Universiteit Leiden

A Coalgebraic Cellular Automaton Model for the Verification of Redstone Systems with Minimal Logical Fragments

Raoul Rutgers (s3879682)

Supervisors:
Henning Basold & Tobias Kappé

Bachelor Thesis– Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 27, 2026

Abstract

Minecraft Redstone systems express complex behaviour through local interactions between components that propagate and transform signals over time. These dynamics resemble those of Cellular Automata (CA), making CA a natural framework for modelling and analysing Redstone circuits. While simulation can demonstrate how a circuit evolves, determining whether behavioural properties hold often requires extensive manual inspection and thus becomes impractical for larger systems.

This thesis investigates the formal verification of Redstone circuits by modelling them as Cellular Automata and applying fragments of modal fixed-point logic. Building upon an existing Haskell framework for logical verification of Cellular Automata, the framework is extended with additional logical operators, including implication and least and greatest fixed-point operators, together with a Cellular Automaton model of Redstone components and circuits. The resulting system enables the specification and verification of behavioural properties such as signal propagation, reachability, stabilisation, persistence, and oscillation.

The primary objective is to determine what properties of Redstone systems can be expressed with different logical features and to identify minimal logical fragments sufficient for their verification. To evaluate the relationship between expressive power and verification effort, several logical fragments are compared experimentally on Redstone circuits varying in complexity. Verification performance is measured through execution time and the number of logical operations performed during model checking. The results provide insight into the trade-off between logical expressiveness and verification efficiency in Cellular Automata models of Redstone systems.

Contents

1 Introduction

Minecraft [17] is an open-world sandbox game set in a world of blocks where players can express their creativity through exploration of the world through deep caves, ancient monuments and other structures, creating their own stories or building whatever they can imagine with the blocks they obtain or craft throughout the game. Of all the block types in the game, there is one that is particularly interesting in its behaviour, namely Redstone Dust. Redstone Dust is the basic resource that allows for crafting a whole branch of blocks known as the Redstone system [15]. The Redstone System is essentially Minecraft's own form of circuitry or a power network to enable the creation of complex circuits or machines. Redstone Dust is used in most crafting recipes for creating Redstone component blocks and the dust itself also acts as a form of wiring, propagating power to connect each of these components. Redstone components can transmit signals, store state, and even implement logical operations, allowing players to construct increasingly complex circuits ranging from simple automated doors to fully programmable computers. In this research, we are most interested in the update behaviour of Redstone. Redstone systems evolve through local interactions between neighbouring components over discrete timesteps, meaning each component updates its state based on the states of the components around it and all components update mostly synchronously (with some exceptions). This behaviour strongly resembles that of Cellular Automata, allowing the creation of a model of the Redstone systems as a Cellular Automaton enabling us to simulate its behaviour programmatically.

A Cellular Automaton (CA) [20] is a mathematical model, often a grid, of cells that evolve synchronously over discrete time steps according to fixed local update rules, possibly per cell. We notice a clear similarity to the behaviour of Redstone. Despite this simplistic definition, CA are capable of producing highly complex global behaviour, including oscillation, propagation, self-organisation, and some CA can even express universal computation or Turing completeness [4]; the ability to perform any computation. The similarity between the behaviours of CA and Redstone make CA a natural framework for modelling Redstone circuits and simulating their evolution through time.

However, simulation alone is often insufficient for understanding the behaviour of such systems. While a simulation can display how a circuit evolves over time, reasoning about whether a desired property holds may require manually inspecting many configurations across many time steps. This rapidly becomes impractical as circuits increase in size and complexity. This research builds on the work of Basold and colleagues [1], who address this by introducing formal logical methods to specify and verify behavioural properties declaratively. Using logic to verify these properties about our CA models of Redstone circuits creates a quick method of reasoning about the global behaviour of these circuits without the need for manual inspection of extensive simulations.

In this thesis, we study different logical subsets, or fragments, for verification of Cellular Automata models of Redstone systems. These fragments are subsets of logical languages with differing expressiveness, such as basic modal logic versus the introduction of fixed point operations. We study these fragments capable of expressing temporal and spatial properties of Redstone circuits, such as signal propagation, stabilization, persistence, and oscillation. While highly expressive logical systems can describe complex recursive behaviour, increased expressiveness is often accompanied by greater computational cost during verification. This motivates the central question of how much logical expressiveness is actually necessary to verify practically relevant behavioural properties. Formally:

- *What are the minimal logical fragments that are sufficient to verify relevant behavioural properties of a CA modelling Redstone systems, and how does restricting logical expressiveness affect verification performance?*

The primary goal of this research is therefore to identify *minimal* logical fragments sufficient for verifying relevant behavioural properties of Cellular Automata modelling Redstone systems. We investigate which logical operators are essential for expressing particular classes of properties, and how restricting the available logical methods affects model-checking performance in practice.

More specifically, this thesis addresses the following research questions:

- *Which behavioural properties of Redstone circuits can be expressed in the original modal logic, and which require explicit fixed-point operators?*
- *How does restricting logical expressiveness affect the computational performance of model checking?*
- *Which logical fragments provide the best balance between expressive power and verification efficiency?*
- *What logical features are necessary to reason about signal propagation, stabilization, persistence, and oscillation in finite Redstone systems?*

To answer these questions, we first formalise Redstone circuits as Cellular Automata and define a modal fixed-point logic interpreted over their configurations. We then implement these models and verification procedures in Haskell, experimentally compare different logical fragments, and evaluate the relationship between expressive power and practical verification performance.

1.1 Thesis overview

This research is conducted for my bachelor thesis for Computer Science at LIACS at the University of Leiden. This research and writing of this thesis were made possible with the support and advice of my supervisors at LIACS, Henning Basold and Tobias Kappé.

This section contains the introduction; Section 2 discusses related work covering CAs, Logic and similar works pertaining to Redstone modelled as CAs or common CA problems being modelled within Redstone; Section 3 provides the reader with the necessary preliminary knowledge required to understand the background, clarifying logical concepts and notations; Section 4 provides necessary background on Minecraft and Redstone, CAs, and Logic verification; Section 5 covers the implementation of the logical methods and Redstone systems within Haskell; Section 6 describes the experimental setup and the results of the experiments, along with discussion about them to answer the research questions; Section 7 concludes the thesis, summarizing the results and discussing future work.

2 Related Work

Cellular Automata (CAs) have long been studied as mathematical models of computation and complex dynamical systems, systems with interdependent parts that evolve non-linearly over time. The field began with the work of von Neumann, who investigated self-reproducing automata

as a means of understanding how machines could replicate themselves and exhibit increasingly complex behaviour through purely local interactions [20]. His work established the foundational idea that simple components following fixed local rules could collectively produce sophisticated global behaviour, a principle that remains central to CA research.

Further research expanded the study of CAs beyond self-reproduction toward computation and emergent behaviour. In particular, Wolfram and Cook systematically investigated large classes of automata and demonstrated that surprisingly complex and often unpredictable patterns can emerge from extremely simple update rules [21, 4]. Their classification of CAs highlighted how local interactions can develop properties such as propagation, periodicity, self-organisation, and apparent randomness, motivating their use as models for a wide range of natural and computational systems.

A major milestone in the field was the demonstration that certain CAs are capable of universal computation. Cook proved that the elementary Cellular Automaton Rule 110 is Turing complete, showing that even a one-dimensional automaton with a binary state space and a simple local update rule can simulate arbitrary computation [4]. This result established that computational universality does not require complex architectures and reinforced the view of CAs as a powerful computational framework rather than simply a mathematical curiosity.

Alongside the development of CA theory, formal verification techniques have become an important tool for reasoning about the behaviour of computational systems. Rather than relying solely on simulation, model checking and logical verification allow properties of a system to be expressed declaratively and verified automatically [3]. Modal and fixed-point logics, particularly the modal μ-calculus, have proven especially successful due to their ability to express temporal properties such as reachability, persistence, recurrence, and stabilisation [11, 2]. These logics provide a thorough framework for reasoning about systems whose behaviour unfolds over time.

Although both CAs and logical verification have been studied extensively, comparatively little work has explored their combination in the context of Minecraft Redstone systems. Existing Minecraft-related CA works have largely focused on simulation, computation, and implementation without any addition of formal verification for improved model checking, to our knowldege.

Several projects have demonstrated the close relationship between Minecraft Redstone and CAs. Land's implementation of Rule 110 in Redstone showed that a known universal CA can be realised directly within Minecraft, providing a practical demonstration of computational universality in the game environment [12]. Other projects such as Minecraft Cellular Automata and Minestom Cellular Automata provide software frameworks for constructing and simulating CAs within Minecraft worlds, enabling experimentation with automaton behaviour at scale [9, 7].

Other research has also explored alternative computational abstractions within Minecraft. MinecraftHDL enables digital circuits to be described using hardware description language concepts before being realised in Redstone, demonstrating great advancements in the computational modelling and design within Minecraft [8].

While these works establish strong connections between CAs, computation, and Minecraft systems, they primarily focus on construction, simulation, or generation. In contrast, this present work investigates the formal verification of Redstone circuits by modelling them as CAs and applying fragments of modal fixed-point logic to reason about their behaviour. To my knowledge, no previous work has examined the expressive requirements and verification performance of modal fixed-point logics in this context.

3 Preliminaries

This section introduces the mathematical terminology and notation used throughout this research. In particular, we introduce monotone functions, fixed points, and environments used to interpret propositional variables.

3.1 Sets and Functions

Let X denote the set of cells in a cellular automaton. We write $\mathcal{P}(X)$ for the power set of X, that is, the set of all subsets of X. For sets $A, B \subseteq X$, we write $A \subseteq B$ if every element of A is also an element of B.

A function

$$F : \mathcal{P}(X) \to \mathcal{P}(X)$$

is *monotone* if, for all $A, B \subseteq X$,

$$A \subseteq B \implies F(A) \subseteq F(B).$$

3.2 Modal Logic

We define formulas to specify verification properties.

Definition 1. Formally, formulas φ are defined by:

$$\varphi ::= \top \mid \bot \mid s \mid V \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \implies \varphi \mid \Diamond_m \varphi \mid \boxtimes \varphi \mid \mu V.\varphi \mid \nu V.\varphi$$

where:

- $\top$ and $\bot$ represent true and false respectively,
- V ranges over propositional variables used to reference a fixed point variable within a recursive formula,
- $s \in S$ represents a state predicate,
- $\vee$, $\wedge$ and $\implies$ are the standard Boolean operators,
- $\Diamond_m$ is a spatial modal operator corresponding to a move $m \in M$ from Definition 2,
- $\boxtimes$ is a temporal "next-step" operator interpreted using the global transition function G,
- μ and ν denote least and greatest fixed-point operators respectively.

