The University of Manchester
Department of Computer Science
Project Report 2022

# A Universal Cellular Automata Simulator

Author: Rakha Djokosoetono

Supervisor: Uli Sattler

## Abstract

This report encapsulates a project aimed to create a Universal Simulator for Cellular Automata. The motivation in creating this project is presented upfront in the beginning, followed with the necessary concepts and background knowledge necessary to build the tool. The project's architecture will be described, and its implementation will be discussed. Design decisions are also justified as to why a specific technology was used, and perhaps why others were not. Feedback for this project is then provided by the intended audience, i.e. University Students in Computer Science, to gain insight to its usability. This project report will finish with a summary of achievements, reflection, and potential extension work.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

To simulate is the act of imitating real world processes or systems over time [3]. It is also well understood that simulations are important in a myriad of contexts. In Science, Technology and Engineering subjects, simulations may be used for performance optimization or tuning, [33] or also ensuring systems provide acceptable levels of safetiness through the discipline of safety engineering [35]. Simulations may also appear in scientific modelling of natural systems and economics [25]. In general, however, simulations are used whenever the real system in question cannot be interfaced with (for whatever reason) [33].

One type of simulating is through the use of computer simulations. Simulating using computers started garnering popularity as far as World War II, when Jon Von Neumann and Stanislaw Ulam were investigating the complex behavior of neutrons during the Manhattan Project [29]. Since then, simulations and modelling have undoubtedly become paramount in a variety of fields today, especially in dealing with complex systems.

Complex systems revolve around us - and oftentimes they can be generated by simple mechanisms [32]. There are a handful of ways where such systems can be modelled using simple rules, but one such way of modelling complex systems can be done by cellular automata.

Commonly used as computer simulations, cellular automata (CA) is a discrete computation model studied in Automata Theory [38]. For most simulations, CA would often appear as a grid of cells, with each cell in having one state from a finite set of states (e.g. *on* state, or *off* state). The states of these cells in the aforementioned grid may change each time-step (or generation), according to a set of transition rules and its neighbouring cells.

CAs have demonstrated their simulation prowess in a variety of fields. They've oftentimes been applied to simulate systems in physics and biology [38], forest fire propagation [10], population dynamics [23] [30], and can even be implemented in game development [19] among many others. Therefore, on the basis of the variety of a cellular automata's uses, this project's aim is to build a universal cellular automata simulator, where the user can define their own cellular automata rules in how the simulator would step.

## 1.2 Objectives

### 1.2.1 Objective 1: Research & Development

To achieve the aim of building a universal cellular automata simulator, background research is essential in understanding the underlying concepts of a cellular automata, its different types, and identify its various applications in addition to understanding what technologies are there to help me. The simulator should also allow the users to input their own rules in the simulation.

Therefore, an outline of my objectives in the scope of Research would be to:

- gain a deeper understanding of cellular automata - understand what they are and different types of them, how they operate, their various applications, and how they are typically implemented in code
- hone and develop skills in JavaScript (JS) and HTML,
- browse various existing frameworks that might work as building blocks for the application
- come up with a rule format that is readable for humans

After research is done, then the goal will be to build a universal cellular automata simulator.

### 1.2.2 Objective 2: Evaluation

The final objective of the project is to evaluate the usefulness of this tool to the computer science university students and gather feedback on it. I define usefulness in this case as how easy the users can understand how to use the simulator. In particular, I'd like to obtain feedback on how easy to understand and use the rules are. Qualitative information will be gathered through small-scale interviews.

The population of computer science students is chosen as the target audience. This is because the end product involves interfacing with text written in the JSON format, and of course cellular automata. The aforementioned group is therefore more likely to understand the JSON format and cellular automata.

Qualitative evaluation will be performed by short interviews where the user will use the system. Without the appropriate user feedback, the system's usability will forever remain subjective speculation to me, the developer.

# Chapter 2

# Background Theory

## 2.1 Cellular Automata

The meaning of *cellular*, in this context, is ”relating or consisting of living cells”. On the other hand, the word *automata* originates from the Greek language meaning ”self-acting, self-willed, or self-moving”. By combining the two words, the term Cellular Automata (henceforth CA) etymologically means living cells that move or act by its own willingness. As briefly touched upon in the previous section, CA are a grid of *living* ”cells”, each cell pertaining to a particular ”state”.

A CA consists of the following necessary components: the cells, states, neighbourhood type, and transition rules. Additionally, the term ”generation” (pl. ”generations”) also need definition. Their meanings are as below:

- **Cells**: The meaning of a cell in CA can be taken from its definition in biology, meaning the smallest, structural unit of an organism. In the context of CA in this project, this means it is the smallest unit of the grid.

- **States**: In a CA, each cell has a state. Systems that are *stateful* means they are designed to remember preceding events [14]. In the context of CA, this information on preceding events is encapsulated in each cell’s state, that may change depending on a certain set of *transition rules* (see definition below). The *state space* is the set of states a system can occupy. As CAs are discrete models [38]

- **Neighbours**: The neighbours of a cell are essentially cells that live in close proximity with the cell in question. There are different types of neighbourhoods, but in the context of this report, these neighbours can be defined simply as other cells that directly touch the cell in question (i.e. they are *neighbouring*).

  A more detailed description of this neighbourhood type can be found in section 2.1.3.

- **Transition Rules**: Transition rules are the rules that determine the next state of each cell with respect to the current state of the cell and the states in its neighbourhood. In most cellular automata, these are typically expressed as mathematical functions [34]. Typically, these rules are applied to the entire CA grid simultaneously, though there are exceptions to how the rules are applied to a specific subset of CA [31]. These exceptions will not be discussed in this paper.

- **Generation**: A ”generation” in a CA grid represents how many times the transition rules have been applied to each of the grid’s cells. The initial state of a CA grid is represented at generation zero, (typically noted as generation time $t = 0$). A new generation is therefore created by advancing $t$ by 1 (i.e. $t = t + 1$). This implies that the transition rules are applied once to the grid. Incrementing $t$ by integer $n$ will therefore apply the transition rules to the CA a number of $n$ times.

In retrospect, CA can be defined asa grid of ”cells”, where each cell has a ”state”, and in which those cells’ states’ evolve according to the a certain set of ”transition rules”. Cellular automata may operate in one, two, or more dimensions. However, the 2D variant will be the model implemented in this project.

### 2.1.1 Totalistic Cellular Automata

Totalistic CAs are a special type of CAs, in the way that their state transition rules operate in a very specific manner. Extending the definition of transition rules from the section above, totalistic CAs have transition rules that determine the next state of each cell with respect to a certain number of states present in the cell in test’s neighbourhood [39]. An example of a totalistic CA is the game development map generation application (more on 2.2.4).

If the transition rules in a totalistic CA also consider the state of the cell in test, then the CA can be labelled an *outer* totalistic CA. *Outer* totalistic CAs mean that the state of a cell at generation $t$ depends on both its own state, and the total of its neighbours both at generation $t - 1$ [15]. An example of an outer totalistic CA is Conway’s Game of Life.

In essence, the only difference between a totalistic CA and an *outer* totalistic CA is that the latter determines the next state of the cell while considering its current state as well.

### 2.1.2 Probabilistic Cellular Automata

Stochastic or Probabilistic cellular automata (henceforth PCA) outline the special state transition rules in a given CA. This means that instead of cells transitioning to a state based on the number of neighbours they have (and their states), each cell has a probability $P$ to transition to another state [20]. This special transition rule is used for Forest Fire Propagation models, particularly the model implemented by Freire [10], such that cells that are flammable have a certain probability of burning down.

### 2.1.3 Moore Neighbourhoods

The Moore-type Neighbourhood can be defined as a two-dimensional lattice. In a square grid, it contains the central cell and the eight immediate cells touching bordering it [36]. This type of neighbourhood is one of the two most common types of neighbourhoods in CA. It is visualized in the following figure:

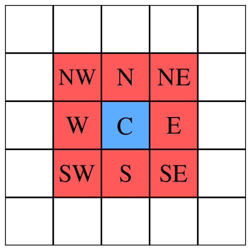


Figure 2.1: Moore Neighbourhood in a 2D Square Grid [38]

Moore-type neighbourhoods can additionally consider a specific range or depth.

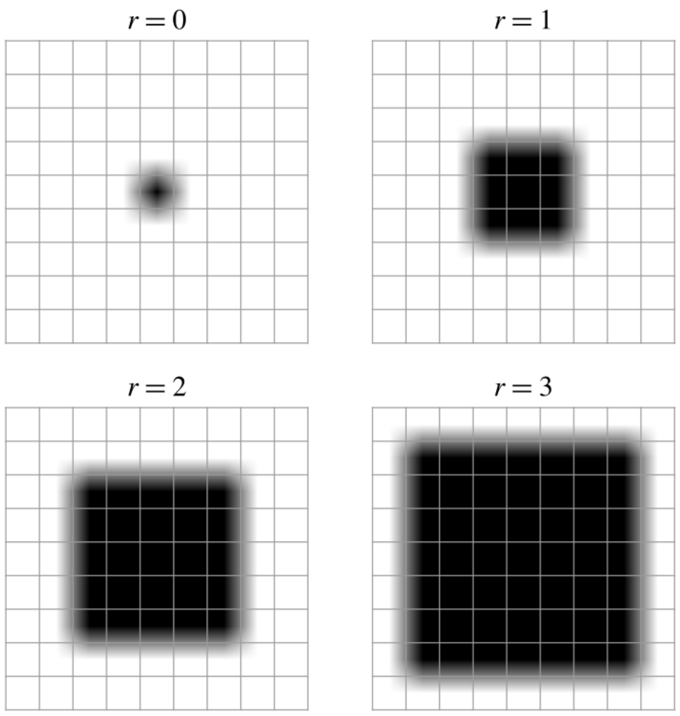


Figure 2.2: Moore Neighbourhood with Ranges [38]

## 2.2 Existing Applications of 2D Cellular Automata

CAs have been implemented in a vast number of fields due to its reliance on relatively simple rules. These fields are briefly touched upon in the first chapter of this report. Furthermore, applying cellular automata on different areas of knowledge rely on manipulating the rules. This would suggest quite a substantial devation in rules when compared to conventional, more popular CAs such as Conway's Game of Life.

