=} t \geq n \wedge a_{t-n}$ delays a signal by n ticks, and $(a \wedge b)_t \stackrel{\text{def}}{=} (a_t \wedge b_t)$ is pointwise AND on signals.
- $\textit{init} := \text{and_gate_pattern}$ specifies that the initial configuration of the GOL cells of this AND gate is the content of `and_gate_pattern`.

4.2 Input and output in GOL circuits

As can be seen in Figure 3, circuit tiles have a square geometry with little ports on the sides through which communication happens. We will now discuss how the IO ports are handled. A gate simulation involves the following stages:

1. The initial state is set up, as in Figure 3.
2. IO ports are included for E/W inputs and N/S outputs, and excluded for N/S inputs and E/W outputs. (See Figure 3f.)
3. 30 GOL steps are performed, during which nothing must escape the bounds.
4. Deletions are performed at N/S output ports.
5. Insertions are performed at N/S input ports.
6. IO ports are included for N/S inputs and E/W outputs, and excluded for E/W inputs and N/S outputs.
7. 30 GOL steps are performed, during which nothing must escape the bounds.
8. Deletions are performed at E/W output ports.
9. Insertions are performed at E/W input ports.
10. Steps 2-9 are repeated for each tick.³

³ For reasons we will get into in section 5.1, because of our use of symbolic evaluation we only actually need to perform steps 2-9 once, but the described gate evolution repeats these steps on each tick.



■ **Figure 5** Composition of an AND gate and a wire. In step (a) \rightarrow (b) we merge the gate areas, resulting in an assembly with an internal input port overlapping an output, highlighted in yellow. In step (b) \rightarrow (c) the matching pair is canceled.

These steps are all expressible through a carefully chosen sequence of GOL-IO modifiers (see Section 3.3).

In other words, for ports going N/S, the IO action scheduled to happen on this tick happens at the end of 30 GOL steps (halfway through the tick), while for E/W ports the action happens after all 60 GOL steps (right at the end of the tick period, before the start of the next tick). The IO action itself is a handoff of an LWSS (or not, depending on the value of the high level signal on that time step) at each output port, and a receipt of an LWSS (or not) at each input port.

The reason for the flip-flopping IO port ownership in steps 2 and 6 is because the producer gate must have ownership of the region prior to the handoff in order to get an LWSS to migrate to that position, and once the deletions and insertions of steps 4 and 5 are performed, the region must be given to the consumer so the LWSS can get out of the port area and into the consumer gate. The phase difference between N/S and E/W ports is to enable crossovers as demonstrated in Figure 3f.

4.3 Composing circuit tiles

Equipped with `circuit_run` and an understanding for how input-output ports work, we now look at how `circuit_run` specifications can be composed.

We use the following theorem when composing two `circuit_run` specifications. The theorem requires the areas owned by these specifications to be disjoint. Furthermore, input (resp. output) port at the edge of one circuit to have a matching output (resp. input) port in the other circuit. Here we overload notation: $(0, 0) + E = (1, 0)$, and $(0, 0) - E = (-1, 0)$.

$$\begin{aligned} \vdash & \text{circuit_run } a_1 \text{ ins}_1 \text{ outs}_1 \text{ init}_1 \wedge \text{circuit_run } a_2 \text{ ins}_2 \text{ outs}_2 \text{ init}_2 \wedge a_1 \cap a_2 = \emptyset \wedge \\ & (\forall p \ d \ r. \\ & ((p, d, r) \in \text{ins}_1 \wedge p - d \in a_2 \Rightarrow (p, d, r) \in \text{outs}_2) \wedge \\ & ((p, d, r) \in \text{outs}_1 \wedge p + d \in a_2 \Rightarrow (p, d, r) \in \text{ins}_2) \wedge \\ & ((p, d, r) \in \text{ins}_2 \wedge p - d \in a_1 \Rightarrow (p, d, r) \in \text{outs}_1) \wedge \\ & ((p, d, r) \in \text{outs}_2 \wedge p + d \in a_1 \Rightarrow (p, d, r) \in \text{ins}_1)) \Rightarrow \\ & \text{circuit_run } (a_1 \cup a_2) (\text{ins}_1 \cup \text{ins}_2) (\text{outs}_1 \cup \text{outs}_2) (\text{init}_1 \cup \text{init}_2) \end{aligned}$$