3.3 Fixed Points

Let

$$F : \mathcal{P}(X) \to \mathcal{P}(X).$$

A set $T \subseteq X$ is a *fixed point* of F if

$$F(T) = T.$$

The *least fixed point* of F, denoted by μF, is the smallest fixed point with respect to set inclusion. Similarly, the *greatest fixed point*, denoted by νF, is the largest fixed point with respect to set inclusion.

For a monotone function $F : \mathcal{P}(X) \to \mathcal{P}(X)$, both μF and νF exist by the Knaster–Tarski fixed-point theorem [10, 19]. They can be characterised as

$$\mu F = \bigcap \{T \subseteq X \mid F(T) \subseteq T\},$$

and

$$\nu F = \bigcup \{T \subseteq X \mid T \subseteq F(T)\}.$$

Since both are fixed points, they satisfy the unfolding properties

$$\mu F = F(\mu F) \qquad \text{and} \qquad \nu F = F(\nu F).$$

These properties are used in the interpretation of recursive fixed-point operators in the modal μ-calculus [2, 11].

3.4 Environments

Let Var be a set of propositional variables. An *environment* is a function

$$env : Var \to \mathcal{P}(X),$$

which assigns to each propositional variable a set of cells.

For a variable $Z \in Var$ and a set $T \subseteq X$, we write

$$env[Z := T]$$

for the environment obtained by assigning T to Z while leaving the interpretation of every other variable unchanged. Thus,

$$env[Z := T](Z) = T$$

and

$$env[Z := T](Y) = env(Y) \qquad \text{for all } Y \in Var,\ Y \neq Z.$$

Environments are used when evaluating formulas containing propositional variables and fixed-point operators [2].

4 Background

This section covers the background that this research builds on, from Minecraft and Redstone, to Cellular Automata and Logic verification methods.

4.1 Minecraft Redstone

Minecraft [17] is an open-world sandbox game in a procedurally generated world consisting entirely of blocks. Players can break, or *mine*, these blocks to obtain them as materials for *crafting* new blocks or simply for placing them down elsewhere. In this way players can express their creativity in building anything they can imagine. One such material the players can craft and build with is Redstone Dust, mined from Redstone Ore underground. While not necessarily a typical structural building block, it has a functional purpose. Redstone Dust unlocks the creation of other Redstone components, forming what is known as the Redstone system [15]. The Redstone system is, in essence, Minecraft's own form of circuitry and (electrical) power. Redstone Dust functions as wiring to propagate power between Redstone components, crafted with this same dust, to connect them, creating complex circuits from automatic doors to memory and logic components, which when combined can be used to create full computers. Players, such as mattbatwings [14, 13] have even created working computers within Minecraft, and similarly players such as sammyuri and colleagues [18] have created a playable version of Minecraft within Minecraft using the Redstone system.

In this research we will only be modelling a subset of Redstone with its basic components. The model focuses on static Redstone circuits, capable of simulating logical circuits, rather than circuits that rely on the physical movement of components, such as automatic doors or flying machines. These mechanical applications are outside the scope of this research, as modelling their physical movement is not necessary for investigating the logical computation capabilities of Redstone, which will be the focus of the larger circuits evaluated in this research. In this research we are only modelling 7 basic Redstone components and Minecraft blocks, which are sufficiently expressive to support universal computation through a combination of logical gates made from these components. In future work it is possible to incorporate moving components, as well as more static ones, however these will be outside the scope of this research.

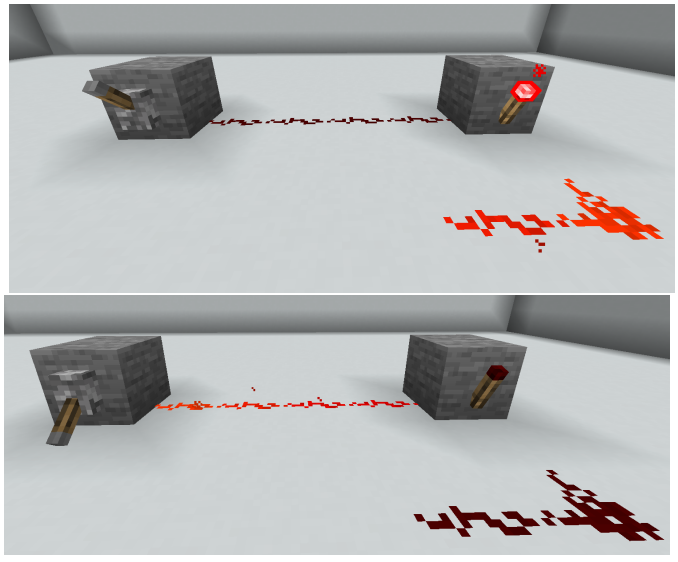

Figure 1: NOT gate created with Redstone, OFF (top) and ON (bottom).

Consider this example circuit in Figure 1 of a NOT gate created with Redstone. This circuit only makes use of a Lever, two Stone blocks (solids), a Redstone Torch and some Redstone Dust

to connect the components. We can see that when the Lever is switched off (in the top image), the Redstone Dust in between the blocks is not glowing, indicating it is off, and that the torch on the right side is lit, indicating it is on and emitting power to the Dust coming out of it. In the bottom image we see the Lever is switched on as the Dust in between the blocks is now glowing and that this causes the Redstone Torch to turn off, which stops power from propagating through the Dust on the end as well. We see that an unpowered input leads to a powered output, and that a powered input leads to an unpowered output. This is indeed the behaviour of a NOT logic gate.

We will further explain the precise behaviour of each of these components in later sections, however this circuit will be used as a reference throughout to explain the process of conversion from Redstone circuit to property verification.

Of particular interest for this research is the behaviour of Redstone and the way circuits evolve over time. Redstone's behaviour is controlled by Minecraft's discrete tick-based update system [16]. Time within the game progresses through successive ticks, with most block updates occurring synchronously at fixed intervals. A standard game tick occurs approximately every 50 milliseconds, while many Redstone-specific updates operate on a two-tick timescale commonly referred to as a Redstone tick.

At each update step, a Redstone component determines its next state according to a fixed ruleset per block type depending on its current state and the states of neighbouring components. Thus, the behaviour of an entire circuit is determined from repeated local interactions between nearby blocks over discrete timesteps. There are some exceptions as certain Redstone components exhibit asynchronous or instantaneous behaviour, most notably Redstone Dust which propagates power across connected paths within a single game tick, however the overall system remains predominantly local and synchronous in nature. In this research we adhere to this behaviour of Redstone Dust propagating instantly within one timestep, and all other components evolving normally; synchronously through the timesteps.

This update behaviour strongly resembles that of Cellular Automata, in which simple local update rules also collectively generate complex global dynamics. These similarities naturally lead to Cellular Automata possibly providing a mathematical framework for modelling and analysing Redstone circuits programmatically, without needing to manually analyse the circuits through simulation in the game.

4.2 Cellular Automata

Cellular Automata (CA) [20], often depicted as grid-like structures, are mathematical systems consisting of cells, whose states evolve over discrete timesteps according to local interaction rules. This means all cells update synchronously in each timestep according to either global update rules or local rules per cell. In either case, the cell updates its state based solely on its own state and the state of its neighbouring cells in the current configuration. And as each cell updates synchronously, the CA is globally updated in each timestep. These simple rules, depending on the ruleset, can collectively make CA capable of exhibiting highly complex behaviour, as shown by Wolfram and Cook [21, 4].

One of the most well-known examples of CA is Conway's *Game of Life* [6], in which each cell on a two-dimensional grid, with an orthogonal neighbourhood consisting of all directly adjacent cells vertically and horizontally, is either alive or dead. Each cell follows only 3 simple rules; cells with 3 live neighbours come to life, cells with exactly 2 or 3 live neighbours stay alive, and all other

cells die; see Figure 2. Despite their simplicity, these rules have shown surprisingly interesting behaviour, including moving patterns, oscillators, and structures capable of performing computation. Elementary one-dimensional Cellular Automata such as Wolfram's Rule 30 similarly demonstrate how highly unpredictable global behaviour can emerge from extremely small local rulesets.

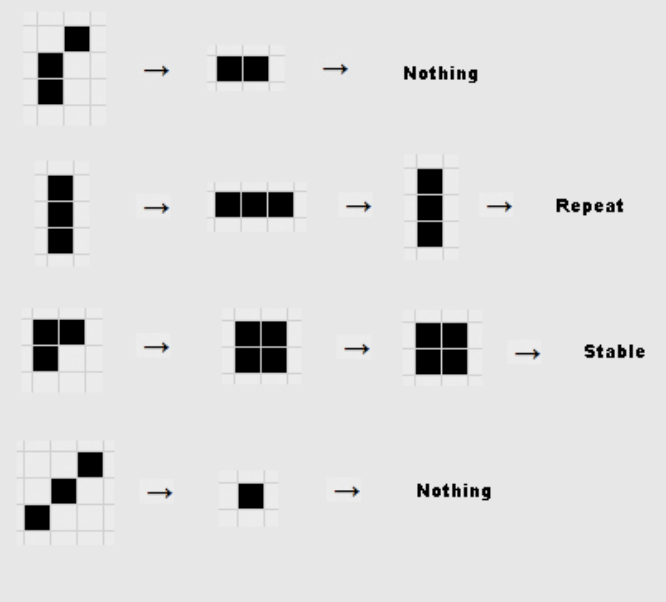

Figure 2: Simple Game of Life example [5]

The relevance of Cellular Automata to this research lies, again, in the strong similarity between the update behaviours of CA and Redstone in Minecraft. Recall that Redstone systems evolve through local interactions between neighbouring blocks, and these interactions occur in a mostly synchronous tick-based manner. The state of a Redstone component at a given moment depends primarily on the states of nearby components during the previous tick. This local and discrete evolution makes Cellular Automata a natural framework for modelling Redstone circuits.

In the work of Basold and colleagues [1], a Cellular Automaton is defined formally as follows:

Definition 2. A cellular automaton is a tuple

$$A = (X, S, M, N, \gamma_1, \gamma_2),$$

where

- $\mathbf{X}$ is a set of cells, with $X \subseteq \mathbb{Z}^3$ for our Redstone circuit models;
- $\mathbf{S}$ is a set of states;
- $(\mathbf{M}, \cdot, \mathbf{1})$ is a monoid representing moves between cells. A monoid is a set defined with an associative binary operation $\cdot$ and an identity element 1. The operation represents the composition of multiple moves, such that applying a set of moves in sequence is equivalent to applying their composition. For example, a move $(0, 1, 1)$ can be applied to cell $[3, 4, 5]$ to obtain cell $[3, 5, 6]$;

- $\mathbf{N \subseteq M}$ is a neighbourhood, a subset of moves $m \in M$ that determines which cells qualify as being *nearby*. The neighborhood is defined relative to the spatial structure γ_1;
- $\boldsymbol{\gamma_1 : X \to [M, X]}$ assigns to each cell its spatial structure, where $[M, X]$ denotes the set of functions from M to X. Thus, $\gamma_1(x) : M \to X$ maps each possible move $m \in M$ to the cell reached from x by applying that move;
- $\boldsymbol{\gamma_2 : X \to [L_x, S]}$ assign a local update rule to each cell. The domain L_x represents the possible local state configurations of x and the cells in its neighbourhood, as determined by the spatial structure $\gamma_1(x)$. The rule $\gamma_2(x)$ maps such a local configuration to a new state $s \in S$.