### 2.2.1 Conway's Game of Life

Conway's Game of Life (henceforth CGOL) is a two-state CA with a special set of transition rules. Developed by John Conway in 1970, CGOL is arguably the most popular two-dimensional cellular automaton [11]. It is an example of an outer totalistic CA. State transitions rely on Moore-type neighbourhoods, with deterministic rules.

The state transition rules for each cell in a CGOL can be written in the form of if-statements:

- **IF** the cell is *alive* and has two or three live neighbours, **THEN** it remains alive in the next generation
- **IF** the cell is *dead* and has exactly three live neighbours, **THEN** it becomes a live cell in the next generation
- **ELSE** the cell dies, or stays dead

Otherwise the following flowchart visualizes how the rules may work:

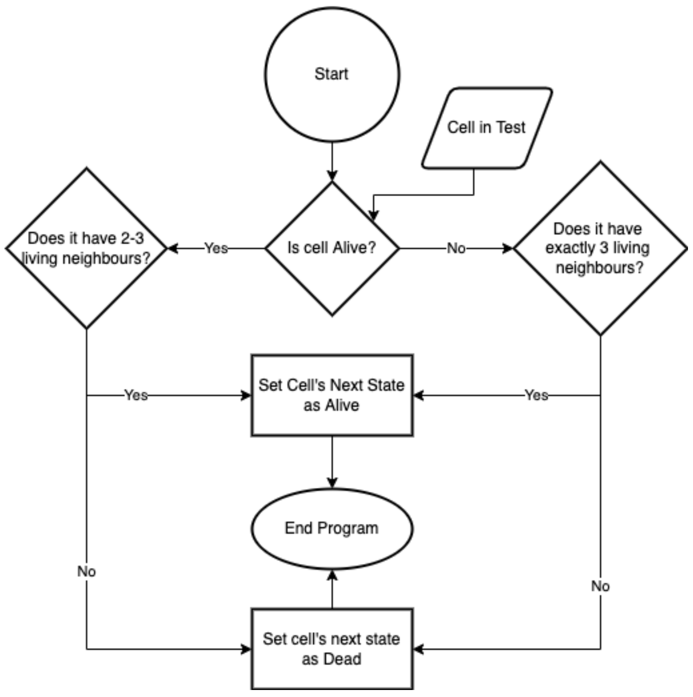

Start
Cell in Test
Is cell Alive?
Yes
No
Does it have 2-3 living neighbours?
Does it have exactly 3 living neighbours?
Yes
Yes
Set Cell's Next State as Alive
End Program
No
No
Set cell's next state as Dead

Figure 2.3: CGOL Rules Flowchart

CGOL is Turing Complete. [28]. The universality proof of CGOL in being Turing Complete was demonstrated by creating signals using the aforementioned Gliders, and combining them to create conventional logic gates (i.e. AND, OR, NOT) [27].

### 2.2.2 Forest Fire Propagation

CAs have extensively been applied in forest fire propagation. Their models typically implement two-dimensional probabilistic cellular automata with three or more states. The CA model outlined by [1] make use of a square grid, with a potential of fire propagating to its eight nearest neighbours (i.e. Moore-type neighbourhoods). The rules according to the aforementioned journal are as follows:

1. A cell that cannot be burnt stays the same
2. A cell that is currently burning down will completely burn down at the next state
3. A burned cell cannot burn again
4. If a cell is currently burning, and its neighbours include cells containing vegetative fuel (i.e. 'burnable' state), then the fire can propagate to said neighbour with a certain probability.

The probability in this case may be affected by a myriad of factors. The article by Freire [10] has modelled propagation probabilities based on wind, slope shape, and vegetation density.

The CA model outlined above, with probability functions integrated in its transition rules, is used to simulate the 2012 Algarve fire of Southern Portugal [10]. The same model, after a number of simulations, has also demonstrated an added value to assist in firefighting resource allocation in the event of a forest fire [10].

### 2.2.3 Population Dynamics

In the study by Mavroudi [23], CAs were used to model population dynamics in the city of Salonica, Greece. Most CAs that are purposed to modelling population growth are mostly developed through trial and error (essentially heuristics) [40]. In the context of city development, the rules in this CA are heavily influenced by its ecosystem. No city is regular, and these organic growths in population are usually affected by historical events and random decision making. [5]. CAs have been used in urban population predictions since the late 1980s [7].

Regarding urban growth and development, the CA rules are still a research topic [6]

### 2.2.4 Game Development: 2D Map Generation

Two-dimensional maps in games can often generated using CA. One of the CAs that can be used for map generationcan be a two-dimensional, two-state totalistic CA with deterministic state transition rules observing their Moore Neighbourhoods [16]. A common rule according to Johnson [16] for two-dimensional map generation is as such:

- **IF** a cell has more than four `wall` neighbours, **THEN** it turns to a `wall`

- **ELSE** the cell turns to a `floor`

The above is commonly used as cavern system generations in a game [2]. Another implementation of CAs in Game Development is a three-state genotype-to-phenotype based CA to recreate the Dune 2 Real-Time-Strategy map. In this implementation, however, the map is generated via genotype represented by matrices. CAs are only used here to create higher resolution maps through a stochastic process [22].

## 2.3 Potentially Useful External Libraries and Tools

To streamline the project development process, there are a number of libraries that may help as building blocks for the end product:

### 2.3.1 JSON and JSON Schema

JSON (short for JavaScript Object Notation) is a common data interchange format. JSON can contain JSON entities on its own, and is easy for humans to write, and computers to parse [4].

JSON Schema on the other hand is a special type of JSON that can be used to validate whether another JSON meets the requirements specified on the schema. It is a schema language. Validators typically validate JSON entities against a schema that is supplied, such as in a MongoDB database instance or API requests [26].

Essentially, JSON Schema is a response to the lack of standardized metadata or schema language for JSON - the most popular data format for API requests and responses. However, this is still under research and development [26].

### 2.3.2 P5JS

P5JS is a JavaScript library, utilized for 'creative coding'. This library is targeted primarily for audiences with an intent to draw, and therefore a myriad of its keywords rotate on the concept of sketching and drawing on a canvas [24]. This library may be useful in visualizing the CA.

### 2.3.3 TV4

Tiny Validator 4 (TV4) is a JS library written for use with JS or HTML/JS applications (typical installation using NPM or Yarn) [21]. This library enables the use of JSON validation with a specified JSON Schema. If there were a validator component in the project, this library will surely prove useful.

### 2.3.4 Git, GitHub and GitHub pages

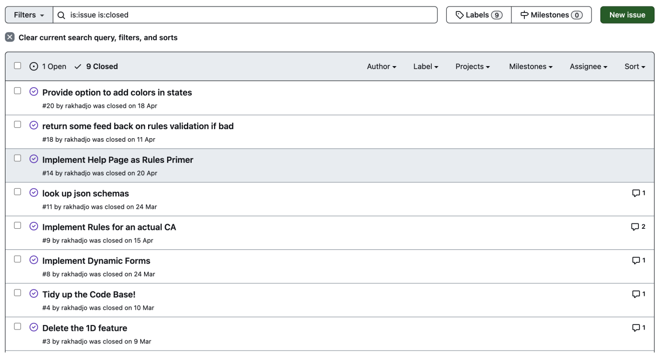
Every step of the development process is synchronized with GitHub version control. Proper workflow is maintained to the best of my abilities, and this has made the development process easier. The final project is hosted on GitHub pages for everyone to see and to provide room for evaluation, using a custom subdomain purchased from Namecheap. The project can be found at `https://projects.rakhadjo.me/y3-project`

# Chapter 3

# Planning and System Design

## 3.1 Planning and Work Management

The project was synchronized with GitHub version control from the very beginning. GitHub issues were used to keep track of features that are to be implemented in the project, in addition to patches and fixes.



Provide option to add colors in states
#20 by rakhadjo was closed on 18 Apr

return some feed back on rules validation if bad
#18 by rakhadjo was closed on 11 Apr

Implement Help Page as Rules Primer
#14 by rakhadjo was closed on 20 Apr

look up json schemas 1
#11 by rakhadjo was closed on 24 Mar

Implement Rules for an actual CA 2
#9 by rakhadjo was closed on 15 Apr

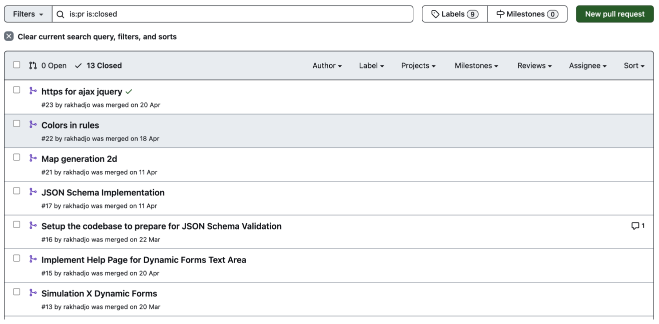
Implement Dynamic Forms 1
#8 by rakhadjo was closed on 24 Mar

Tidy up the Code Base! 1
#4 by rakhadjo was closed on 10 Mar

Delete the 1D feature 1
#3 by rakhadjo was closed on 9 Mar

Figure 3.1: GitHub Issues for Tasks Tracking

Each issue calls for its own feature branch, and each feature is then integrated by merging pull requests from their respective branches to the `development` branch.



https for ajax jquery ✓
#23 by rakhadjo was merged on 20 Apr

Colors in rules
#22 by rakhadjo was merged on 18 Apr

Map generation 2d
#21 by rakhadjo was merged on 11 Apr

JSON Schema Implementation
#17 by rakhadjo was merged on 11 Apr

Setup the codebase to prepare for JSON Schema Validation 1
#16 by rakhadjo was merged on 22 Mar

Implement Help Page for Dynamic Forms Text Area
#15 by rakhadjo was merged on 20 Apr

Simulation X Dynamic Forms
#13 by rakhadjo was merged on 20 Mar

Figure 3.2: GitHub Pull Requests for Feature Merging

A pull request to the `master` branch from the aforementioned `development` branch is opened from the very beginning, and merged upon the code freeze for submission.