The conclusion of the composition theorem above unions each component of the two circuits. This means that matching input and output ports then appear both as input and output ports of the resulting `circuit_run` specification (Figure 5b). The following input-output internalization theorem allows us to delete matching IO ports.

$$\begin{aligned} \vdash & \text{circuit_run } \text{area } \text{ins } \text{outs } \text{init} \wedge m \subseteq \text{ins} \wedge m \subseteq \text{outs} \Rightarrow \\ & \text{circuit_run } \text{area } (\text{ins} - m) (\text{outs} - m) \text{init} \end{aligned}$$

To illustrate, suppose we want to compose an AND gate and a wire, as depicted in Figure 5.

1. $\vdash \text{circuit_run } \{(0, 0)\} \{((-1, 0), \text{E}, a), ((0, 1), \text{N}, b)\}$ AND gate spec
 $\{((1, 0), \text{E}, a^{[5]} \wedge b^{[5]}) \text{ and_gate_pattern}$
2. $\vdash \text{circuit_run } \{(0, 0)\} \{((-1, 0), \text{E}, a)\} \{((1, 0), \text{E}, a^{[5]}) \emptyset$ wire spec
3. $\vdash \text{circuit_run } \{(2, 0)\} \{((1, 0), \text{E}, a)\} \{((3, 0), \text{E}, a^{[5]}) \emptyset$ translate (2)
4. $\vdash \text{circuit_run } \{(2, 0)\} \{((1, 0), \text{E}, a^{[5]} \wedge b^{[5]})\}$ substitute (3)
 $\{((3, 0), \text{E}, (a^{[5]} \wedge b^{[5]})^{[5]}) \emptyset$
5. $\vdash \text{circuit_run } \{(0, 0), (2, 0)\}$ compose (1,4)
 $\{((-1, 0), \text{E}, a), ((0, 1), \text{N}, b), ((1, 0), \text{E}, a^{[5]} \wedge b^{[5]})\}$
 $\{((1, 0), \text{E}, a^{[5]} \wedge b^{[5]}), ((3, 0), \text{E}, (a^{[5]} \wedge b^{[5]})^{[5]})\}$
 and_gate_pattern
6. $\vdash \text{circuit_run } \{(0, 0), (2, 0)\} \{((-1, 0), \text{E}, a), ((0, 1), \text{N}, b)\}$ internalize (5)
 $\{((3, 0), \text{E}, a^{[10]} \wedge b^{[10]}) \text{ and_gate_pattern}$

The AND gate specification is familiar from section 4.1. The wire is similar, but it does not need any initial pattern because LWSSs can travel through empty space. We first translate the wire by $(2, 0)$ so it lies next to the AND gate, then substitute a to $a^{[5]} \wedge b^{[5]}$ so that it matches with the output of the AND gate. We can then compose them in step 5, and the redundant input/output pair is cancelled in step 6, with the delay distributing into the expression.

Here we used a binary version of the composition theorem but in our formalization we also prove and use a more general form of the composition theorem which can compose an arbitrary set of circuits. This is used in particular to tile \mathbb{Z}^2 -many copies of a gate.

4.4 Approximate signals

The above-described composition process results in exact descriptions of the stream outputs from a set of gates. However, we run into issues when taking into account delay mismatches where the same signal travels via two paths, resulting in expressions like $a^{[5]} \wedge a^{[6]}$ which we cannot simplify to $a^{[6]}$, even though all we really care about is that the a signal arrives in at most n ticks (in this case, $n = 6$).

In fact, this issue can even arise within a single gate. The half-adder is supposed to calculate the XOR of two signals on one output and AND on the other output. However, the actual circuit theorem we obtain looks like this:

$$\vdash \text{circuit_run } \{(0, 0), (2, 0), (0, 2), (2, 2)\} \{((-1, 0), \text{E}, a), ((-1, 2), \text{E}, b)\}$$

$$\{((3, 0), \text{E}, (a^{[15]} \wedge ((a^{[12]} \wedge \neg b^{[18]}) \vee (\neg a^{[12]} \wedge b^{[15]} \wedge \neg b^{[18]}))) \vee (\neg a^{[15]} \wedge (a^{[12]} \vee b^{[15]}))),$$

$$((3, 2), \text{E}, a^{[17]} \wedge b^{[15]})\}$$

$$\text{half_adder_gate_pattern}$$

If we could erase all the delays from the expression at $(3, 0)$, we would be able to simplify it to simply $a^{[?]} \oplus b^{[?]}$, but because the delays are different it is simply a somewhat arbitrary function on four boolean values $a^{[12]}, a^{[15]}, b^{[15]}, b^{[18]}$. To make matters worse, this self-delay issue compounds as we push through more gates – if we were to feed this expression into another XOR we would get even more mirror copies of the signals.