The monoid structure allows moves to be composed associatively. This is especially relevant to the spatial structure of the cellular automaton, as the move operation can be used to define how the boundaries of the grid connect. In the CA models in the initial work of Basold and colleagues [1], the spatial structure wraps the right boundary around to the left boundary and the top boundary around to the bottom boundary. A move that extends outside one boundary can reach a cell on the opposite boundary.

For the Redstone models considered in this research, this wrapping behaviour is not required as it does not exist within Minecraft. we ignore this wrapping behaviour by adding a padding row to the modelled grid, preventing the circuit from interacting with the boundaries. No circuit in this research is ever as large as the size of the grid.

In the formal definition, the neighbourhood N is determined through the spatial structure γ_1, with each move in N identifying a neighbouring cell. In our implementation, this is simplified by directly defining the neighbourhood of each cell as a list of its adjacent cells. For the Redstone models considered in this research, a cell's neighbourhood consists of the cells directly adjacent to it in the six cardinal directions in 3D space. Thus, for a cell x, its neighbourhood can be written as $\mathcal{N}(x) = \{x + (1,0,0), x + (-1,0,0), x + (0,1,0), x + (0,-1,0), x + (0,0,1), x + (0,0,-1)\} \cap X$. This direct definition does not make use of the monoid M or the spatial mapping γ_1 during simulation, but produces the same fixed neighbourhood relation required by the model.

A configuration is a function

$$c : X \to S$$

assigning a state to every cell.

The global evolution of the system is determined by a transition function

$$G : S^X \to S^X$$

defined by

$$G(c)(x) = \gamma_2(x)\big(c \circ \gamma_1(x) \restriction_N\big).$$

The new state of cell x is obtained by applying the local update rule $\gamma_2(x)$ to the current states of the cells in its neighbourhood. The current configuration mapping the spatial structure of x restricted to its neighbours in N to their states is defined by $c \circ \gamma_1(x) \restriction_N$; this is an element of L_x.

Intuitively, the function G computes the next global configuration by applying the local update rule of every cell simultaneously. Each cell observes only the states within its neighbourhood and updates according to its associated rule. Repeated application of G therefore generates the temporal evolution of the entire automaton.

This formalisation is sufficiently general to model many different classes of Cellular Automata, including non-uniform systems in which cells may have different local rules or neighbourhood structures. In the context of this thesis, it provides a convenient mathematical framework for modelling and simulating finite Redstone circuits and reasoning about their temporal behaviour.

Using this definition, we will model the NOT gate example from Figure 1 as a CA. The Redstone definition with its state space will be shown more in-depth in the implementation section, however we will show a simplified example here. We define the tuple $A = (X, S, M, N, \gamma_1, \gamma_2)$. First we take $X = \{(x, y, z) \in \mathbb{Z}^3 \mid 0 \leq x < 6, 0 \leq y < 1, 0 \leq z < 3\}$; all relevant cells in 3D space, in this case an area $6 \times 1 \times 3$ as those are the dimensions of this circuit. As all components are on the same y level, we can model this circuit as a 2D CA for simplicity, taking $X = \{(x, z) \in \mathbb{Z}^2 \mid 0 \leq x < 6, 0 \leq z < 3\}$. The state of a cell in our model is represented by the tuple $s = (\text{blockId}, \text{power}, \text{direction}, \text{age}, \text{target})$. We simplify this for our example, taking only $S = S_{blockId} \times S_{power} = \{\text{ON}, \text{OFF}\}$; reducing the state space for each component to only its blockId and a power level of ON or OFF. M represents the monoid of moves between cells, as described previously. In 2D space, these moves correspond to movement in the four cardinal directions. The monoid is part of the formal CA definition, but is not explicitly used to determine the neighbourhood in our implementation. As shown previously, N represents the neighbourhood of each cell, which in a 2D space is all adjacent cells in the four cardinal directions; $N = \{x + (1, 0), x + (-1, 0), x + (0, 1), x + (0, -1)\} \cap X$. In this particular circuit, each cell has either 1 or 2 relevant neighbour cells, thus it could be represented in a one-dimensional CA if we set the Lever and Redstone Torch on opposing sides of their respective blocks. The spatial structure γ_1 describes the relationship between cells and their possible moves in the formal definition. In our implementation, this relationship is represented directly by the neighbourhood lists described above, rather than by explicitly applying moves from M through γ_1. γ_2 assigns local update rules to cells; we will be doing this per block-type, denoted by the blockId state, as follows: Lever is on or off and does not update, Stone block is on if it receives power from a neighbour or propagates power to all neighbours, Dust propagates power to all neighbours if it receives power from any neighbour, just as Stone, and finally the Redstone Torch remains powered unless the block it is attached to becomes powered. We will make use of the simplified circuit and simulate its evolution in both the ON and OFF case.

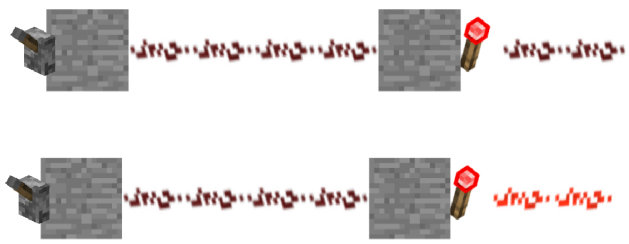

Figure 3: Simplified one-dimensional Redstone NOT gate, OFF case evolution

In the OFF case in Figure 3, we see that only the Torch near the end is powered. This power will propagate through the Dust at the end and reach a stable state, resulting in a powered (ON) output. In Minecraft, the dust would propagate along its network instantly. Our implementation does not follow this rule and is instead entirely sequential, however, the steps of power passing along the wire network will not explicitly be shown for this example for brevity.

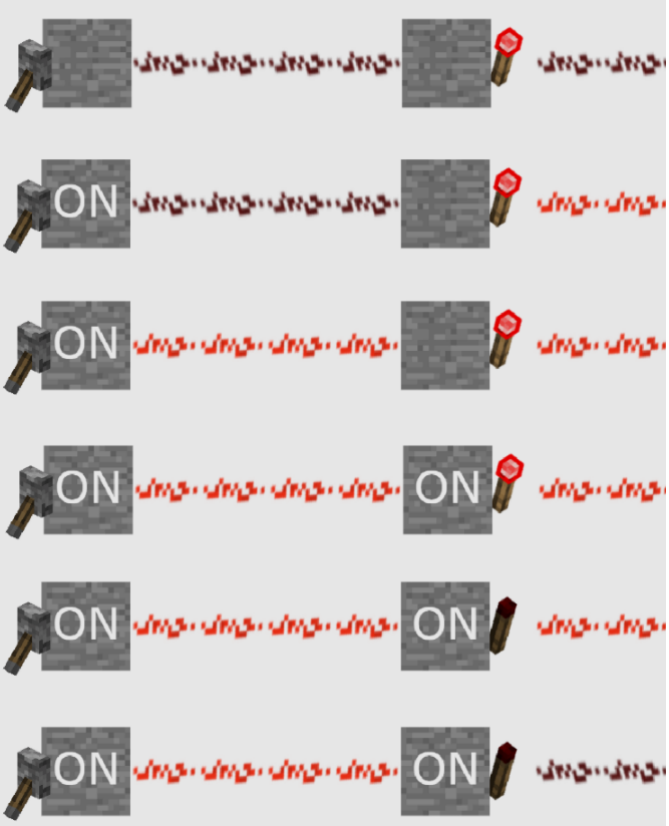

Figure 4: Simplified one-dimensional Redstone NOT gate, ON case evolution

In the ON case in Figure 4, we see that both the Torch near the end and the Lever at the front are powered. The Torch will propagate power to the Dust at the end, just as before, and the Lever will propagate its power through the Stone, to the Dust in the middle, through the final Stone, and finally to the torch, shutting it off, which in turn unpowers the Dust at the end to reach a stable state, resulting in an unpowered (OFF) output.

However, simple simulation and analysis of a CA model of Redstone is not enough to practically reason about the behaviour of a circuit. If we wanted to understand its behaviour or determine whether some property holds, we would need to run through the simulation and manually inspect the global state in each timestep to note any needed information. This is inefficient and quickly becomes impractical in larger or more complex circuits. This motivates the use of formal logical methods for reasoning about the behaviour of Redstone circuits, eliminating the need for manual analysis.

4.3 Logic

Logical methods allow us to verify properties and reason about our CA model of Redstone without needing to analyse states manually for each step in a CA simulation.

4.3.1 Logical Verification

For this research we are particularly interested in verifying properties such as reachability, whether a signal eventually reaches some component; persistence, whether a signal remains unchanging forever; stabilisation, whether a circuit stabilises after finitely many steps; and oscillation, whether a configuration oscillates indefinitely.

To verify these properties and reason about our CA models, we define a simple modal logic extended with spatial and temporal operators to express movement between cells and steps in time, as implemented by Basold and colleagues [1], and newly implemented fixed-point operators, allowing verification of recursive temporal and spatial properties, i.e. properties requiring an unknown number of operators to verify. For example, we typically can not know how many temporal operators; applications of the global evolution on the CA, may be required for a circuit to stabilise.

Recall the definition of a formula from Section 3. Intuitively, the formula $\Diamond_m \varphi$ states that φ holds at a neighbouring cell reachable by move m, while $\boxtimes\varphi$ states that φ holds after one application of the global transition function G. The addition of fixed-point operators allows us to express recursive temporal and spatial properties such as eventuality, persistence, oscillation, existence, and patterns.

We interpret formulas as sets of cells satisfying the formula in a given configuration. For a configuration c, we define:

$$[\![\varphi]\!]_c^{env} \subseteq X.$$

recalling that X is the set of all cells in the automaton from Definition 2.