## 3.2 Features

The main idea is to create a user interface which should be usable by university students in computer science, with some exposure to Cellular Automata. Note that this project does not aim to introduce the concept of cellular automata to its users. This means that provided the idea of cellular automata and a brief overview of how to use the rules, the user should be able to manipulate the rules and use the simulator.

### 3.2.1 Functional Requirements

Functional requirements outline the bare minimum as to how the system should behave. This project requires the system to, at the bare minimum and for all kinds of CAs:

- **Graphical User Interface**: The GUI will take inspiration from an IDE, and be as ergonomic as possible.
- **Modify the CA rules**: Editable through the Rule Editor on the left hand side of the GUI.
- **Change any of the state's colors**: This will be done by modifying the rules' contents in the `$_meta` tag, for preference or real-world visualization.
- **Change the CA rules**: Using the rules' contents from the text editor.
- **Parse and Validate JSON Rules**: Potentially TV4/JSON Schema implementation.
- **Count Neighbours of Each Individual Cell**: CAs determine next state based on neighbour's states.
- **Determine next state**: Provided the rules and the neighbour stats.

### 3.2.2 Non-Functional Features

Non-functional features outline how the system is meant to behave, while describing its limits in its functionality. In the context of this project, the non-functional features are as follows:

- **24/7 Availability**: Hosted on an always-on system.
- **Platform Independence**: Hosted on GitHub Pages, accessible and usable to everyone on the internet.

## 3.3 Language Selection

Due to the time constraint in the project, the final software was written in HTML and JavaScript. This allows the project to be instantly deployed as a static web page during testing, in addition to providing instant availability to the project only through a web browser without any unnecessary downloads.

Java and JavaFX was considered at one point, but JavaFX support online was relatively minimal. Additionally, different versions Java installations wouldn't have given the final project the same level of flexibility in deployment. Installation would therefore be impractical.

Despite the tech stack being entirely front-end, I am defining the term ”front end” as code that functions to change the user interface, and ”back end” as code whose functions are not directly seen.

## 3.4 Architectural Diagram

As the details of each components and micro-components are discussed in the sections following, the below diagram represents the general architectural flow, alongside the names of the components.

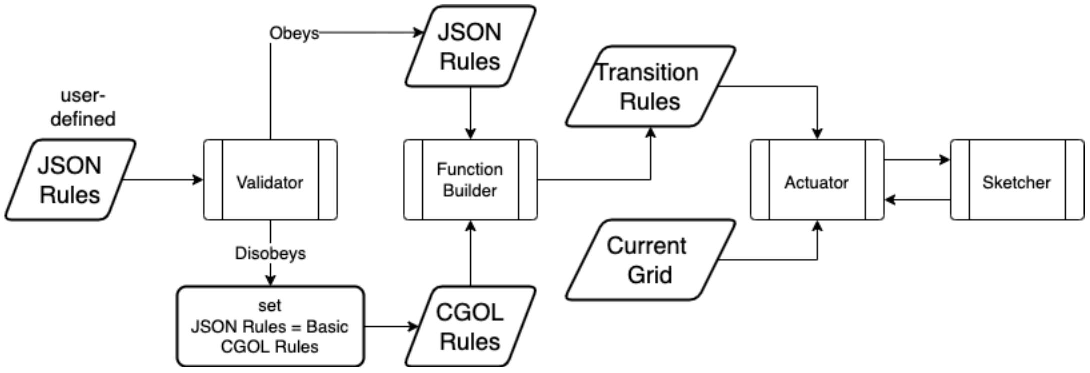


Figure 3.3: Architectural Diagram

It is important to note that the validator, function builder, actuator and sketcher components are displayed as ’predefined’, because its implementation will be shown here soon.

Some components are not present in the diagram. That is because the dynamic rules format are somewhat related to the Validator, and the GUI overarches everything.

## 3.5 Architectural Components and Implementation

The following subsections outline the different components working in tandem to fulfill the described features above, and justifying the decisions that were made. Additionally, the final implementation of each components are described in this section.

### 3.5.1 Graphical User Interface (GUI)

The GUI is designed to hold three main components to the system: the CA simulator, the rules editor, and the control buttons. An early sketch of the GUI is provided below:

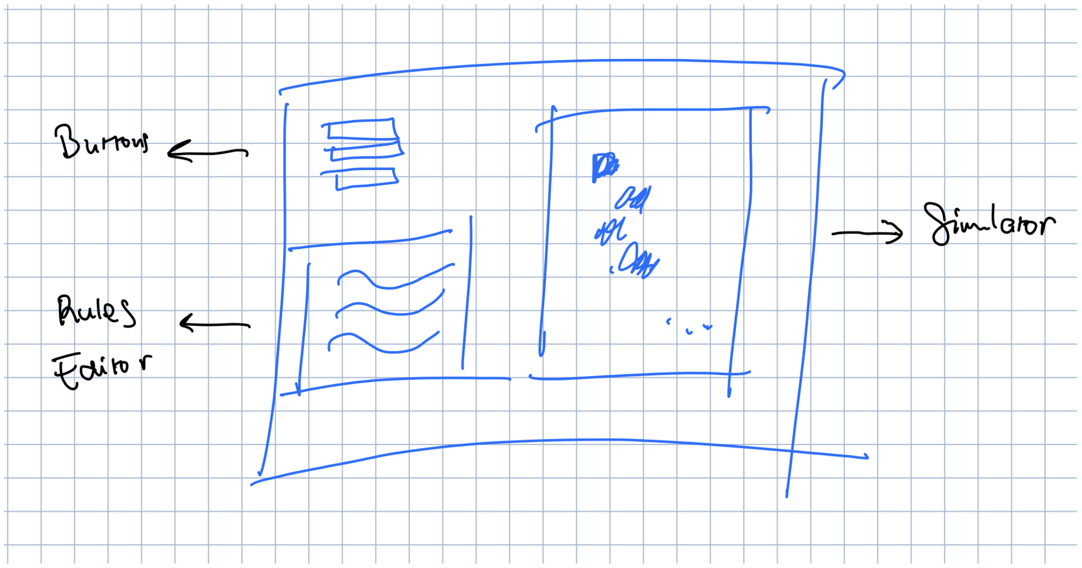
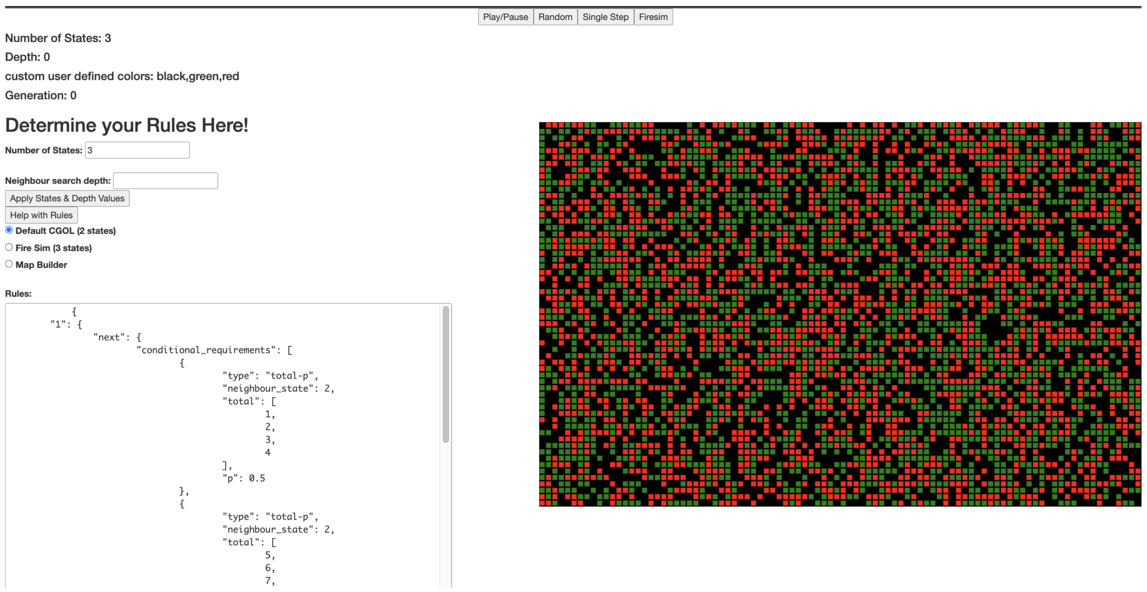


Figure 3.4: Early GUI Sketch

The rules editor and buttons are to be positioned on the left side of the screen, whereas the simulator is placed on the right hand side. This design is inspired by a variety of IDEs, where buttons are typically placed on top of the editor area, and the output (in this case the simulator) is positioned on the right (oftentimes bottom). The choice of inspiration from an IDE is that the intended audience (for evaluation) of this project is Computer Science Students. There is a higher chance that the audience has interfaced with an IDE before, and my hypothesis is that this design will increase usability. This IDE-inspired design hopes to be ergonomic by adopting the principle of humans reading from left to right, up to down.