To resolve this, we weaken our constraints on signals. Rather than specifically asserting that a signal is equal to a given value, each wire is associated to an element in a type `value`, whose denotation is a *set* of possible signals. This amounts to a Hoare-triple-like precondition at each “program point” (= IO port).

We use $s \vdash A$ to assert that $s : \text{stream}$ is in the denotation of $A : \text{value}$, where $\text{stream} \stackrel{\text{def}}{=} \mathbb{Z}^2 \rightarrow \mathbb{N} \rightarrow \text{bool}$. A stream has values $s_z(t)$ saying whether there is an LWSS or not each tick, parameterized over $z \in \mathbb{Z}^2$, which is the mega-cell index.

There are two main kinds of signals, *exact signals* and *approximate signals*. Most wires in the circuit carry approximate signals.

```

avalue = cell( $\mathbb{Z}, \mathbb{Z}$ ) |  $\neg$ avalue | avalue  $\wedge$  avalue | avalue  $\vee$  avalue | avalue  $\oplus$  avalue
evalue = ck |  $\neg$ ck | this | this  $\wedge$  ck | this  $\wedge$   $\neg$ ck
value = avalue[ $\mathbb{N}$ ] | evalue[ $\mathbb{Z}$ ] |  $\top$ 

```

An approximate signal $A^{[n]}$ represents a signal that holds the value $A : \text{avalue}$ after n ticks, and can have any value before that point. Signals have a refinement relation $A \subseteq B \stackrel{\text{def}}{=} \forall s. s \vdash A \Rightarrow s \vdash B$, and we have $m \leq n \implies A^{[m]} \subseteq A^{[n]}$.

Exact signals are much rarer, and they deal with all of the signals that are involved in the latch and clock, where it is important that we do not allow garbage values to enter. In order for exact signals to support a lossless $(-)^{[n]}$ delay operation, an exact signal denotes a $\mathbb{Z} \rightarrow \text{bool}$ stream, that is, one that extends into the past (even though our GOL simulation normally only deals with $\mathbb{N} \rightarrow \text{bool}$ streams). There are two main exact signals, which can be combined in a limited way by logical operators:

- ck, the clock signal, is 1 from tick 0 to 22 (the pulse width), and then 0 from tick 22 to 586 (the clock period), and then it repeats.
- this is a signal which denotes the current mega-cell value $GOL(z, 0)$ for 586 ticks, then $GOL(z, 1)$ (the next time step), and so on.

For example, the wire at the top of the latch in Figure 4 has the value $\text{this}^{[-15]}$, which means that in the initial state it is holding the current value of this cell, and it will continue to hold that value until tick $586 - 15$, at which point it will switch to holding the next value that this cell should have, and it will switch again 586 ticks later.

Approximate signals allow arbitrary boolean combinations of the variables $\text{cell}(m, n)$, which denote the value of a nearby cell $GOL(x + m, y + n, t)$. Note that $\text{this}^{[m]} \subseteq \text{cell}(0, 0)^{[n]}$ provided that $0 \leq m \leq n$, and we perform this “decay” operation early so that most of the gates never have to see an exact signal.

The value \top represents failure and any signal satisfies it; it is used whenever operations go outside the expected bounds. For example, when applying the negation function to *this*, since there is no *evalue* representing \neg *this*, the result is instead \top .