The interpretation is given inductively for each operator, where i is a finite integer, by:

$$\begin{aligned}
[\![\top]\!]_c^{env} &= X, \\
[\![\bot]\!]_c^{env} &= \varnothing, \\
[\![s]\!]_c^{env} &= \{x \in X \mid c(x) = s\}, \\
[\![V]\!]_c^{env} &= env(V), \\
[\![\bigvee_i \varphi_i]\!]_c^{env} &= \bigcup_i [\![\varphi_i]\!]_c^{env}, \\
[\![\bigwedge_i \varphi_i]\!]_c^{env} &= \bigcap_i [\![\varphi_i]\!]_c^{env}, \\
[\![\varphi \implies \psi]\!]_c^{env} &= X \setminus [\![\varphi]\!]_c^{env} \cup [\![\psi]\!]_c^{env}, \\
[\![\boxtimes\varphi]\!]_c^{env} &= [\![\varphi]\!]_{G(c)}^{env}, \\
[\![\Diamond_m \varphi]\!]_c^{env} &= \{x \in X \mid \gamma_1(x)(m) \in [\![\varphi]\!]_c^{env}\}.
\end{aligned}$$

The atomic formula s selects all cells currently in state s. Disjunction takes the union of cells satisfying each formula, while conjunction takes their intersection. Implication selects all cells for which either the antecedent is false or the consequent is true: $\varphi \implies \psi$ is interpreted as $\neg\varphi \vee \psi$.

The propositional variables simply select the cells they represent within the environment.

A cell satisfies $\Diamond_m \varphi$ whenever the neighbouring cell reached by move m satisfies φ. In this way, modal operators allow formulas to reason locally about neighbouring cells and signal propagation through the system.

The temporal operator $\boxtimes$ shifts reasoning forward by one timestep. A cell satisfies $\boxtimes\varphi$ if φ holds after one application of the global transition function G. This directly connects the logic to the temporal evolution of the Cellular Automaton.

While these operators allow us to reason about local structure and single future steps, many important properties of Redstone circuits are inherently recursive. For example, we may wish to express that:

- **a signal eventually reaches a component:** if φ denotes "the component is powered", then this can be expressed as $\varphi \vee \boxtimes\varphi \vee \boxtimes^2\varphi \vee \cdots$,
- **a circuit eventually stabilises:** if φ denotes "the configuration no longer changes", then this can be expressed as $\varphi \vee \boxtimes\varphi \vee \boxtimes^2\varphi \vee \cdots$,
- **a state remains true forever:** if φ denotes "the component is powered", then this can be expressed as $\varphi \wedge \boxtimes\varphi \wedge \boxtimes^2\varphi \wedge \cdots$,
- **a configuration repeatedly oscillates:** if φ denotes "the configuration matches its initial state", then this can be expressed as $\varphi \vee \boxtimes^k\varphi \vee \boxtimes^{2k}\varphi \vee \cdots$ for some period $k > 1$.

Such properties cannot always be expressed using only finitely many applications of $\boxtimes$ as in the example. It is generally not knowable how many operators are required to verify some property in an arbitrary circuit. It is possible to perform any finite number of operators manually, continuously adding 1 each time you run the program until you possibly find a stable state or a state in which your property holds. It is also possible to manually inspect the circuit and compute the required number of step before you reach a terminating state, however both of these options quickly become unreasonable for larger or more complex circuits. To reasonably describe behaviour over arbitrarily many timesteps practically, we introduce fixed-point operators.

For fixed-point formulas, the bound variable must occur only positively. This means that increasing the set assigned to the variable cannot decrease the resulting set of cells. In our logic, a variable occurs positively when it is used directly, within conjunctions or disjunctions, or in the consequent of an implication. An occurrence in the antecedent of an implication is negative. This restriction ensures that the operator called by the formula is monotone as defined in Section 3. Given a formula φ and an environment env, this operator is defined as

$$F_\varphi^{env}(T) = [\![\varphi]\!]_c^{env[Z:=T]}.$$

By the Knaster–Tarski fixed-point theorem [10, 19], a monotone operator on $\mathcal{P}(X)$ has a least and a greatest fixed point. Thus, the semantics of $\mu Z.\varphi$ and $\nu Z.\varphi$ are well-defined.

Using fixed points, we can express important temporal and spatial properties. Since the world is bounded and the state space is finite, the automaton does not have an infinite number of possible configurations and thus it will eventually either stabilise or fall into repetition. Once this occurs, a state will be found with that has already been seen before and thus we know we can stop searching.

Eventually The formula

$$\mu X.(\varphi \vee \boxtimes X)$$

expresses that φ eventually becomes true.

A fixed point satisfies the equation defined by its operator. Thus, for $F(X) = \varphi \vee \boxtimes X$, we have

$$\mu X.(\varphi \vee \boxtimes X) = \varphi \vee \boxtimes(\mu X.(\varphi \vee \boxtimes X)).$$

Substituting the fixed point into itself allows us to unfold the formula repeatedly:

$$\mu X.(\varphi \vee \boxtimes X) = \varphi \vee \boxtimes(\varphi \vee \boxtimes(\varphi \vee \cdots)).$$

This characterises all cells for which φ becomes true after finitely many applications of the transition function G.

The least fixed point begins with an empty set $\varnothing$ and iteratively applies $\boxtimes$, checking for φ at each iteration and adding cells for which φ will hold to the set. Since this is finite, the least fixed point will terminate and we will end up with the smallest set of all cells that will hold for φ at some point after some finite number of applications for $\boxtimes$. In other words, all cells that eventually hold for φ.

Always Similarly,

$$\nu X.(\varphi \wedge \boxtimes X)$$

expresses that φ remains true forever.

Unfolding gives:

$$\varphi \wedge \boxtimes(\varphi \wedge \boxtimes(\varphi \wedge \cdots)).$$

Thus φ must hold now, after one timestep, after two timesteps, and so on indefinitely.

The greatest fixed point begins with all cells X and continuously applies $\boxtimes$ removing all cells that will not hold for φ at that iteration. Since this search is finite, it will eventually terminate and leave us with the largest set of all cells that will never not hold for φ after some finite number of applications of $\boxtimes$. In other words, the largest set of cells that will always hold for φ. And since the search only terminates after stabilisation or repetition, this will hold for infinite applications of $\boxtimes$ as well.

These constructions allow the logic to express many of the temporal behaviours relevant to Redstone circuits, including signal propagation, stabilisation, persistence, and oscillation.

Finally, a formula φ is said to hold at a cell x in configuration c whenever $x \in [\![\varphi]\!]_c^{env}$. Thus, $[\![\varphi]\!]_c^{env} \subseteq X$ represents the set of all cells satisfying φ in the current configuration.

4.3.2 Logical Fragments

An important concept in this research is that of a logical fragment. A logical fragment is a restricted subset of the full logic obtained by limiting which operators may appear in formulas. For example, one fragment may exclude fixed-point operators entirely, while another may restrict temporal operations or other recursive constructions.

More expressive fragments are capable of specifying more sophisticated behavioural properties, but often at the cost of more expensive verification procedures. Simpler fragments may be computationally efficient, but unable to express certain recursive temporal properties.

Of particular interest are minimal logical fragments; the smallest fragments capable of expressing and verifying a given class of behavioural properties. The central aim of this research is to determine which logical features are genuinely necessary for reasoning about important Redstone behaviours, and how or whether restricting logical expressiveness affects verification performance in practice.

A fragment is considered smaller when it allows fewer logical operators or constructs. A fragment is minimal for a given class of properties if removing any more operators would make it impossible to express that class of properties. Thus, finding a minimal fragment equates to identifying the smallest set of logical features that is still sufficient for the properties to be verified.

5 Approach

Having established that the fixed-point semantics are well-defined and that the verification process correctly determines whether a formula holds, we now show how these methods are implemented in Haskell.

5.1 Logic Implementation

As mentioned previously, this research builds on the work of Basold and colleagues [1]. Their work allowed for the verification of some random global configuration, generated using the `uniformRM` function, or a given configuration for some simple CA models. Specifically, it allows model checking on `torusQ2R` and `ringMod` through logical verification using basic modal logic. These models will not be explained in depth as they will not be used for this research. We are only making use of the CA and Logic verification framework in this work. We also do not make use of the random function in this research.

The initial verification logic framework, relating to the definitions in Section 4.3, consisted of the operators for:

- *Top*, $\top$;
- *Bot*, $\bot$;
- *StateF*, $s \in S$ of the current cell;
- *Space*, $\Diamond$;
- *Time*, $\boxtimes$;
- *Conj*, $\wedge$; and
- *Disj*, $\vee$.

The program is run from the command line, making use of keywords to perfrom different tasks. This is generally done using the following argument structure: `stack run -- [general arguments] COMMAND [command arguments]` where `COMMAND` denotes the task that will be performed.

In order to perform a logical verification, a user would need to run the program along with a valid model type and optionally an initial configuration, followed by the `check` command along with the formula to verify the model on. This will return the set of all cells that hold for the given formula from the initial configuration. It is also possible to simply simulate the CA evolving through a certain number of global state progressions using the `run` command in order to inspect the states in each step manually.

Recall from Section 4 that these simple modal logic operators do not have a notion of recursion and thus cannot reasonably express verification problems over an arbitrary number of time steps or spatial steps, and that we address this by introducing fixed point logic. We extend the work of Basold and colleagues with the implementations of a few more logic operators. These operators had already been defined syntactically, however they had not yet been implemented. We implemented the following operators, relating to those in Section 4.3 where possible:

- *PropVar*, V, implemented by creating an environment map storing the values of the propositional variables. This map is searched to find the value of a given variable;
- *Implies*, $\rightarrow$, implemented through the equivalence $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$;
- *LfpF*, representing the least fixed point operator μ, used to express recursive properties involving reachability and eventuality; and
- *GfpF*, representing the greatest fixed point operator ν, used to express recursive properties involving persistence, invariance, and recurring behaviour.

All of these operators are executed in the checker. The original work already consisted of a framework for the checker consisting of the aforementioned operators, a few of which, again, had not yet been implemented. This original checker took a formula as input and searched through the entire model returning all cells that held for the given formula. This new work introduces an alternate checker that additionally takes a cell as input and only returns True or False depending on whether the given formula holds for the given cell. This was changed to allow for the fixed points to work properly as they had proved difficult to get right in the previous implementation. In the original checker, fixed-point evaluation could terminate based on the result of an individual cell, even though the fixed point still needed to be determined for all other cells in the model. Through a wrapper function, it is still possible to input multiple cells or simply all cells in the new implementation, which will apply the single-cell checker on all given cells and return a list of all cells for which the checker returns true on a given formula.

Apart from the not-yet-implemented operators from the original work, we also introduced two new operators for checking the state of a cell in the current global state during an experiment.

- *FieldEq*, checks whether a given i'th field in the state space (0 to 4) is equal to some given value v. Where i denotes the position of the field we are concerned about in the state tuple: $s = (\text{blockId}, \text{power}, \text{direction}, \text{age}, \text{target})$
- *FieldGT*, checks whether a given i'th field in the state space is greater than some given value v.