The final GUI is shown in the following:



Number of States: 3

Depth: 0

custom user defined colors: black,green,red

Generation: 0

Determine your Rules Here!

Number of States: 3

Neighbour search depth:

[✓ Default CGOL (2 states)]
Fire Sim (3 states)
Map Builder

Rules:

```
        {
    "1": {
        "next": {
            "conditional_requirements": [
                {
                    "type": "total-p",
                    "neighbour_state": 2,
                    "total": [
                        1,
                        2,
                        3,
                        4
                    ],
                    "p": 0.5
                },
                {
                    "type": "total-p",
                    "neighbour_state": 2,
                    "total": [
                        5,
                        6,
                        7,
```

Figure 3.5: Final GUI Design

In the end, a number of changes were added to the buttons section outlined from the early sketch above by the addition of states and color descriptions, and the number of elapsed generations. Additionally, two input fields, each for *depth* and *states* are presented. The buttons, on the other hand, have been moved to the center top of the page.

### 3.5.2 Dynamic Rules Components

The dynamic rules components are a collection of functionalities that allow the input, output and preparation of user-defined rules for validation. In the GUI, this component will form the back and front end parts of the rules section.

The editor is essentially a set of radio buttons, a customized HTML textarea component, and a 'submit' button to try and apply any user-defined rules. The rule editor (textarea) element also serves the purpose of a 'display' of the currently adopted rules of the system. The radio buttons allow for instant changes to the editor content to any system default rules. How this works is that a new textarea of the same size is rendered, based on the changes made in the radio button, as the preset rules are changed.

In the code, the dynamic rules component can be located in the `dynamic-rules-form.js` file in the `utils` folder. This component utilizes the functionality of the validator component (described in 3.5.4), and passes the rules defined by the user to the function builder (described in 3.5.5) when appropriate.

### 3.5.3 Rules Format

User-defined rules take the form of a JSON object (see section 2.3.1) with a few objects contained by its mandatory keys.

The use of a JSON format for the rules is due to its flexibility in defining nested JSON entities within its keys. Alternative formats were considered, e.g. XML, and so were different approaches (e.g. having the user write their own JavaScript functions). However, the JSON format is adopted as it's simpler and easily understandable for users due to the fact that users won't have to mark up every term [13].

Though the entire structure is validated against a JSON schema (see appendix 2.3.1), the rules have two required objects and keys: `$_meta` and `default`. Their descriptions are described in the following:

- `$_meta`: The object holding this key contains two additional objects with the following keys: `num_states` and `colors`
  - `num_states`: The `num_states` key contains a single positive integer value that identifies the number of states the rules cover.
  - `colors`: The `colors` object essentially outlines the colors associated to each state. Assume the above key `num_states` contains Integer *N*. The `colors` object should contain numeric keys from 0 to *N-1* where each key contains string representing a color. These strings can be verbal color identifiers (e.g. red, blue), hexadecimal representations, RGB triples, or numbers between 0-255 (for grayscale colors)

| Type | Additional Required Arguments |
|---|---|
| ”totalling” | neighbour_state (int)<br>total (array[int]) |
| ”probability” | p (float) |
| ”total-p” | neighbour_state (int)<br>total (array[int])<br>p (float) |
| ”expression” | lhs (object: neighbour_states (array(int)))<br>cmp (enum: [”<”, ”>”, ”=”])<br>rhs (object: neighbour_states (array(int))) |

Table 3.1: Table showing Required Arguments for Each Type of Conditional Requirements

- `default`: The `default` object defines a state transition rule. The `default` object contains the key `next`, which in turn contains the following objects that satisfy a state transition rule:

  - `conditional_requirements`: The `conditional_requirements` key contains a list of objects describing conditions to determine the next state. Each conditional requirement observes the neighbours of the cell, but does so in four different ways. The `type` key indicates the way they consider the roles of the neighbours. Depending on the value of `type`, the type of requirement calls for different parameters (keys) used to determine its satisfiability. Below are the accepted values for key `type`, how they observe neighbours, and their own required keys.

    The following table outlines the required additional arguments for each type of conditional requirements:

    1. `totalling`: The `totalling` type of a cellular automata indicates that a cell looks for a certain state type in all its neighbours. If a cell has a certain number of neighbours in a specific state, then transition conditions are met. An example use case of these rules would be in CGOL.
       Required keys:
       * `neighbour_state`: integer, indicating which state to look for in its neighbours
       * `total`: list of integers, where if the total number of neighbours with state `neighbour_state` are in the list, then state transition requirement is fulfilled.
    2. `probability`: The `probability` type requirement simply outputs *True* or *False* depending on the provided parameter `p` (for probability).
       Required keys:
       * `p`: floating point number between 0 and 1.
    3. `total-p`: The `total-p` type is a mixture of both the `totalling` and `probability` type. This means that if given a cell that has a certain number of neighbours in a certain state, it has a certain probability of transitioning to another state. An example use case of this state transition condition is of forest fire propagation
       Required keys:

        * `p`: floating point number between 0 and 1.
        * `neighbour_state`: integer, indicating which state to look for in its neighbours
        * `total`: list of integers, where if the total number of neighbours with state `neighbour_state` are in the list, then has probability `p` of transitioning.
    4. `expression`: The `expression` type of requirement is unique from the others categorical types as it looks for multiple states from its neighbours.
       Required keys:
        * `lhs`: a list of ints indicating state types
        * `cmp`: a comparator (i.e. $<$, $>$, etc.)
        * `rhs`: a list of ints indicating state types

       Essentially, this condition checks if neighbour's states defined in `lhs` are of a certain relation `cmp` compared to `rhs`
  - `satisfied` and `else` keys: These two keys contain an integer outlining which state it should transition to, depending on the satisfaction of the described conditional requirements (preceding key). If any of the conditions in `conditional_requirements` are met, then transition to state `satisfied`. Otherwise to `else`.

More objects identified by numeric keys (associating to individual states) can be defined with the exact same properties of the `default` object, ultimately defining an outer totalistic cellular automata.

The UML diagram of the rules format is shown below. Note that the following diagram's purpose is only to visualize the relationships between each fields:

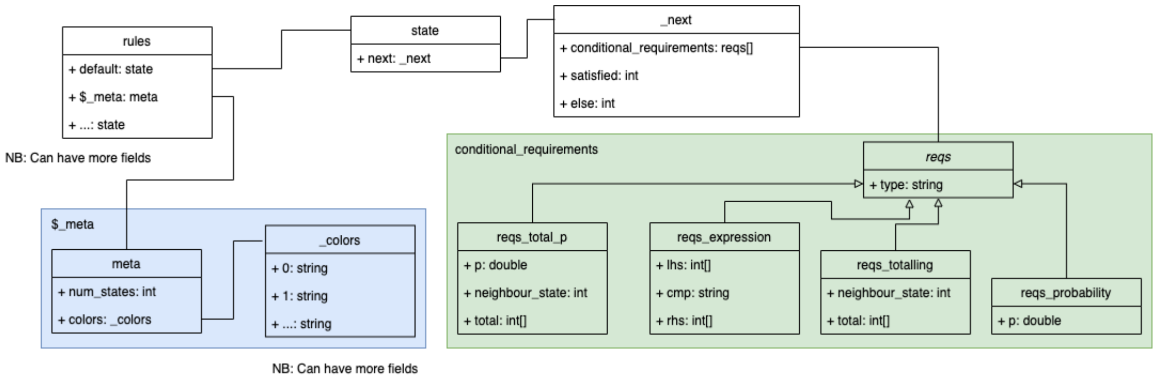


Figure 3.6: Rules Format in a UML Diagram

Overall, the reason of nesting the `conditional_requirements`, `satisfied` and `else` keys in a `next` key after the state is mainly for code readability, particularly in the function builder, e.g. compare `json_rules[0].satisfied` to `json_rules[0].next.satisfied`. I argue the latter is more readable and verbose.

#### 3.5.3.1 Examples of the Rules

In order to grasp a better understanding of the rules, some examples are necessary. Their relationships are visualized in Figure 3.6 above for reference.