The upshot of this is that we now get much nicer provable gate specifications:

```

 $\vdash$  circuit_run'  $\{(0, 0), (2, 0), (0, 2), (2, 2)\} \{((-1, 0), \text{E}, A), ((-1, 2), \text{E}, B)\}$ 
 $\{((3, 0), \text{E}, A^{[15]} \oplus B^{[18]}), ((3, 2), \text{E}, A^{[17]} \wedge B^{[15]})\}$ 
half_adder_gate_pattern

```

A concern one might have at this point is that while we can now *assert* that things eventually settle down, what about the circuit *ensures* that this actually happens? The idea here is that we start with a budget of 586 ticks worth of stability, and this budget is cut into every time we go through a gate (for example, the above half adder output as much as 18 ticks fewer of stability compared to its inputs). By the time we go through the whole circuit, we have lines that are almost entirely noise but for a few ticks where they have the computed value. The key that lets us turn back the entropy clock is the clock and latch, and this rule (see section 5.2): $\text{ck}^{[m]} \wedge v^{[n]} = (\text{this} \wedge \text{ck})^{[m]}$ provided $n \leq m + 586$, $m \leq -22$, and $v = \text{nextCell}$. Here $v^{[n]}$ will be our computed output holding the *avalue* *nextCell* with only

a few ticks left of juice, and the clock (which is only on for a short period) masks out all the garbage, leaving an exact signal. The latch accepts and retains this value, resulting in a clean signal for the next 586 ticks of computation.

4.5 Building the mega-cell

For the main part of the mega-cell construction, we build up a set of gates, respecting all the inputs and output relations. For this, we will make use of the floodfill function.

The basic idea is that we will build up a state consisting of the following components:

- $area \subseteq 2\mathbb{Z} \times 2\mathbb{Z}$ is a set of points which currently contain a gate or part of a gate. This is to ensure new gates do not overlap existing gates. This set must also stay contained within $[0, 42]^2$ which is a single mega-cell (the extent of Figure 4).
- $ins : (\mathbb{Z}^2 \times dir \times value)$ list is the list of unmatched input ports, and the values that they carry (treated up to permutation, i.e. as a multiset).
- $outs : (\mathbb{Z}^2 \times dir \times value)$ list is the multiset of unmatched output ports.
- $crosses : (\mathbb{Z}^2 \times \mathbb{Z}^2 \times dir)$ list is the multiset of unmatched crossovers, which are treated specially because they do not yet have values associated with them.
- $gates : (\mathbb{Z}^2 \times gate)$ list is the resulting gate list.

The definition of floodfill $area\ ins\ outs\ crosses\ gates$ is somewhat complex, and it is easier to understand it in terms of the theorems it satisfies (simplified for presentation).

- **floodfill_add_ins**: Disjoint gate insertion.⁴ The side conditions assert that $a_1 + p$ is disjoint from a and contained in $[0, 42]^2$. Furthermore, the inputs in i_1 must be exact and must not collide with any other inputs or outputs.

$$\begin{aligned} & \text{floodfill } a\ i\ o\ c\ gs, \quad \text{is_gate } g\ a_1\ i_1\ o_1, \quad (\forall (_, _), v) \in i_1. \text{is_exact } v), \\ & (a_1 + p) \cap a = \emptyset, \quad (a_1 + p) \subseteq [0, 42]^2, \quad (i_1 + p) \cap \text{fst}[i + o] = \emptyset \\ & \vdash \text{floodfill } ((a_1 + p) \# a) ((i_1 + p) \# i) ((o_1 + p) \# o) c ((p, g) :: gs) \end{aligned}$$

- **floodfill_add_gate**: Adding a regular gate. The inputs to the gate must match pre-existing outputs, which are removed from the list.

$$\begin{aligned} & \text{floodfill } a\ i\ ((i_1 + p) \# o) c\ gs, \quad \text{is_gate } g\ a_1\ i_1\ o_1, \\ & (a_1 + p) \cap a = \emptyset, \quad (a_1 + p) \subseteq [0, 42]^2 \\ & \vdash \text{floodfill } ((a_1 + p) \# a) i ((o_1 + p) \# o) c ((p, g) :: gs) \end{aligned}$$

- **floodfill_add_crossover**. Matches one input and puts the other one on the work queue.

$$\begin{aligned} & \text{floodfill } a\ i\ ((ia + p, A) :: o) c\ gs, \quad \text{is_crossover } g\ a_1\ [ia, ib]\ [oa, ob], \\ & (a_1 + p) \cap a = \emptyset, \quad (a_1 + p) \subseteq [0, 42]^2 \\ & \vdash \text{floodfill } ((a_1 + p) \# a) i ((oa + p, A^{[5]}) :: o) ((ib + p, ob + p) :: c) ((p, g) :: gs) \end{aligned}$$

- **floodfill_finish_crossover**. Matches its second input with a pre-existing output.

$$\begin{aligned} & \text{floodfill } a\ i\ ((ib, B) :: o) ((ib, ob) :: c) gs \\ & \vdash \text{floodfill } a\ i\ ((ob, B^{[5]}) :: o) c gs \end{aligned}$$

⁴ The expression $area_1 + p$ is used to mean translating the set $area_1$ by $p : \mathbb{Z}^2$. Similarly, $ins_1 + p$ translates the port positions by p .