These operators are used to check specific values or properties in verification and are used to create helper predicates for simplification. For example, with *FieldGT*, we can create a helper for `isPowered` by checking whether the power variable is greater than zero.

5.2 Redstone Implementation

We introduce the Redstone implementation with the definition of CA defined in Section 4.2. The Redstone CA model sits in a three-dimensional grid space, i.e. the set of cells $X \in \mathbb{Z}^3$ confined to

finite dimensions, with a five-tuple integer state space and a different neighbourhood and update rule for each block type. In our implementation, we use a grid space of $35 \times 35 \times 35$. The state space consists of the following:

- **blockId**, $x = \{n \in \mathbb{Z} \mid 0 \leq n \leq 6\}$ where x corresponds to the block type in the position of the cell. From 0 = Air, an empty cell, to 6 = Lever, a Lever block type. These values are translated to and from custom data types per block type for use in formulas, however they are stored as integers in the state variable as the framework implementation by Basold and colleagues [1] defined the state variable as a list of integers. We aimed to add Redstone without having to significantly modify the existing CA framework or break the example CAs.

- **power**, $x = \{n \in \mathbb{Z} \mid 0 \leq n \leq 15\}$ where x denotes the current power level of the block in the cell. Some blocks, such as Redstone Torches, can only be either on or off, and thus fluctuate between 0 or 15, whereas Redstone Dust can have any value from 0 to 15 depending on its distance from the source of the power.

- **dir**, $x = \{n \in \mathbb{Z} \mid 0 \leq n \leq 5\}$ where x corresponds to the direction in which the block in the cell is facing. This value is important for some blocks that do not emit power in all directions, such as Redstone Torches or Repeaters, to denote in which direction power should be propagated. This value is also translated to and from direction vectors and stored as an integer for the same reason as the blockId. From 0 corresponding to a positive in the x axis, to 5 corresponding to a negative in the z axis, with y values along the vertical axis.

- **age**, $x = \{n \in \mathbb{Z} \mid 0 \leq n \leq 10\}$ where x denotes how long a cell has had its current power level. The age variable increments by one every tick if a block retains the same power as in the previous state. If the power of a block changes in a time step, the age will reset back to zero. In the current implementation this value is used for counting down repeater delay and ensuring wires do not propagate power back and forth. Without a check for age, wires may "ping pong" an old power level back and forth and slowly reducing it after the source of that power has turned off, as they only consider the values of their neighbouring wires. An added check for age ensures wires do not propagate an old value back to wires with a more recent value. If the work is extended with more block types in the future, this value may also be used for countdowns on Pistons, Observers, Redstone Lamps, etc. as these blocks remain activated for more than one tick.

- **target**, $x = \{n \in \mathbb{Z} \mid 0 \leq n \leq 4\}$ where x denotes the target value for blocks that require a timer. The age variable will increment each tick until the target value is reached, upon which the power will change according to the block's update rules.

The general neighbourhood for each cell consists of all six cells orthogonally adjacent in each of the x, y and z directions, as well as the cell itself. This is visualised in figure 5 as all blocks directly adjacent to a face of the block (cube) in the current cell, giving us 6 neighbours. However, not all neighbours will be relevant in the update rules of every block. Each block has some input directions from which they may take power and some output directions in which they may propagate power to. The input and output sets are not necessarily disjoint.

Table 1 shows a table consisting of each block type and its possible states. The naming convention for blocks in this research slightly differ from the official block names in Minecraft

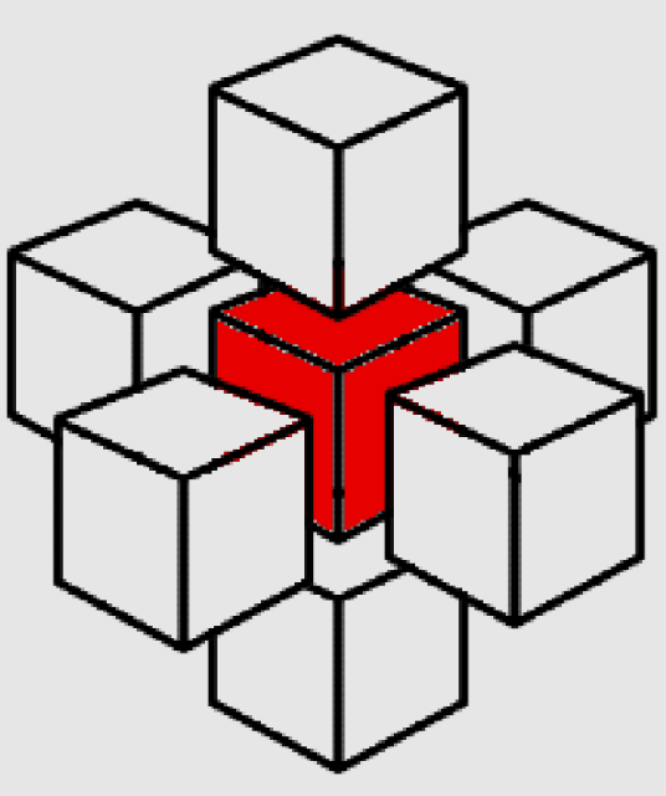

Figure 5: Neighbourhood of a cell

for brevity. Specifically Redstone Torches have been reduced to Torches, as regular Torches are not included in the Redstone system; Redstone Dust has been reduced to Wires, named after its function; the Redstone Block has been reduced to Source, named after its function; and the group of full blocks that interact the in the same way with Redstone components such as Stone, Grass, etc. have been combined under the term Solids.

The directional notations for dir and input/output (I/O) will follow the $\mathbb{Z}^3$ coordinate space, using values such as $+x$ to denote the positive x or right direction, or $-y$ to denote the negative y or down direction. Additionally capital letters, such as X, may be used to signify both the positive and negative direction along the respective axis, for brevity. The age variable has been omitted as this lies within 0 to 10 for all block types.

We continue by describing each block type and their update behaviour. As the current set of blocks does not feature any moving components, the block type within a cell remains static. If more mechanical or moving blocks were added such as Pistons, however, the positions of blocks may be changed. Blocks would move between cells and change the configuration drastically. These block types are not necessarily required to model universal computation. All blocks will update their state once every tick. This does abstract from the behaviour in Minecraft as Redstone Dust (wires) in Minecraft may update between ticks, however this simplification will not have great effect on the behaviour between the block types in this research.

- **Air** blocks are not necessarily blocks, but are used to denote an empty cell. This block does not update.

Block	Power	Dir	Target	I/O
Air	-	-	-	-
Solid	0 or 15	-	-	Input: X, Y, Z Output: X, Y, Z
Source	15	-	-	Input: - Output: X, Y, Z
Wire	[0,..,15]	-	-	Input: X, Y, Z Output: X, Y, Z
Torch	0 or 15	X, Z = on block horizontally $+y$ = on top of block	-	Input: dir Output: $(X, Y, Z) \setminus$ dir
Repeater	0 or 15	X, Z = facing direction	1-4	Input: -dir Output: dir
Lever	0 or 15	X, Z = on block horizontally Y = on block vertically	-	Input: - Output: dir

Table 1: State space for block types

- **Solid** blocks are not a specific block on their own, but are interchangeable for any conductive, non-transparent block, such as Dirt or Stone. These blocks can be powered from any neighbour and propagate power to any neighbour, however they will not further propagate their power through other solids or full blocks like the source block. In Minecraft, a solid may power adjacent solid blocks, however they may not propagate their power further if they have been powered by a source or solid. In this research we abstract from this, removing the ability for solids to propagate power to other solids entirely. This will not have any effect in the circuits we will experiment on as we will not be testing any circuit where this behaviour is relevant. Thus the additional logic would only create longer processing times.

- **Source** blocks are actually Redstone blocks in Minecraft. These blocks emit a constant maximum power of 15 in all directions and do not take input as they only have one state. This block will also not update for the same reason.

- **Wires** are in fact Redstone Dust. This block type propagates its power in all directions and can take input from all directions as well. As these wires propagate power along other wires, the propagated power will decrease by one for each wire in the network away from the source. Wires propagating power to a different component type will not decrease the power level. As the power level is maximal at 15, a line of Redstone Dust cannot propagate power across more than 15 blocks on its own. A piece of Redstone Dust at any power level greater than zero, will power another Redstone component just as well as a maximally powered piece of Dust, with a few exceptions such as Redstone Dust itself or Comparators, among other components not implemented in this Research.

- **Torches** refer specifically to Redstone Torches in Minecraft. These torches, along with emitting light, are able to interact with Redstone components. Their default state is powered, emitting 15 power in all directions except the direction of the block they are attached to, the opposite of their *dir* value. Redstone torches, just as any non-full-block must be placed on a block and cannot float in mid-air. All torches can be placed on any block horizontally with that block acting as a wall or on top of any block acting as a floor, but never on the bottom of a block. This direction is not enforced in the creation of circuits, however none of the circuits we tested on have impossible configurations in Minecraft. If their attached block is powered, Redstone Torches will turn off, emitting 0 power.

- **Repeaters** are oriented in one direction horizontally (along the X or Z axes) and only propagate power along that direction. They make use of the additional *target* variable, as these components feature a delay. Redstone Repeaters, when powered horizontally from the direction opposite of their *dir* value (opposite of output), will store that power for the duration of their delay, ranging from 1 to 4 ticks, and finally emit a power of 15 in their output direction. The *target* variable stores the delay that the Repeater is set to while the *age* variable increments from 0 as the power value is updated. Age continues to increment every tick until it is equal to *target*, upon which the repeater will emit power and reset its age. As their output direction is defined as a constant just as all other components, they constantly emit their current value in that direction. Thus during the delay, their power is set to -1 until the delay is over, setting their power to 15 which is emitted for one tick or as long as the input remains powered, after which it resets back to to 0. Components never accept a power level below 0, thus no other components will be affected during the delay.

- **Levers**, similar to torches, can be attached to a block in multiple directions, however unlike torches, they may also be placed on the bottom of a block; the ceiling. Levers can either be on or off and do not take any input from other Redstone components, meaning they can only be toggled through external interference. For the purpose of this research, levers may only be set in the initial state and will not update. While on, Levers will power the block they are attached to, which will then propagate its power further. A Lever that is powered off will do nothing.

When initializing a circuit configuration, we create new blocks in each cell where necessary, specifying any state variables that are not default for each block. Blocks where certain state variables do not have an effect, such as the *dir* and *target* variables in most blocks, simply have those values set to zero.

6 Experiments

This section will cover the experiments conducted in order to find an answer to the research questions.