The general structure of the rules take the following form:

```
{
  $_meta: { ... },
  default: { ... },
  0: { ... },
  1: { ... },
  ...
}
```

Figure 3.7: General Structure of Rules

Note that the dots between the curly braces indicate a nested JSON object. The `$_meta` object only contains two key-value pairs. Below is an example of what the object may look like, and illustrates three valid ways of defining colors in the rules:

```
$_meta: {
  num_states: 3,
  colors: {
    0: "red",
    1: 255,
    2: "#FF0000"
  }
}
```

Figure 3.8: Structure of `$_meta` object

As the previous section has stated, objects contained in the `default` key and any other numeric keys (0, 1, 2, etc.) contain the same structure. These objects are the **state transition rules**. In the following example, the `default` key contents will be shown:

```
default: {
  next: {
    conditional_requirements: [{ ... }],
    satisfied: 1,
    else: 0
  }
}
```

Figure 3.9: Structure of a basic state transition rule

The keys `satisfied` and `else` contain plain integers. The `conditional_requirements` key contains a key of objects with the following formats:

```
conditional_requirements: [
    {
      type: "totalling",
      neighbour_state: 1,
      total: [3],
    },
    {
      type: "probability",
      p: 1,
    },
    {
      type: "total-p",
      neighbour_state: 2,
      total: [1, 2, 3, 4],
      p: 0.8,
    },
    {
      type: "expression",
      lhs: {
        neighbour_states: [3, 4],
      },
      cmp: ">",
      rhs: {
        neighbour_states: [1, 2],
      },
    }
]
```

Figure 3.10: Structure of conditional requirements list and different ways of specifying requirements

There are four ways of specifying a requirement for transition. And as shown in the Diagram in Figure 3.6, each variation of the requirements inherit the `type` parameter from its parent. For the above Figure 3.10, each requirement above can be translated to the following words in plain English, presented by the following table:

| **Conditional Requirement** | **Meaning** |
|---|---|
| `type: "totalling",`<br>`neighbour_state: 1,`<br>`total: [3]` | There are a total of THREE neighbours with state ID 1 |
| `type: "probability",`<br>`p: 1` | There is a probability of 1 |
| `type: "total-p",`<br>`neighbour_state: 2,`<br>`total: [1, 2, 3, 4],`<br>`p: 0.8` | If there are 1, 2, 3, 4 neighbours with state ID 2,<br>there is a probability of 0.8 |
| `type: "expression",`<br>`lhs: {`<br>`neighbour_states: [3, 4],`<br>`},`<br>`cmp: ">",`<br>`rhs: {`<br>`neighbour_states: [1, 2]`<br>`}` | If there are more of states 3 and 4 than 1 or 2 |

Table 3.2: Table of Interpretation from Rules to its meaning in English

### 3.5.4 Validator Component

The Validator component is a standalone micro-component that compares the user-defined rules against the aforementioned JSON schema. Its functionality is called in the `dynamic-rules-form.js` in validating the textarea's contents, in the form of the function `obeysJsonSchema()`. This component is an implementation of the TV4 library described in 2.3.3.

The design choice of building the validator as the implementation of the TV4 library compared against a JSON schema is a practical choice done to save time. Developing an algorithm specific to this use case of iterating and testing the validity manually will require a great deal of time for research and implementation attempts. A general diagram on how the validator micro-component works is shown below:

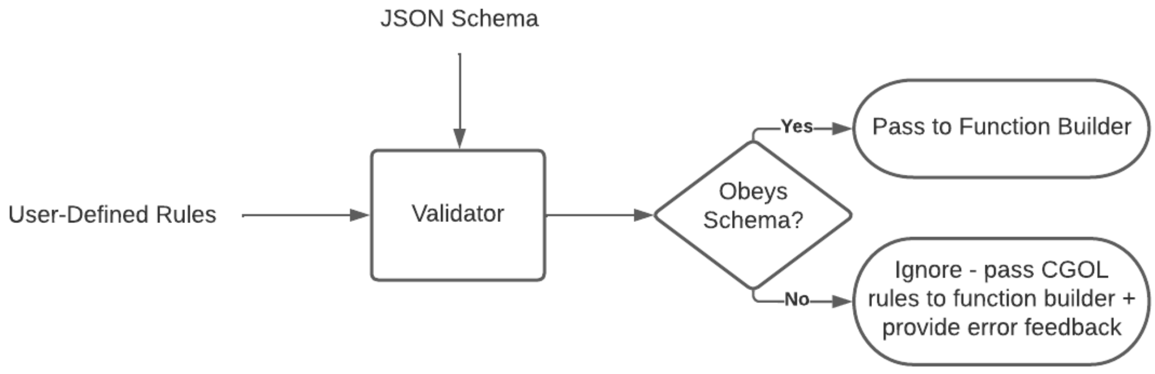


Figure 3.11: Validator Component Diagram

The validator attempts to determine the validity of rules passed to it in the system by parsing it (check to see if it's a valid JSON document), and the validity of it will be obtained by comparing against the provided JSON schema. Should the rules return invalid (or fail to parse), the validator will default to one of the system-default rules in the system, preventing the software to break.

The standalone `.js` file provided in the GitHub repository 2.3.3 had a few integration problems such that the library was unable to use my supplied JSON schema 2.3.1 despite being able to use it online. That meant the simulator's validator had to use the Node Package variant of the library.

The **Browserify** library was then used to compile the TV4 to yet another standalone `.js` file so it can provide support for browser applications [8].

### 3.5.5 Function Builder

The function builder will accept values approved by the validator. This will either be one of the system's default rules, or the valid user-defined rules provided from the rule editor. CA rules have the capability of being defined by conditional statements (i.e. if-else statements). The pipeline of the function creator component will consist of parsing the given `conditional_requirements` from the provided rules into if-statements, and then passed as an anonymous function to be used as the simulator's transition rules.

The design of each the components making up the function builder component will now be described in the following subsections. This component and other micro-components contained can be found in the `function-builder.js` file in the `utils` folder.

Overall, there are three major micro-components of the function builder: that is the condition builder, the if-statement builder, and the anonymous function builder. The architectural diagram for this module's running is provided as below:

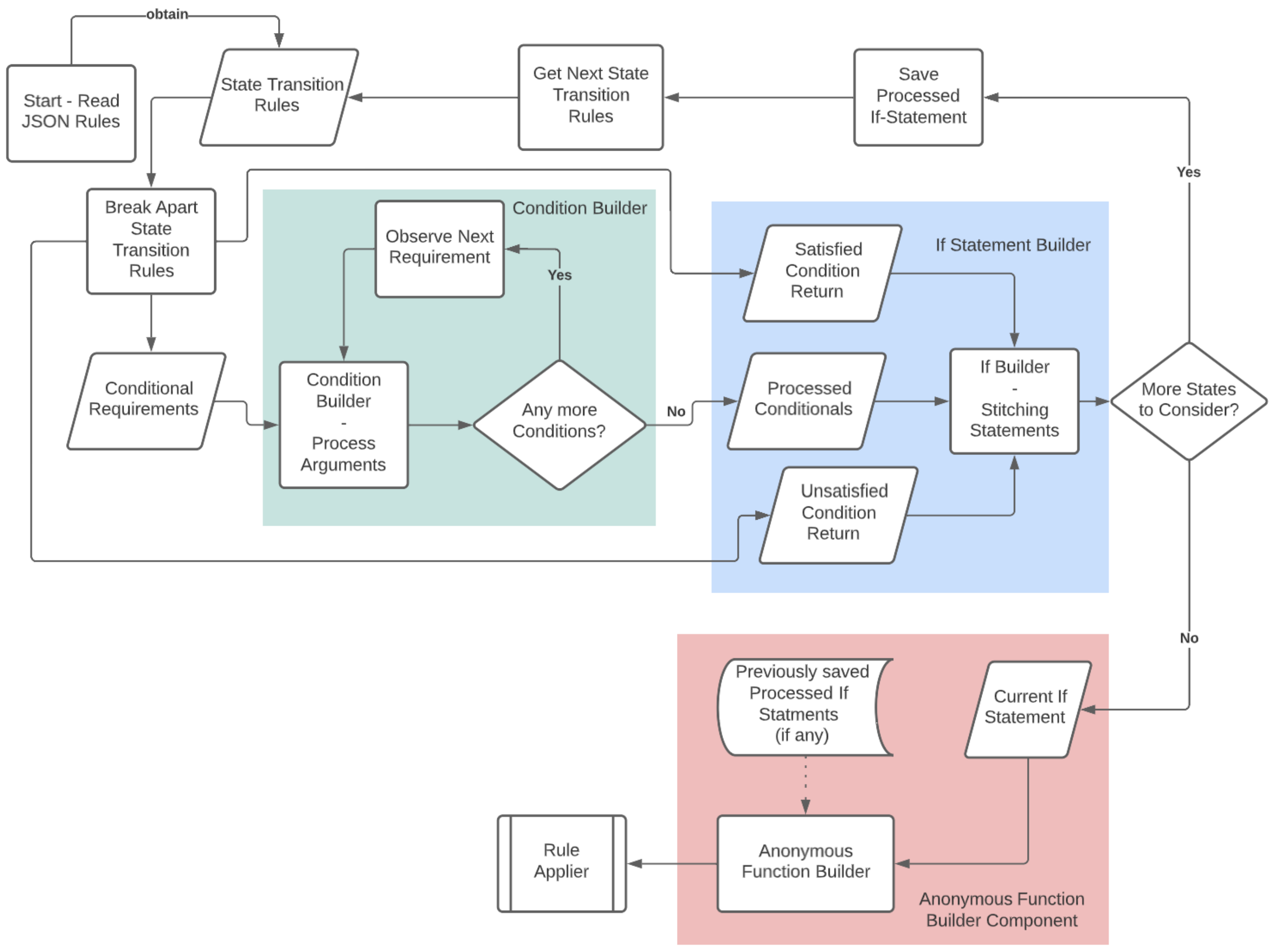

obtain
Start - Read JSON Rules
State Transition Rules
Get Next State Transition Rules
Save Processed If-Statement
Yes
Break Apart State Transition Rules
Condition Builder
Observe Next Requirement
Yes
If Statement Builder
Satisfied Condition Return
Conditional Requirements
Condition Builder - Process Arguments
Any more Conditions?
No
Processed Conditionals
If Builder - Stitching Statements
More States to Consider?
Unsatisfied Condition Return
No
Previously saved Processed If Statments (if any)
Current If Statement
Rule Applier
Anonymous Function Builder
Anonymous Function Builder Component

Figure 3.12: Function Builder Component and Micro Components

#### 3.5.5.1 Condition Builder

I define conditions in this project as the values that are placed inside an if-statement. The condition builder is a micro-component within the function builder component that processes the inner most values of the rules from the format, i.e. the `conditional_requirements` list. As described in section 3.5.3 on the rules format, the condition builder builds four types of conditions.