- `floodfill_teleport`: “Teleports” an output by a multiple of 42 half-tiles, by reindexing the underlying stream and changing the basepoint p to $p + 42z$ to compensate. This is how the mega-cell communicates with neighboring mega-cells. Here the operation $A + z$ for $A : \text{value}$, $z : \mathbb{Z}^2$ does $\text{cell } p + z = \text{cell } (p + z)$ and distributes over boolean operations.

$$\begin{aligned} & \text{floodfill } a \ i \ ((p, A) :: o) \ c \ gs \\ \vdash & \text{floodfill } a \ i \ ((p + 42z, A + z) :: o) \ c \ gs \end{aligned}$$

These lemmas enable the circuit to be built up by starting from the empty set, disjointly inserting a gate at the clock and at the latch, then using `floodfill_add_gate` to add the rest of the gates, in propagation order. The crossover handling is because `floodfill_add_gate` requires all inputs to a gate to have their values assigned before the gate can be added, but a quick glance at Figure 4 confirms that the wire leading out of the latch doesn’t get very far before having to duck under a crossing wire which has not yet received a value. So in these cases we put it on the crossover queue with `floodfill_add_crossover`, and pop it off with `floodfill_finish_crossover` when we get the value for the other wire.

We will discuss `is_gate` $g \ a_1 \ i_1 \ o_1$ further in section 5.1, but this specification is similar to `circuit_run'` from section 4.4, and in particular it is proved with variables for the input values, so when we use `floodfill_add_gate` repeatedly we build up large expressions on the outputs representing the resulting values.

4.6 Satisfying the floodfill lemmas

The advantage of the lemmas in section 4.5 is that they are easily computable – the expressions `area`, `ins`, `outs`, `crosses`, `gates` are all concrete expressions, which makes it easy to compose the lemmas by applying them and simplifying the results. However, they entail a rather sophisticated logical structure for `floodfill` itself. So in this section we instead look at the definition of `floodfill`, and how it connects to the circuit specifications from section 4.1.

For $s : \text{stream}$ and $v : \text{value}$, let $s \vdash v$ mean that s is in the denotation of v as described in section 4.4. Now `floodfill area ins outs crosses gates` holds if:

- for all $(p, g) \in \text{gates}$, g is valid to be placed at p ; and
- there exist $s_{\text{in}} : \text{stream list}$ and $s_{\text{out}} : \text{stream list}$ such that:
 - $\forall i. (s_{\text{in}})_i \vdash \text{ins}_i$ and $\forall i. (s_{\text{out}})_i \vdash \text{outs}_i$, and
 - for all $s_{\text{cross}} : \text{stream list}$ such that $|s_{\text{cross}}| = |\text{crosses}|$ and $\forall i. \exists v. (s_{\text{cross}})_i \vdash v$, `floodfill_run ins' outs'` holds, where
 - * $\text{ins}' \stackrel{\text{def}}{=} (\text{fst } \text{ins}_i, (s_{\text{in}})_i) \# (\text{in } \text{crosses}_i, (s_{\text{cross}})_i)$
 - * $\text{outs}' \stackrel{\text{def}}{=} (\text{fst } \text{out}_i, (s_{\text{out}})_i) \# (\text{out } \text{crosses}_i, (s_{\text{cross}})_i)$.