6.1 Experimental Setup

This research aims to investigate the relationship between logical expressiveness and the verification of Minecraft Redstone circuits modelled as Cellular Automata. In order to do this, we will be

conducting experiments on certain Redstone circuits of varying complexity, each displaying some behavioural properties of Redstone. We will establish and prove these properties using formal logic, while identifying the minimal logical fragment sufficient to do so. This demonstrates both the correctness of the model according to its intended behaviour and the logical expressiveness required for different classes of properties.

For each circuit, we will be explaining the behavioural property it aims to show, as well as what logical verification is used to prove this, attempting to identify a minimal logical fragment required for this verification. Each experiment will output the number of formula evaluations performed to reach its conclusion as well as its execution time. The number of formula evaluations corresponds to the number of times the `check` function is called, either to perform some operator or through recursion in fixed point operators. These values provide an indication of the computational cost required by the verification process and offer additional context when comparing experiments.

6.2 Predicates and derived operators

We first define some predicates in Table 2 as well as some abbreviated logical operators in Table 3 used to simplify the verification formulas. For each predicate or abbreviation, we will show its corresponding logical formula and a short description.

Predicate	Formula	Description
isWire	FieldEq 0 3	Checks whether the block type of the cell is 3, corresponding to a wire. Similar predicates exist for all block types.
isPowered	FieldGT 1 0	Checks whether the power of the cell is greater than 0, i.e. whether the cell is powered.
notPowered	FieldEq 1 0	Checks whether the power of the cell is equivalent to 0.
powerEq p	FieldEq 1 p	Checks whether the power of the cell is equivalent to a given value p.

Table 2: Predicates defined for experiments

Abbreviated operator	Formula	Description
eventually	$\mu X.F \vee \boxtimes X$	The temporal least fixed point operator, finds cells for which a property holds at after some number steps in the circuit's evolution.
always	$\nu X.F \wedge \boxtimes X$	The temporal greatest fixed point operator, finds cells for which a property always holds forever throughout a circuit's evolution.
neighbor	$\bigvee_{d \in D} \Diamond_d X$	A disjunction of all neighbours, performs Space operator in all directions $d \in D$.
somewhere	$\mu X.F \vee \bigvee_{d \in D} \Diamond_d X$	Spatial least fixed point operator, finds cells for which there exists a path through neighbouring cells to a cell for which a property holds in the current state.
everywhere	$\nu X.F \wedge \bigwedge_{d \in D} \Diamond_d X$	Spatial greatest fixed point operator, finds cells for which a property holds for all cells reachable through its neighbouring cells in the current state.
wireSomewhere	$\mu X.F \vee (isWire \wedge neighbor(X))$	The same spatial least fixed point operator, but only considering wires (Redstone Dust) as valid neighbouring cells. Holds if a cell satisfying the property is reachable via a connected path of Redstone wire.
wireEverywhere	$\nu X.(isWire \rightarrow F) \wedge (neighbor(isWire \wedge X))$	The spatial greatest fixed point operator, restricted to connected Redstone wire cells. It holds if every reachable wire cell satisfies the property.
timeN	$\boxtimes^N F$	An abbreviated form for performing the Time operator N times.
spaceN	$\Diamond_d^N F$	An abbreviated form for performing N Space operations in a given direction d.

Table 3: Abbreviated logical operators for experiments

6.3 Experiments

We conduct experiments both for temporal and spatial properties. We will explain a few experiments in greater detail, describing their circuits and identifying a minimal fragment for the verification. All results will be shown in Table 4, however they will not all be explained in depth for brevity. All experiments are run on a three-dimensional CA grid with a size of $35 \times 35 \times 35$. The execution time may vary based on the world size.

6.3.1 Basic component properties

Wire propagation Wires propagate power to each other and to other components. This experiment is executed on a simple row of three wires connected to a source block. The final wire becomes powered after 3 timesteps; the source powers the first wire in step one and this power propagates along the row one at a time. We perform two versions of this experiment using different logical fragments. Namely basic modal logic and modal logic with a fixed point operator:

(Cell [3,0,0] , **timeN 3** (Conj isWire isPowered))

(Cell [3,0,0] , **eventually** (Conj isWire isPowered))

The experiment with the basic temporal operator only requires 6 evaluations to reach a conclusion, whereas the experiment with fixed points needs 23. As this is a very simple example, we see this property can simply be expressed without the need for fixed points. Using the more expressive logical fragment is more expensive computationally, however we can only make use of the repeated Time operator if we already know how many time steps are required for the property to become true.

Torch inversion Torches are always attached to a solid. When this solid block becomes powered, the torch should invert and shut off. This experiment is executed on a circuit containing a solid with an adjacent lever and a torch attached. The lever is switched on, which causes the torch to invert and shut off after 2 timesteps. Similarly to wire propagation, we perform two versions of this experiment as we know how many time steps it *should* take for our property to hold (torch turning off).

(Cell [1,1,0] , timeN 2 (Conj isTorch notPowered))

(Cell [1,1,0] , eventually (Conj isTorch notPowered))

These experiments require 5 and 17 evaluations respectively, with the fixed point, again, being more expensive in return for its expressiveness.

Repeater Repeaters should extend power propagation. A repeater has a set delay of 1 to 4 and, upon being powered, waits that many ticks before outputting a maximum power of 15. We perform a few tests to show the properties of repeaters. The experiments are executed on a circuit with a source going into two wires and through a repeater into two more wires. We first show repeater delay with two tests:

(Cell [5,0,0] , timeN 6 (Conj isWire isPowered))

(Cell [5,0,0] , timeN 10 (Conj isWire isPowered))

The first test checks whether the final wire is powered after 6 time steps, which is more than the length of the circuit, however it returns false as the repeater delay is still withholding the power. The second test checks the value after 10 time steps, which is enough for the delay to have completed and the power to have propagated. Similarly to the previous two experiments, we may use a fixed point variant of this second test as well, producing similar results. To show the signal extending property of repeaters, we perform one final evaluation on the circuit:

(Cell [5,0,0] , eventually (Conj isWire (powerGT 12))

This test checks whether the power on the final wire is greater than its expected value if it were a regular line of wires. We use a fixed point here, however we may use a smaller logical fragment with ten Time operators as we did for the second repeater test. The test passes.

6.3.2 Logical gates

NOT The inverting property of Redstone Torches allow for the creation of a simple NOT gate. We pass a lever into a wire through a solid with a torch attached and then place another wire next to the torch to obtain an inverted power. See Figure 6 for a visualisation of this CA. We show this behaviour with two tests to display each input case. Both tests are performed with the same verification formula:

(Cell [4,0,0] , eventually (always (Conj isWire isPowered)))

As expected, in the case in which the lever is switched on, the verification returns true, and in the case the lever is switched off, it returns false. This verification makes use of a mix of least and greatest fixed points (LFP and GFP respectively), however we may simplify it down to not using any fixed points at all if we know after how many time steps the circuit stabilizes. In this case, the circuit is simple and we can tell it is 4 at a glance. We show this modal test variant:

(Cell [4,0,0] , timeN 4 (Conj isWire isPowered))

This version only requires 7 evaluations as opposed to 85 for the positive case or 139 for the negative case. Unfortunately we can not always know when the circuit stabilizes at a glance, thus this is not a simplification we can often perform. In this case we can also reduce the verification to a single fixed point verification. We may leave out the LFP (eventually) and still produce reliable results if we know if and when the output wire becomes powered before reaching a stable state. If the verification only makes use of the GFP (always), it is still valid if the output wire is powered from the zero state or if we perform a Time operation until it becomes powered by the torch, in this case after 1 time step, as the output wire is only one cell away from the torch. If the input is powered, the output will become unpowered after some number of time steps and the verification will become false and vice versa for an unpowered input.

(Cell [4,0,0] , Time (always (Conj isWire isPowered)))

This version requires 24 evaluations in the on case or 68 in the off case, a great improvement from the double fixed point version, however it did require some knowledge about the circuit in order to replace the LFP with a time step. If the output wire had been further from the torch, we would have required more time steps and would have to know how many are required. We can not use only an LFP however, as without the GFP, we would only be checking whether the final wire is ever powered at least once; in both cases; either the wire being powered in the initial state, or after the torch propagates power to it, this is already true after 1 time step. The torch powers the final wire before any possible power could shut the torch off from the input, which would cause a LFP formula to return true before the circuit reaches a stable state from the input propagating to the output, thus it is an unreliable result.