Referring to Table 3.3, the conditional requirements are translated into the following conditions in the following format:

| **Conditional Requirement** | **Resulting Conditional** |
|---|---|
| `type: "totalling",` `neighbour_state: 1,` `total: [3]` | `[3].includes[neighbours[1]]` |
| `type: "probability",` `p: 1` | `probability(1)` |
| `type: "total-p",` `neighbour_state: 2,` `total: [1, 2, 3, 4],` `p: 0.8` | `(probability(0.8) &&` `[1,2,3,4].includes(neighbours[2])` |
| `type: "expression",` `lhs: {` `neighbour_states: [3, 4],` `},` `cmp: ">",` `rhs: {` `neighbour_states: [1, 2]` `}` | `neigh[3] + neigh[4] > neigh[1] + neigh[2]` |

Table 3.3: Translation Table from Rules to JS Conditions Implementation

Important to note that `neigh` is the same as `neighbours` and are dictionaries. They are defined in Section 3.5.6.5 and are allocated every cell. In essence, `neighbours[1]` holds the amount of type `1` neighbours the cell in test has.

When there are multiple conditional requirements provided in the list, then the conditionals are connected via an `OR` connection. There was no specific reason in choosing to connect the conditionals with an `OR` instead of `AND`, apart from the fact that configuring the connectives could be a feature to be added another day.

#### 3.5.5.2 Probability Calculator (Micro Component)

The probability calculator micro component complements the functionality of the condition builder. Its purpose is defined above in number 2 of the `type` key value of the `conditional_requirement` object. Its evaluation, however, is called in the Actuator component. That being said, it is not shown in the diagram. Its overall usage is described in the 3.5.6 subsection below.

#### 3.5.5.3 If Builder

The If Builder is a straightforward micro-component that accepts values (conditions) passed from the condition builders. It takes two integer arguments (`satisfied_rtn` and `otherwise`) for the states to transition to, and a string argument `conditions`. This micro-component simply returns a literal string value of the built if statement.

#### 3.5.5.4 Anonymous Function Generator

Once the rules are read and the appropriate micro-components are utilized, the final part of the function builder relies on a JS anonymous function builder. The anonymous functions are then passed to the actuator, where the rules are ultimately applied in the simulator.

### 3.5.6 Actuator

The actuator will function as the simulator's back-end. Two major functionalities of the actuator is counting the Moore Neighbours of each cell in the CA and applying the rules supplied by the function builder. These functionalities are triggered by the buttons. Additionally, the actuator is the first component that is called to action in the system as it renders the entire page as the `setup()` function is executed in the `sketch.js` file. The `setup()` function operates quite trivially as it only assigns the default starting variables, and it therefore won't be discussed here.

#### 3.5.6.1 Initializer

The initializer is a micro-component for the actuator. It is the first part of the actuator that is called when a user accesses the tool, or make changes to the rules and system. More importantly, the initializer renders the front end. This component is therefore both a back-end and a front-end component. In essence, the initializer determines the following variables:

- **Number of States**: This variable is supplied to a variety of components, i.e. the color allocator micro component (see 3.5.6.3)
- **Active Rules**: The active rules are the transition rules for the simulator, after the approval from the validator component (see 3.5.4) and processed by the function builder (see 3.5.5).
- **2D Array Dimensions**: These dimensions (rows and columns) are determined in this setup. These values are passed on to the respective 2D Array Creator (see subsection below).

#### 3.5.6.2 Two-Dimensional Array Creator (Micro Component)

As the heading suggests, the actuator contains a two-dimensional array creator. This functionality is used by the initializer (see above) and by the Rule Applier micro component within this component (see 3.5.6.5). Two-dimensional arrays are used as the CA simulator two-dimensional grid, thus emphasizing its importance.

Rows and Columns of this two-dimensional array are calculated by dividing the width and height of the user's screen with the resolution.

#### 3.5.6.3 Color Allocator (Micro Component)

The color allocator is a micro component within the actuator that focuses on color allocation for each state. In essence, it returns an Array whose indexes correspond to the state number, and the contents contained at the index is the state's color. For example, consider a five-state CA with states labelled from 0-4:

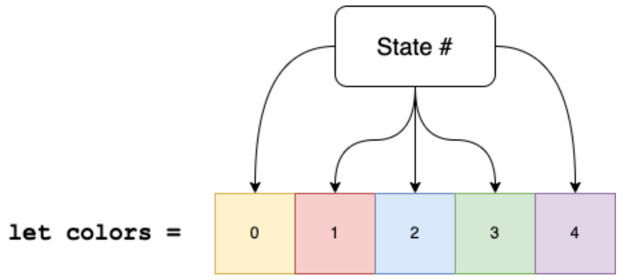


Figure 3.13: State Number Corresponding to Color

The color of State # 0 is therefore contained in `colors[0]` (yellow), State # 1 in `colors[1]`, and so on. The color of state # k is contained in `colors[k]`.

The allocator firstly checks whether the `colors` key exists in the `$_meta` object and if the number of states are equal to the number of keys contained in `colors`, and they values contained in the keys are of valid color representations. Otherwise, random colors are allocated to the states. The flowchart of this micro-component can be seen below:

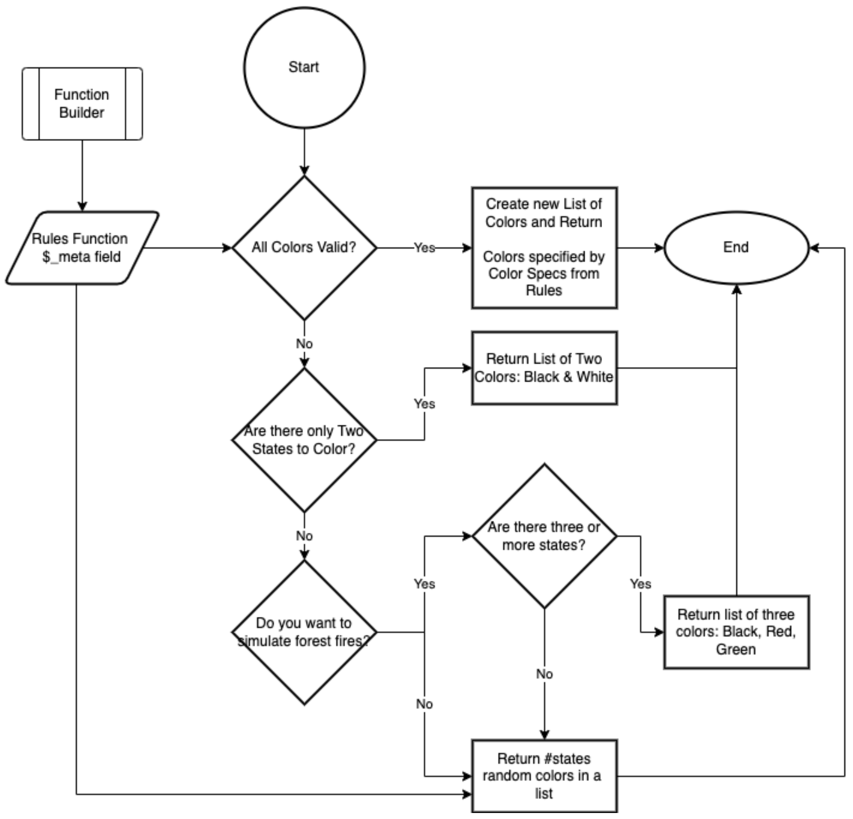

Function Builder
Start
Rules Function $_meta field
All Colors Valid?
Yes
Create new List of Colors and Return
Colors specified by Color Specs from Rules
End
No
Return List of Two Colors: Black & White
Yes
Are there only Two States to Color?
No
Are there three or more states?
Yes
Yes
Return list of three colors: Black, Red, Green
Do you want to simulate forest fires?
No
No
Return #states random colors in a list

Figure 3.14: Color Allocator Flowchart

#### 3.5.6.4 Actuator Buttons

The buttons (seen in the top part of figure 3.5) serve as the inputs (operating similarly to remote) to the actuator. Though the buttons are essentially rendered as part of the GUI, its main functionality is defined in the actuator class. The following buttons are provided, and their actions:

- Play/Pause: There is a `pause` variable of type `boolean` in the system that drives whether the simulator should execute or not. While `pause` is initially set to *True*, this button negates the value of `pause`, where there is another function `draw()` essentially runs the simulator.

- Single Step: The single step button operates the same way as the play/pause button, but what this button does in a single run is explicitly declare `pause` to *True*, calls the `draw()` function, and then resets `pause` to *False*

- Randomize: The randomize button essentially creates a new two-dimensional array according to the number of states and associated number of If colors are not provided to each state from the `$_meta` tag, then they are randomly allocated by the color allocation micro-component above in section 3.5.6.3.