This part of the definition handles the delayed assignments to crossovers, and the conversion from value assignments to stream assignments. At the core of it is another definition `floodfill_run ins outs`, where $\text{ins}, \text{outs} : (\mathbb{Z}^2 \times \text{dir} \times \text{stream}) \text{ list}$. Let $S^* = \{p + 42z \mid p \in S, z \in \mathbb{Z}^2\}$. Then `floodfill_run ins outs` holds if:

- $\text{area} \subseteq [0, 42)^2 \cap (2\mathbb{Z} \times 2\mathbb{Z})$; (1)
 - for all $(p, d, _) \in \text{ins}$, $p + d \in \text{area}$; (1)
 - for all $(p, d, _) \in \text{outs}$, $p - d \in \text{area}$; (2)
 - `map f ins` and `map f outs` have no duplicates, where $f(p, d, _) = (\{p\}^*, d)$; and
 - if
 - for all $(p, d, v) \in \text{ins}$, if $p - d \in \text{area}^*$ then $\exists z. (p + 42z, d, v + z) \in \text{outs}$ (3)
 - for all $(p, d, v) \in \text{outs}$, if $p + d \in \text{area}^*$ then $\exists z. (p + 42z, d, v + z) \in \text{ins}$ (4)
- then `circuit_run area* (f ins) (f outs)`, where
- $f \text{ ls} \stackrel{\text{def}}{=} \{(p + 42z, d, v \ z) \mid (p, d, v) \in \text{ls}, z \in \mathbb{Z}^2\}$.

Needless to say, this definition was tricky to discover, and much of the hard work of the formalization was spent showing that all of the lemmas of section 4.5 hold for this definition. The way this was done was to start from the theorems, as they represent the desired computation strategy, and then reasoning out what properties we obtain after combining them. The definition was refined several times in order to have enough to prove the theorems.

One interesting part of the definition above is the precondition (3,4) before `circuit_run`, which means that it is possible for `floodfill` to allow adding a gate even if it has an output which points at a gate with no matching input. Intuitively, this is okay because the *outs* in `floodfill` are effectively proof obligations, and they cannot be discharged without adding another gate on the other side of the output to match it.

Another important observation is that the `circuit_run` expression is over *area**, so it is an infinite circuit, tiled over space. (This is how we are able to satisfy `floodfill_teleport`, which moves an output by a multiple of the mega-cell, which amounts to a reindexing of the lattice of copies of that particular wire.) When we add a gate, we first use the infinitary version of the composition lemma from section 4.3 to duplicate the gate over all of space, then use the binary composition lemma to combine that with the current floodfill area. The preconditions (1,2,3,4) are used to prove the various side conditions in the composition lemma.

5 GOL Circuit Proofs via Computation

5.1 Symbolic evaluation of individual gates

We prove individual gate specifications (in terms of `circuit_run` from Section 4.1) by symbolically simulating the steps a circuit and its IO interfaces cycle through (Section 4.2). That is, rather than simulating concrete GOL states, in our simulator each GOL cell contains a `bexp`, given by the following grammar:

$$\text{bexp} = \top \mid \perp \mid \text{var}_N \mid \neg \text{bexp} \mid \text{bexp} \wedge \text{bexp} \mid \text{bexp} \vee \text{bexp}, \quad \text{var} = A \mid B$$

Our symbolic simulations are set up to take two input streams, *A* and *B*. Variable *A*₃ is the value at index 3 of the input stream *A*. (We call this index the “age” of the variable.)

The goal of the symbolic simulations is to prove circuit specifications for individual gates that describe all the states a gate can be in, including all the states the internal in-progress LWSSs can be in. We do this in two steps:

1. Outside of the proof assistant,⁵ we run several ticks of symbolic simulation starting from a completely concrete initial state but feeding in variables in the places where input LWSSs are to arrive. We are done once these LWSSs have propagated through the entire gate and the simulation reaches a “steady state” where stepping by one tick results in a state equal to the previous state but with every variable aged by one, e.g., *A*₃ becomes *A*₄.
2. Inside the proof assistant, we run one tick of symbolic simulation according to section 4.2 on the symbolic state found outside of the proof assistant, and formally confirm its finding that the resulting symbolic state is equal to the initial symbolic state, aged by one.

The final check shows that gates have stable behaviour over any number of ticks and function like LWSS conveyor belts, with each LWSS making steady progress through the gate.

Figure 3e shows the verified symbolic state of the AND gate, using color to indicate the cells with variable expressions. (Age is not represented in the diagram, but each LWSS in the diagram has a different age. The leftmost LWSS is *A*₄, and the rightmost is *A*₀ \wedge *B*₀.)

⁵ More accurately: in SML code in the script file, but not going through the kernel.