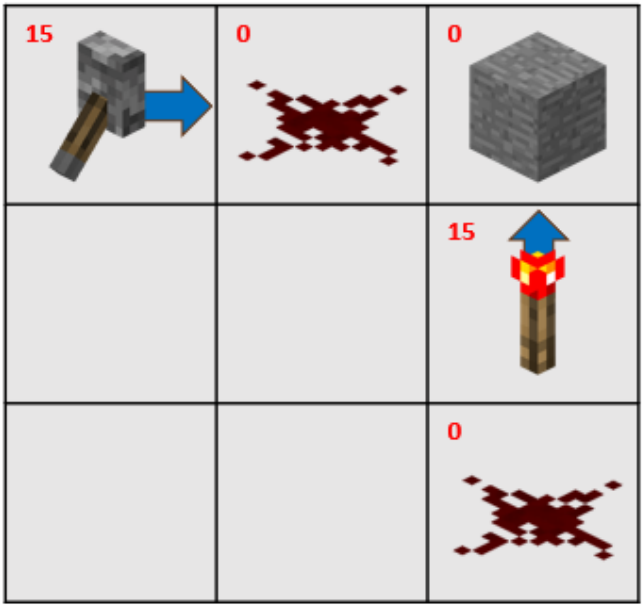

Figure 6: NOT Gate Cellular Automaton

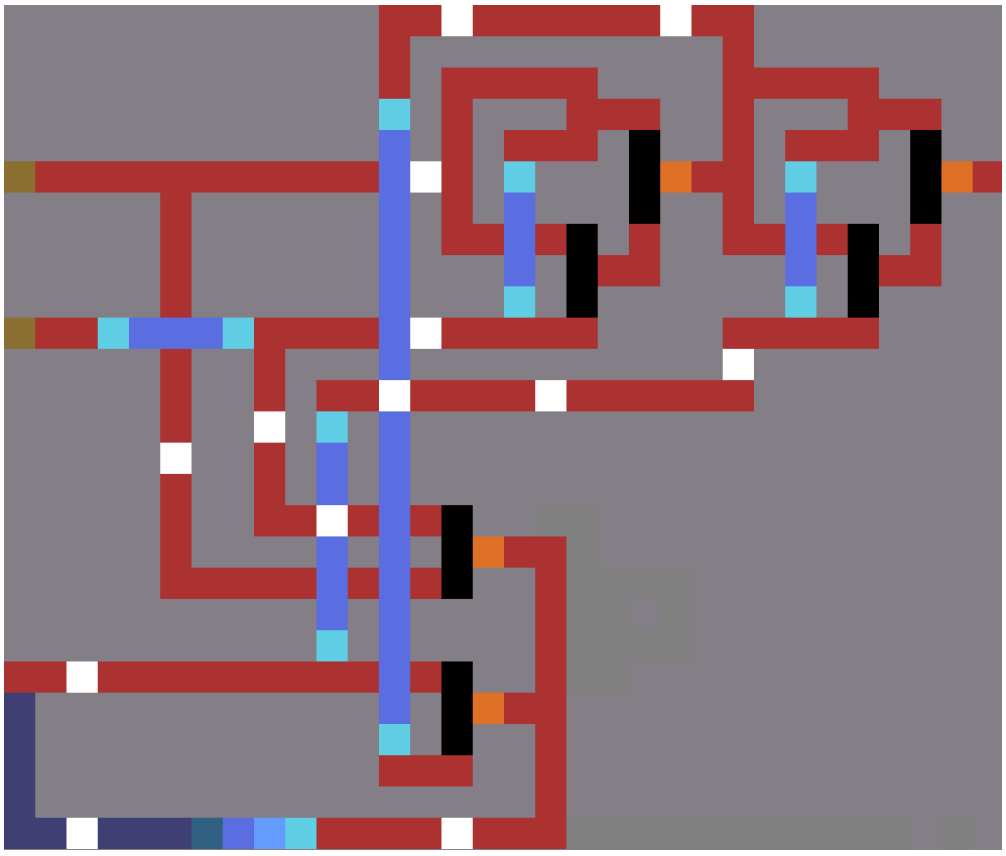

Figure 7: Adder circuit layer

6.3.3 Complex circuits

Adder Combining the designs for the AND and OR gates, along with a simple $(A' \vee B')$ gate, which is equivalent to an AND gate without the final negation, we can create a full bit adder. With an A and B input as well as a path for a carry input (C_{in}), we create an adder by linking each value through gates. Each layer of the adder corresponds to one input bit.

In Figure 7 we see a general schematic of one layer of the adder. The red squares correspond to wires; brown squares correspond to levers for input bits; blue squares correspond to elevated wires, or blocks with wires on top, with darker shades of blue corresponding to higher elevation; orange squares correspond to torches; white squares correspond to repeaters to ensure the power level never reaches 0 in long wire networks; black squares correspond to solids with wires or torches on top. The solids always appear in a row of 3 where the outer blocks have a torch on top and the middle block has a wire on top. These torches negate their incoming power such that the middle wire is only powered if either input is off. This is a NOR gate, which becomes an AND gate when another torch is added on the output, denoted with the orange squares.

The topmost brown square in the image corresponds to one bit of the A input and the second brown square corresponds to one bit of the the B input. The third wire near the left edge of the image corresponds to the carry input. The carry output runs along the bottom edge of the image and climbs up 6 blocks where the next layer may be created. The layers are 6 blocks apart to ensure no other elevated wires interact with the layers above them. A new layer is placed at an elevation 6 higher than the previous on top of a layer of solids to ensure there are no floating wires. The topmost layer corresponds to the most significant bit. Note that there is at least one row of padding on the right and bottom of the circuit to ensure the borders do not wrap and interact with opposing cells.

The output wire corresponding to the sum is the rightmost wire in the image. The sum is computed as $(A \vee B) \wedge (A' \vee B') = s_1$ and $(s_1 \vee C_{in}) \wedge (s_1' \vee C_{in}')$, which is equivalent to $(A \veebar B) \veebar C_{in}$. Then the carry output (C_{out}) is computed as $(A \wedge B) \vee (C_{in} \wedge s_1)$.

The C_{out} is then led to the next layer on top for the next bit to be computed. We performed two experiments on a 4-bit adder for two different computations.

(Cell [31,18,5] ,
eventually (always (Conj
(Conj isWire **isPowered**) (Conj
(spaceN 6 dirY' (Conj isWire **notPowered**)) (Conj
(spaceN 12 dirY' (Conj isWire **isPowered**))
(spaceN 18 dirY' (Conj isWire **notPowered**)))))))

(Cell [31,18,5] ,
eventually (always (Conj
(Conj isWire **notPowered**) (Conj
(spaceN 6 dirY' (Conj isWire **isPowered**)) (Conj
(spaceN 12 dirY' (Conj isWire **notPowered**))
(spaceN 18 dirY' (Conj isWire **isPowered**)))))))

The first experiment is run on an initial state with inputs corresponding to $7 + 3$ and the second on a state corresponding to $1 + 4$. Both verifications are essentially the same with the only difference

being whether each of the output cells is checked for `isPowered` or `notPowered` corresponding to a 1 or 0 bit respectively. The given cell points to the wire containing the sum of the most significant bit and each conjunction moves down in space by 6 steps from the initial cell to check the value of the next sum bit, as one layer of the adder is 6 blocks tall.

The verification checks that after some number of time steps, the current cell and the three cells below it in steps of 6, will always retain the given values representing the sum of the two inputs. For 7 + 3 this is 10, or 1010 and for 1 + 4 this is 5, or 0101.

This experiment makes use of both spatial and temporal operators and is quite expensive as it contains mixed fixed point operations with many conjunctions. These conjunctions and the spatial operator may be removed if the verification is performed for one cell at a time, however that would require running the full simulation four times until it stabilizes, which is more expensive for a complex circuit such as this.

(Cell [31,18,5] , eventually (always (Conj isWire notPowered)))
(Cell [31,12,5] , eventually (always (Conj isWire isPowered)))
(Cell [31,6,5] , eventually (always (Conj isWire notPowered)))
(Cell [31,0,5] , eventually (always (Conj isWire isPowered)))

The single evaluation with the spatial operator on 1 + 4 requires 3897 evaluations whereas the 4 separate evaluations require 2870 verifications. Interestingly, as each test is only checking a single cell, a much smaller number of operators are required to reach the desired result, however we notice that the execution time is much longer in this experiment compared to the one with the spatial property. We attribute this to a large overhead cost in running separate experiments, causing the verification with the spatial operator to be more efficient despire the increased expressiveness as it is able to perform the task in a single run. Removing the LFP is technically possible as it was for the NOT gate, however, it would similarly require knowledge about the circuit beforehand. Removing the GFP is not possible as the sum wires may become powered due to the A and B inputs before the Carry input arrives, which could then unpower it. If it becomes unpowered, it will still have been powered at some point, thus it will not be a reliable result without the GFP ensuring the circuit has stabilized.

Thus, while not the smallest fragment, this appears to be the most efficient reasonable fragment for evaluating the correctness of the adder among those we tested, as removing the spatial operator would in fact increase the computational cost as it would require splitting the evaluation into multiple duplicates.

6.3.4 Spatial properties

Global power decay As power decays along wire paths, it is necessary for all paths along connected wires to eventually decay to zero power or for the wire path to end. This experiment performs a spatial evaluation on a circuit with a single torch surrounded by wires horizontally and vertically upwards with solids in between each wire layer.

(Cell [16,0,16] , always (wireEverywhere (wireSomewhere (Disj notPowered wireEnd))))

This evaluation checks whether for all time steps, it holds that every wire reachable through paths across neighbouring wires from the current cell, will have its own path across neighbouring wires that eventually leads to either an unpowered wire or the end of a wire path.

Removing the always operator would make the evaluation only check the current circuit, which consists of a field of unpowered wires and a torch. As all wires are still unpowered, the evaluation would return true, but it would be an unreliable result. Removing the wireSomewhere operator would result in a check for all cells being either unpowered or the end of a wire path, which in this case will only be true for the initial state. Removing the wireEverywhere would make the evaluation only relevant to the current cell. As we are only moving along connected wires for both everywhere and somewhere, this is actually fine, because if a wire path exists to an unpowered wire or end of a wire, then all wires reachable from the current cell would have the same property, as the current cell is also necessarily reachable from those cells. The transitive property then states that those cells can also reach the same unpowered wire cell or end of wire path. This leads to a smaller fragment evaluation.

(Cell [16,0,16] , always (wireSomewhere (Disj notPowered wireEnd)))

From the results we notice this actually produces the same number of evaluations, however, it takes less time to execute. This may be attributed simply to external background processes producing noise in the results. This result seems to show that the wireEverywhere operator is entirely unnecessary in this evaluation and can thus be excluded as in the second formula.

Reachability The spatial operator may also be used for path finding. In a circuit of a maze of wires with only one wire path leading to a torch, we can traverse along wire neighbours until either reaching a torch or the end of the wire. We perform two evaluations following two different wire paths.

(Cell [0,0,13] , wireSomewhere isTorch)

(Cell [0,0,15] , wireSomewhere isTorch)

This evaluation checks whether from the given cell there is a path across wire neighbours that leads to a torch. The first evaluation follows the correct path. This evaluation can not be simplified reasonably as the path of the wires towards the torch is not visible at a glance. Thus the minimal fragment is modal logic with spatial and LFP operators.

6.4 Results and RQ Discussion

All experiment results can be found in Table 4

RQ1 Which behavioural properties of Redstone circuits can be expressed in the original modal logic, and which require explicit fixed-point operators? From this experimentation we can see that bounded behavioural properties can be expressed with basic modal logic. Knowing the boundaries of the circuit and how many time steps, for example, are required to reach a desired state allows for simple notation using basic modal logic. However, unbounded circuits or circuits for which we do not have much knowledge seem to require fixed points in some way, as we require the recursion.

The experiments support this difference. When the number of time or space steps was known, the original modal operators were enough. For example, wire propagation could be checked with

`timeN 3`, which needed only 6 formula evaluations. Torch inversion could also be checked with two time steps and needed 5 evaluations. The repeater delay could be checked in the same way when the delay was known. Fixed points were therefore not needed for these bounded properties.

Fixed points were useful when we did not know how many steps would be needed. The `eventually` operator was used when a property should become true at some point in the future, while `always` was used when a property should stay true. The spatial fixed point used by `wireSomewhere` allowed the checker to find a connected path through any number of wires.

Overall, the experiments show that the original modal logic is enough for bounded properties when the required number of steps is known. Fixed points are useful when the number of time or space steps is not known beforehand. However, a property that can be written with a fixed point can sometimes also be written with normal modal operators if enough is known about the circuit.

RQ2 How does restricting logical expressiveness affect the computational performance of model checking? In most cases, restricting the fragments where possible allows for faster, less expensive computations. However this mainly applies to the removing of fixed point operations from a verification. If we were to remove some other operator, it may result in a solution becoming more expensive.