The flowcharts for the buttons are shown below:

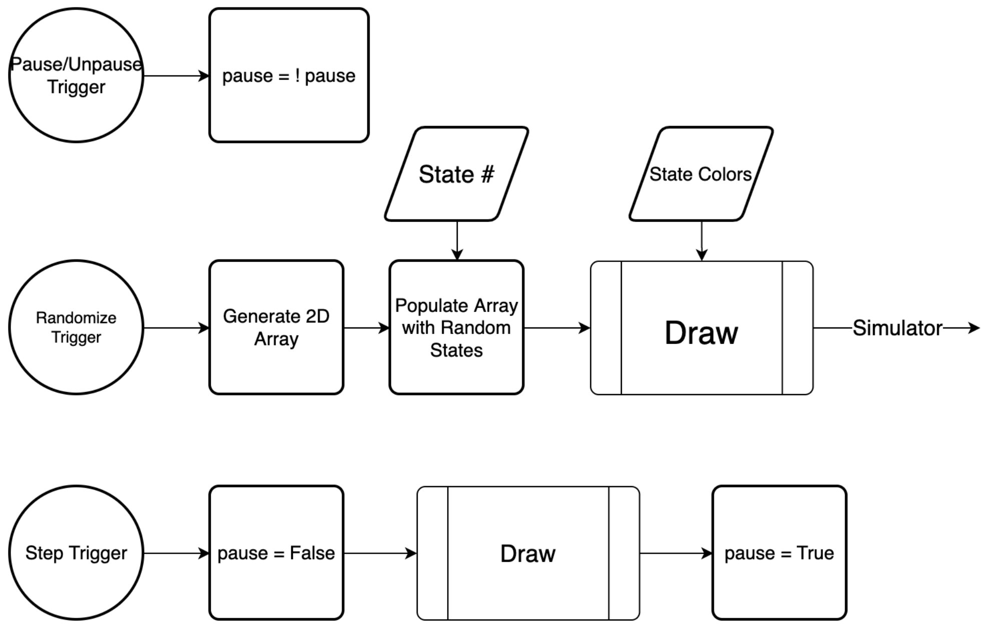


Figure 3.15: Button Actions Flowchart

### 3.5.6.5 Neighbour Counter & Rule Applier

CAs commonly consider the neighbour numbers and states before deciding what to do next. That being said, neighbour numbers and their states are typically calculated in one single function, that is when the rules are applied. However, integrating both functionalities into a single micro-component introduces a higher level of cognitive complexity in the algorithms, making it potentially harder to manage, and maybe even add features.

For that reason, these micro-components are separately defined (in different functions) for modularity. However, in the running process of the rule applier micro-component, it calls the function of the neighbour counter. Following is the flowchart of the Rule Applier:

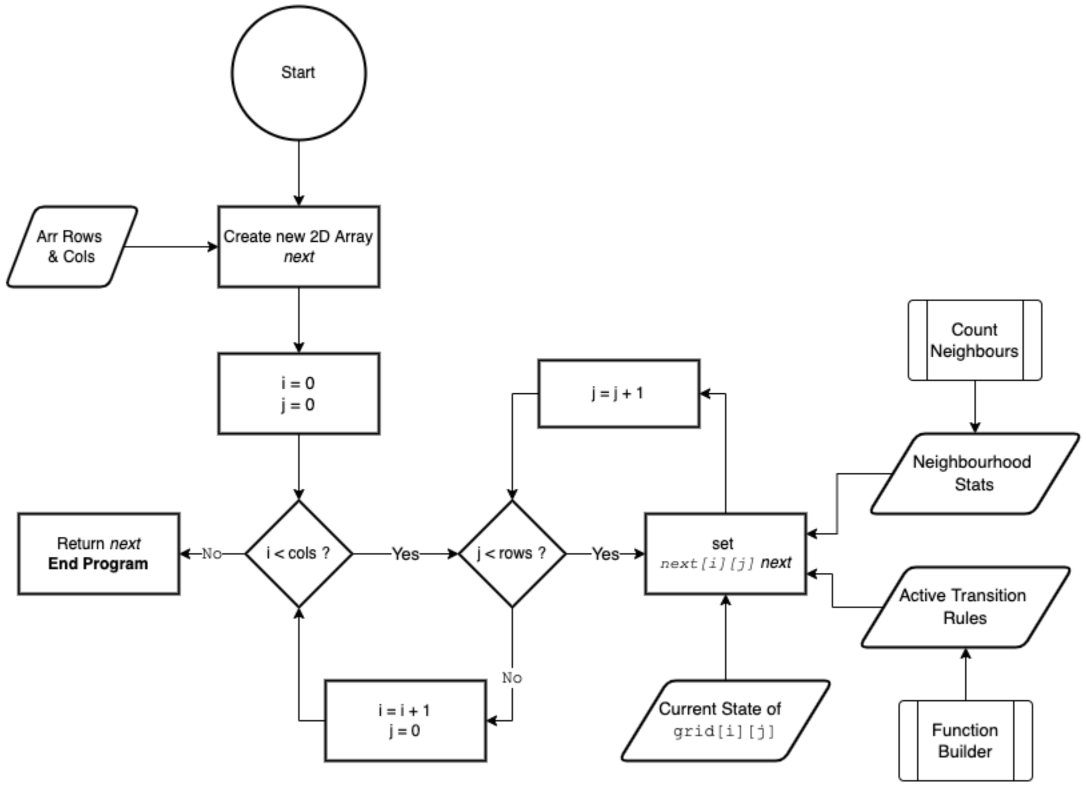


Figure 3.16: Rule Applier Flowchart

The rule applier essentially creates a new two dimensional array of cells with the same dimensions as the current grid, iterates each cell from top left to bottom right by using a nested for-loop, assigning new states to the cells at the new array (with input from neighbour count and cell's current state).

The neighbour counter operates as specified by the following chart:

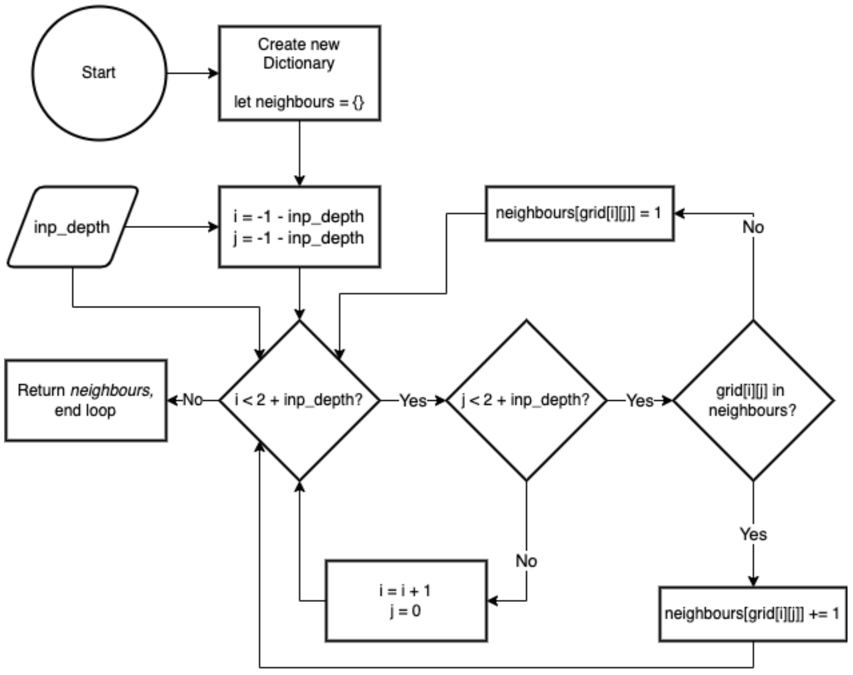


Figure 3.17: Neighbour Counter Flowchart

The design choice of implementing an infinite two-dimensional grid (instead of a finite grid in the provided frame) was made due to the fact that CAs are typically based on an infinite grid, particularly CGOL [18]. The way this was implemented is reflected in the rule applier code, where the cells wrap around each other (e.g. bottom right cell's neighbour includes top left cell, and vice versa).

### 3.5.7 Sketcher

The sketcher (or simulator) visualizes the current CA grid and its colors. This component has mostly stayed the same in the context of its GUI appearance, and is purely front end. It accepts two dimensional arrays containing the new state of the CA. This component utilizes the P5JS library, described in 2.3.2. A small part of the sketching process occurs in the highlighted area in the flowchart below:

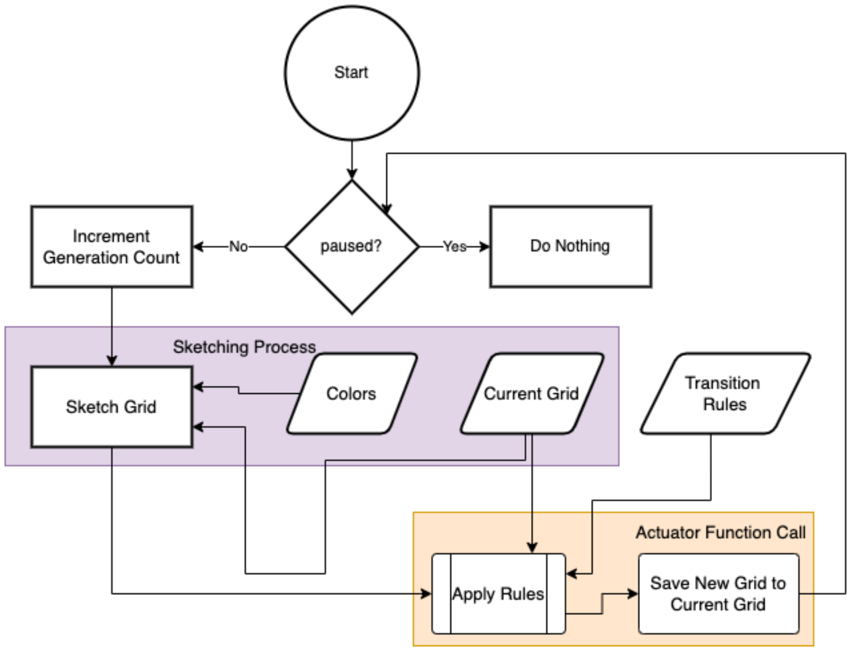


Figure 3.18: Flowchart of Sketcher

The `sketch()` function attempts to visualize the CAs in the simulator. Additionally, this module calls the functionality of the actuator (indicated as well in the flowchart) to keep its process running.