The most interesting aspect of the symbolic simulation is how we determine the next symbolic state of a GOL cell given the current symbolic state of it and its neighbors. The approach we take is to:

1. collect all the variables appearing in any of the relevant symbolic cells,
2. for each possible assignment to those variables, we evaluate the GOL rules,
3. we summarize all results in one expression as a nested if-expression,
4. we simplify the if-expressions and represent it as a single `bexp`.

Step 3 can, for example, result in the following if-expression if a GOL cell and its neighbors use two variables, A_2 and B_1 , and only evaluates to true if both are true.

if A_2 then (if B_1 then true else false) else false

The expression above simplifies to $A_2 \wedge B_1$. This approach is exponential in the local variable count, but this was not a significant issue because our simulation only goes as high as 4 variables in practice (see section 4.4).

5.2 Putting it all together

The floodfill lemmas of section 4.5 are designed so that the main part of the construction can be done fully automatically.

- An SML program takes as input an ASCII-art version of Figure 4.
- It is parsed to get gate positioning and orientation, and the appropriate symbolic evaluation theorem from section 5.1 is selected.
- The initial gates (at the latch and clock), with their values, are additional inputs, and it uses `floodfill_add_ins` to add these.
- It then performs a depth first traversal of the diagram.
 - If an output is facing a gate which has all of its inputs in the output list, use `floodfill_add_gate`.
 - If an output is facing a crossover, use `floodfill_add_crossover`, or `floodfill_finish_crossover` if it is the second time we have visited this gate.
 - If an output is facing the edge of the tile, use `floodfill_teleport` to wrap it back in bounds.
- These steps are repeated until nothing can make progress. In the process we work out all of the formulas associated to each IO port.
- The value type from section 4.4 has functions defined on it for \wedge , \vee , \neg , which do the obvious thing on `avalue` values, but there are a few interesting cases designed to handle the latch area:
 - $\text{this}^{[m]} \wedge (\neg\text{ck})^{[n]} = (\text{this} \wedge \neg\text{ck})^{[n]}$ provided $n \leq m \leq n + 22$
 - $\text{ck}^{[m]} \wedge v^{[n]} = (\text{this} \wedge \text{ck})^{[m]}$ provided $n \leq m + 586$, $m \leq -22$, and $v = \text{nextCell}$
 - $(\text{this} \wedge \text{ck})^{[n]} \vee (\text{this} \wedge \neg\text{ck})^{[n]} = \text{this}^{[n]}$

Here `nextCell : avalue` is the specific formula that the mega-cell circuit computes.

- `nextCell` is a boolean combination of `cell(m, n)` values for $-1 \leq m, n \leq 1$, and we prove it is equal to the GOL step function from section 3.1 by enumerating the 512 possibilities.
- The final floodfill theorem has only two outputs, which overlap the two inputs, and therefore they cancel and produce a complete GOL (not GOL-IO) simulation. In particular, since one of these inputs has value `this`^[-15], we know that in the final simulation, if we sample a particular pixel in this IO port at multiples of 586 ticks, it will be on iff the corresponding mega GOL simulation pixel is on.

In the end, the final theorem we obtain looks like this (`gol_in_gol_circuit_thm`):

$$\vdash \forall n S. \text{step}^n S = \text{read_mega_cells} (\text{step}^{n \times 60 \times 586} (\text{build_mega_cells } S))$$

where `step` is the GOL step function; `build_mega_cells s` takes an input GOL state S and tiles the plane with two versions of the mega-cell in Figure 4, which differ only slightly, in the internal state of the latch; and `read_mega_cells $S = \{p \mid 3150p + (1726, 599) \in S\}$` performs the aforementioned sampling.

6 Conclusion, Related Work and Future Work

In this paper, we have demonstrated that it is possible to formally verify circuits built in GOL and we have verified a circuit that implements GOL itself inside GOL. To the best of our knowledge, this is the first work to formally verify, in an interactive theorem prover (ITP), constructions in a cellular automata. The formalization is roughly 9 500 LOC.

There has been significant prior work on formalizing more traditional models of computation in ITPs, e.g., Turing machines [11, 2, 20, 6], register machines [12, 3], λ -calculus [16, 13, 10, 9], μ -recursive functions [5] and more [17]. We refer to Forster [8] for more in depth discussions on computability in ITPs. Rule 110 [7] is another simple universal CA.