The results also show that the number of formula evaluations does not always match the execution time. For example, wire propagation needed 6 evaluations with bounded temporal logic and 23 with a fixed point, but the fixed-point version was faster (884790 μs compared with 1427487 μs). Torch inversion shows the same pattern: the fixed-point version needed 17 evaluations instead of 5, but was still faster (426317 μs compared with 953568 μs). This could mean that one formula evaluation is not always the same amount of work. Some operators may be more expensive to evaluate than others.

The adder also shows that fewer evaluations do not always mean a faster verification. When the spatial operator is removed, the property can still be verified, but the verification takes much longer even though only 2870 evaluations are performed. This suggests that the individual evaluations are more expensive in the restricted formula. Without the spatial operator, the checker verifies the formula for each of the 4 cells individually, which requires more work during each evaluation, likely due to overhead from repeatedly running a new experiment.

This means that the number of evaluations is not enough to predict verification time. A formula can use fewer evaluations but still take longer if each evaluation requires more work. Therefore, restricting the logic does not always make the verification faster.

The spatial experiments show another case where removing an operator does not help much. When `everywhere` was removed from the global power-decay verification, the number of evaluations stayed the same at 65066. The execution time actually increased from 41700402 μs to 44556157 μs. This shows that using fewer operators does not always mean that the verification will be faster.

RQ3 Which logical fragments provide the best balance between expressive power and verification efficiency? The most efficient fragment would contain no fixed points at all, however they are often not avoidable, thus the most practical solution is typically a single fixed point. A least fixed point can often be replaced with a small number of Time or Space operators. A greatest fixed point is typically harder to replace, however.

This is supported by the NOT gate results, where a fragment containing only the fixed-point operator required by the property provided a better trade-off than the full logic. The results

therefore suggest that the optimal fragment is property-dependent rather than a single fragment that is optimal for all circuits.

The adder also shows that the smallest logical fragment is not always the fastest. The spatial operator allowed all four output bits to be checked together. Without it, the output cells had to be checked separately. This increased the total amount of work. The best fragment therefore depends not only on how many operators are used, but also on how the property has to be checked.

RQ4 What logical features are necessary to reason about signal propagation, stabilization, persistence, and oscillation in finite Redstone systems? For signal propagation, basic modal logic is typically enough, however on an arbitrarily long or complex circuit, fixed points may be required to find unknown cells that may not properly be propagating, for example for debugging a circuit.

To prove stabilization and persistence, a greatest fixed point (GFP) is required. A greatest fixed point is needed to prove that the state is unchanging in a single verification, however, depending on whether it is known when the circuit stabilizes, modal logic with temporal operators or a least fixed point (LFP) are also required. If we know when it stabilizes, the LFP is unnecessary, as we can simply perform Time operators until the stable state.

Oscillation requires a mixed temporal fixed point to be proven. Just a GFP is not enough, as an oscillating circuit will be changing its state throughout, thus is not persistent on its own. Only a LFP is also not enough, as this can only prove whether a state has been reached once, not whether it continues to do so. With both fixed points combined as always eventually, we can prove oscillation. This checks whether it holds for all time steps that some property will hold after some number of time steps. Eventually always checks for persistence after some number of time steps, and is thus incorrect.

This distinction is also visible in the experimental results: when the propagation depth was known, a bounded sequence of temporal or spatial operators was sufficient, whereas fixed points became necessary when the required depth was not known beforehand.

The experiments support these differences, although not every property was tested with its own circuit. Signal propagation was tested with the wire propagation and repeater experiments. When the number of steps was known, normal temporal and spatial operators were enough. The fixed-point version allowed the same type of property to be checked without knowing the number of steps first.

Persistence was tested through examples such as source persistence and the greatest fixed point used in the NOT gate experiments. A greatest fixed point is useful when a property must stay true in all future states. Stabilisation can be checked by showing that the circuit eventually reaches a state where the property stays true. If the number of steps is already known, fixed points are not needed and a finite sequence of temporal operators can be used instead.

The oscillation claim needs to be treated more carefully. A formula such as $\nu X.(\mu Y.(\varphi \vee \boxtimes Y) \wedge \boxtimes X)$ can be used to describe repeating behaviour. A simpler "always eventually" formula can also describe a property that keeps becoming true. However, no dedicated oscillating Redstone circuit was tested in these experiments. Therefore, the experiments do not directly prove that oscillation can be verified. A future experiment using a Redstone clock or another repeating circuit would be needed to test this. A clock circuit was attempted when creating the experiment cases for this research, however we could not obtain correct results in all cases for some reason, thus the experiment output was unreliable.

Main RQ What are the minimal logical fragments that are sufficient to verify relevant behavioural properties of a CA modelling Redstone systems, and how does restricting logical expressiveness affect verification performance? The experiments demonstrate that the minimal logical fragment depends on the behavioural property being verified.

For bounded properties with a known required evaluation count, the original modal logic is sufficient and results in the lowest verification cost. When reasoning about behaviours over an unknown number of time steps or recursive spatial paths, LFP and GFP operators become necessary. In some cases, such as the NOT gate, intermediate fragments using only one fixed-point operator are more efficient than the full logic.

Overall, restricting logical expressiveness generally reduces the computational effort of model checking, but only when the restricted fragment remains expressive enough to prove the desired property. Thus, the most effective verification method is to use the smallest logical fragment capable of expressing the desired property, rather than always employing the full modal fixed-point logic.

Overall, the experiments show that there is no single smallest fragment that works best for every Redstone property. The required fragment depends on the property being checked. Bounded properties can use the original modal operators when the required number of steps is known. When the number of steps is unknown, fixed points may be needed. Spatial fixed points are useful for properties that need to search through an unknown number of connected cells. Properties that combine eventual behaviour and persistence can need both least and greatest fixed points.

The experiments also show that using fewer logical operators does not always make verification faster. Wire propagation and torch inversion both had more evaluations with fixed points but still had lower execution times. The adder became more expensive when the spatial operator was removed because the verification had to be repeated for different output cells. The global power-decay experiment also took longer after removing an operator, even though the number of evaluations stayed the same.

The main conclusion is therefore that the best logical fragment depends on both the property and the verification method. Using the smallest possible fragment is a good starting point, but it is not always the fastest choice. Fixed points are useful when we need to reason about an unknown number of steps, while normal modal operators are simpler when the required number of steps is already known.

These conclusions only apply to the Redstone CA model and the checker used in this thesis. The experiments show a trade-off between logical expressiveness and verification cost for the circuits that were tested. More circuits, more tests of different fragments, and repeated measurements would be needed to know whether the same results apply to other Redstone systems.

Experiment	Evaluations	Time (μs)
Wire propagation	6	1427487
Wire propagation FP	23	884790
Torch inversion	5	953568
Torch inversion FP	17	426317
Source persistence	13	911667
Repeater delay unfinished	9	2816102
Repeater delay finished	13	4692849
Repeater delay finished FP	65	3608232
Repeater signal boost	65	2460489
Lever on	68	4051032
Lever off	6	198415
NOT 0	77	4099597
NOT 1 Modal	7	1143545
NOT 1 GFP only	24	1030407
NOT 1 both FP	157	7051231
AND 00	157	6542869
AND 01	157	7039687
AND 11	161	7793437
OR 00	17	450336
OR 01	107	5696082
OR 11	101	4694972
XOR 00	243	10619694
XOR 01	287	13455471
XOR 11	299	13234527
Adder 7 + 3	2496	83526455
Adder 1 + 4	3843	39097360
Adder 1 + 4 no space	2870	148183594
Global power decay	65066	40488997
Global power decay w/o everywhere	65066	38342331
Path finding right	1870	883025
Path finding wrong	1896	919137

Table 4: All experiment results

7 Conclusions and Further Research

In this research, we investigated the formal verification of Minecraft Redstone circuits by modelling them as Cellular Automata and reasoning about their behaviour using fragments of modal fixed-point logic. By extending an existing Haskell verification framework with implication and least and greatest fixed-point operators, it became possible to verify properties such as signal propagation, persistence, stabilisation, oscillation, and spatial reachability.

The experiments show that the choice of logical fragment depends on the property being verified. Basic modal logic is sufficient for bounded properties where the required number of time steps is known, while fixed-point operators are necessary for expressing recursive temporal and spatial

properties with lacking knowledge of the system's behaviour. Although these more expressive fragments generally result in a higher computational cost, they allow for verification of more complex properties.

Future work could extend the Redstone model with additional components and update semantics more accurate to Minecraft, allowing larger and more realistic circuits to be verified. Further optimisation of the model-checking algorithms may also improve the efficiency of fixed-point verification. Additionally, there is work to be done to enable the uploading of circuits from text files or even Minecraft world files. Currently circuits are still typed out manually.

Overall, this research demonstrates that Cellular Automata and modal fixed-point logic provide a practical framework for formally modelling and verifying the behaviour of Redstone circuits, while emphasizing the trade-off between logical expressiveness and verification performance.

References

[1] Henning Basold, Chase Ford, and Lulof Pirée. An expressive coalgebraic modal logic for cellular automata. 2026. Preprint.

[2] Julian Bradfield and Colin Stirling. Modal mu-calculi. In Patrick Blackburn, Johan van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 721–756. Elsevier, 2007.

[3] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[4] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[5] Cornell Mathematics Explorations Center. Lesson 6: Cellular automata. https://pi.math.cornell.edu/~lipa/mec/lesson6.html, n.d. Accessed: 2026-06-08.

[6] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[7] GoldenStack. Minestom cellular automata, 2024. GitHub repository. Accessed: 2026-06-08.

[8] itsfrank. Minecrafthdl, 2021. GitHub repository. Accessed: 2026-06-08.

[9] Jackrekirby. Minecraft cellular automata (typescript), 2024. GitHub repository. Accessed: 2026-06-08.

[10] Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134, 1928.

[11] Dexter Kozen. Results on the propositional -calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[12] Ben Land. Rule 110 cellular automatons and universality in minecraft redstone, 2021. Accessed: 2026-06-08.

[13] Mattbatwings. Batpu, 2023. Supporting code for an 8-bit computer built in Minecraft.

[14] Mattbatwings. I made a working computer with just redstone!, 2023.

[15] Minecraft Wiki contributors. Redstone - minecraft wiki. https://minecraft.wiki/w/Redstone, 2026. Accessed: 2026-06-04.

[16] Minecraft Wiki contributors. Tick - minecraft wiki. https://minecraft.wiki/w/Tick, 2026. Accessed: 2026-06-04.

[17] Mojang Studios. Minecraft official site. https://www.minecraft.net/en-us, 2026. Accessed: 2026-02-20.

[18] Sammyuri, Uwerta, and StackDoubleFlow. I made minecraft in minecraft with redstone!, 2022.

[19] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[20] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, IL, 1966.

[21] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.