In future work, it would be interesting to explore ITP proofs connecting GOL with more traditional forms of computability. Also, the tools used here could be generalized to prove other GOL circuits, other cellular automata, as well as low level hardware correctness proofs.

References

- 1 Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Hrutvik Kanabar, and Johannes Åman Pohjola. Fast, Verified Computation for HOL ITPs. *J. Autom. Reason.*, 69(1), February 2025. doi:10.1007/s10817-025-09719-8.
- 2 Andrea Asperti and Wilmer Ricciotti. Formalizing Turing Machines. In C.-H. Luke Ong and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation - 19th International Workshop (WoLLIC)*, volume 7456 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2012. doi:10.1007/978-3-642-32621-9_1.
- 3 Jonas Bayer, Marco David, Abhik Pal, Benedikt Stock, and Dierk Schleicher. The DPRM Theorem in Isabelle (Short Paper). In John Harrison, John O’Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP)*, volume 141 of *LIPICs*, pages 33:1–33:7. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ITP.2019.33.
- 4 Nicholas Carlini. Improved Logic Gates on Conway’s Game of Life – Part 3, 2021. URL: <https://nicholas.carlini.com/writing/2021/improved-logic-gates-game-of-life.html>.
- 5 Mario Carneiro. Formalizing computability theory via partial recursive functions. *CoRR*, abs/1810.08380, 2018. arXiv:1810.08380.
- 6 Alberto Ciaffaglione. Towards Turing computability via coinduction. *Sci. Comput. Program.*, 126:31–51, 2016. doi:10.1016/J.SCIC0.2016.02.004.
- 7 Matthew Cook. Universality in Elementary Cellular Automata. *Complex Systems*, 15, March 2004. doi:10.25088/ComplexSystems.15.1.1.
- 8 Yannick Forster. *Computability in constructive type theory*. PhD thesis, Saarland University, Germany, 2021. URL: <https://d-nb.info/1255182792>.
- 9 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ -calculus is reasonable for both time and space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2020. doi:10.1145/3371095.

- 10 Yannick Forster, Fabian Kunze, Gert Smolka, and Maxi Wuttke. A Mechanised Proof of the Time Invariance Thesis for the Weak Call-By-Value λ -Calculus. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP)*, volume 193 of *LIPICs*, pages 19:1–19:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ITP.2021.19.
- 11 Yannick Forster, Fabian Kunze, and Maxi Wuttke. Verified programming of Turing machines in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 114–128. ACM, 2020. doi:10.1145/3372885.3373816.
- 12 Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and minsky machines. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 104–117. ACM, 2019. doi:10.1145/3293880.3294096.
- 13 Yannick Forster and Gert Smolka. Call-by-Value Lambda Calculus as a Model of Computation in Coq. *J. Autom. Reason.*, 63(2):393–413, 2019. doi:10.1007/S10817-018-9484-2.
- 14 Martin Gardner. Mathematical Games: The Game of Life. *Scientific American*, 223(4):120–123, October 1970. URL: <https://static.scientificamerican.com/sciam/cache/file/868064C1-379D-4992-9A60F4615C3D713F.pdf>.
- 15 Nathaniel Johnston and Dave Greene. *Conway’s Game of Life, Mathematics and Construction*. Self-published, 2023. doi:10.5281/zenodo.6097284.
- 16 Michael Norrish. Mechanised Computability Theory. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - Second International Conference (ITP)*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. doi:10.1007/978-3-642-22863-6_22.
- 17 Thiago Mendonça Ferreira Ramos, César A. Muñoz, Mauricio Ayala-Rincón, Mariano M. Moscato, Aaron Dutle, and Anthony Narkawicz. Formalization of the Undecidability of the Halting Problem for a Functional Language. In Lawrence S. Moss, Ruy J. G. B. de Queiroz, and Maricarmen Martínez, editors, *Logic, Language, Information, and Computation - 25th International Workshop (WoLLIC)*, volume 10944 of *Lecture Notes in Computer Science*, pages 196–209. Springer, 2018. doi:10.1007/978-3-662-57669-4_11.
- 18 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.
- 19 Gian Marco Todesco. *Cellular Automata: the Game of Life*, pages 231–243. Springer Milan, Milano, 2013. doi:10.1007/978-88-470-2889-0_25.
- 20 Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference (ITP)*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013. doi:10.1007/978-3-642-39634-2_13.