### 3.5.8 Help Page

Finally, there is a basic help page that is essentially a static `.html` page. It describes the basics of CAs. Additionally, this page outlines a basic tutorial on how to use the tool, and more importantly learn the proper syntax of the rules to input to the system through the editor. As this component is a simple HTML page, no component diagram is specified, nor will this be represented in the general architectural diagram.

The help page can be accessed through a button above the rules editor.

## 3.6 Dependency Graph

This graph shows the dependencies needed by each of the written components in order to properly function. Arrows from component **A** pointing towards another component **B** signify that ”component **A** depends on component **B**”. Additionally, the external libraries are highlighted in the diagram.

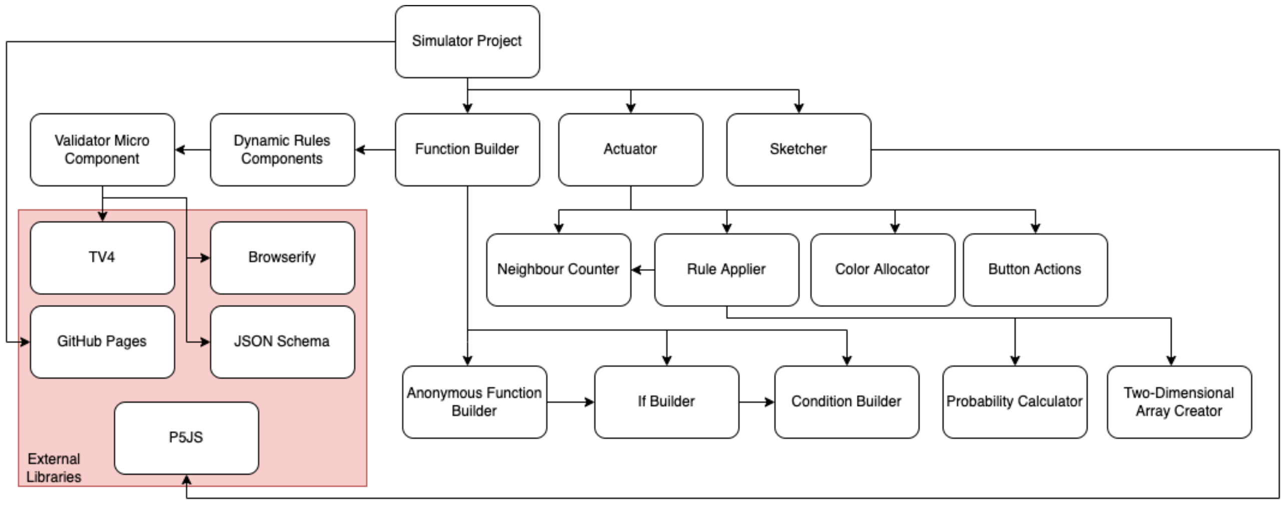


Figure 3.19: Dependency Diagram

Note that some component names were not listed such as the GUI (see 3.5.1) and the Initializer (see 3.5.6.1) as their functionalities are purely front end. Inherently, they depend on each and every one of the components and form as part of the simulator project itself.

# Chapter 4

# Testing & Evaluation

A known limitation of the system is that there is no user-friendly way of defining the dimensions of the CA simulator. That being said, to test the rules' accuracies and how well the simulation scales, some changes need to be made to the code base at the lower level.

The scalability of the simulator is tested as the simulator size is dependent on the size of the computer screen. No performance enhancements will be made, as only the display size will be changed.

## 4.1 Simulation Scaling Test

Before going to the details on the procedure of this test, it is important to understand how the program determines the dimensions of the two-dimensional array for the simulator. This is discussed in 3.5.6.2. We measure how well the simulation scales by measuring the amount of time taken between each generation's transition, i.e. how long it takes for the simulator to apply the transition rules to all the cells, and the average time taken to do so. The average time taken to transition between ten generations is the dependent variable. The independent variable will be a coefficient introduced to the resolution, as the size of the simulator is determined in the `setup()` function where the rows and column count are divided by the resolution. Only the code from the `sketch.js` file are changed to support logging. The following changes are introduced to the system to evaluate its scalability:

1. A new Array `times_10` will be made to store the times taken for transition between 10 generations.

2. A new floating point number, `coeff` to be multiplied to the resolution in the `setup` function.

3. The `draw()` function will automatically stop after ten iterations, then log the times taken.

The rules operated in this test will be the CGOL and Forest Fire Simulator Rules.

It is important to note that the number of rows and columns for the grid are directly proportional to the resolution coefficient, as shown in the table below and the graph that follows:

| Resolution Coefficient | Rows | Cols |
|---|---|---|
| 1 | 60 | 96 |
| 1.5 | 90 | 144 |
| 2 | 120 | 192 |
| 2.5 | 150 | 240 |
| 3 | 180 | 288 |
| 3.5 | 210 | 336 |
| 4 | 240 | 384 |
| 4.5 | 270 | 432 |

Table 4.1: Table of Resolution Coefficient to Rows and Columns

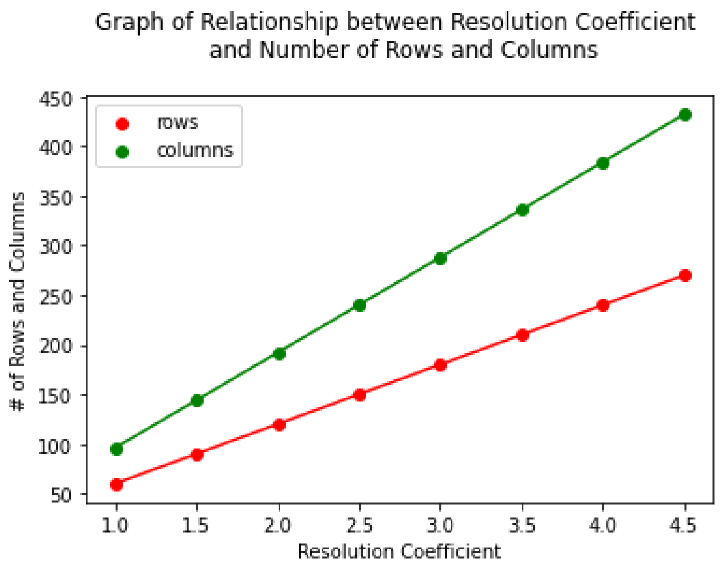


Figure 4.1: Graph showing of Resolution Coefficient vs. no. of Rows and Columns

That being said, this section will observe the relationship between the coefficient (inherently the number of rows and columns) and the time it takes to apply the transition rules:

| Resolution Coefficient | Generation Transition Times (ms) | | | | | | | | | | Average (ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 61 | 40 | 39 | 39 | 39 | 30 | 31 | 31 | 31 | 32 | 37.3 |
| 1.5 | 105 | 116 | 109 | 92 | 90 | 91 | 92 | 95 | 95 | 95 | 98 |
| 2 | 548 | 564 | 559 | 498 | 508 | 506 | 507 | 509 | 465 | 493 | 517.7 |
| 2.5 | 645 | 609 | 626 | 552 | 595 | 586 | 623 | 552 | 590 | 561 | 593.9 |
| 3 | 1434 | 1307 | 1254 | 1209 | 1213 | 1156 | 1169 | 1174 | 1180 | 1146 | 1224.2 |
| 3.5 | 1876 | 1759 | 1790 | 1923 | 1822 | 1748 | 1703 | 1786 | 1686 | 1663 | 1775.6 |
| 4 | 2345 | 1802 | 1855 | 1891 | 1810 | 1794 | 1772 | 1786 | 1820 | 1954 | 1882.9 |
| 4.5 | 3070 | 2946 | 3059 | 2835 | 2805 | 2738 | 2792 | 2824 | 2735 | 2644 | 2844.8 |

Table 4.2: Table of Resolution Coefficient and Generation Transition Times for CGOL Rules in ms

From the table above, it can be seen that there is a positive correlation between the resolution coefficient and the average time taken for the transition to occur to the entirety of the CA grid. The same experiment can be done for the forest fire simulator rules. The results are provided in the table below:

| Resolution Coefficient | Generation Transition Times (ms) | | | | | | | | | | Average (ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 45 | 22 | 23 | 26 | 27 | 28 | 24 | 23 | 40 | 24 | 28.2 |
| 1.5 | 93 | 98 | 76 | 81 | 80 | 72 | 80 | 92 | 76 | 82 | 83.0 |
| 2 | 524 | 466 | 564 | 510 | 453 | 508 | 495 | 464 | 446 | 478 | 490.8 |
| 2.5 | 592 | 613 | 523 | 559 | 551 | 588 | 547 | 545 | 592 | 566 | 567.6 |
| 3 | 1250 | 1185 | 1266 | 1149 | 1158 | 1159 | 1117 | 1134 | 1108 | 1084 | 1161.0 |
| 3.5 | 1718 | 1728 | 1795 | 1724 | 1709 | 1640 | 1647 | 1586 | 1678 | 1591 | 1681.6 |
| 4 | 2029 | 2188 | 2294 | 1850 | 1901 | 1799 | 1896 | 1886 | 1800 | 1881 | 1952.4 |
| 4.5 | 2551 | 2658 | 2552 | 2547 | 2509 | 2511 | 2520 | 2545 | 2575 | 2748 | 2571.6 |

Table 4.3: Table of Resolution Coefficient and Generation Transition Times for Forest Fire Simulator Rules in ms

The curve below visualizes the relationship between the resolution coefficient and the average transition time from one generation to another.

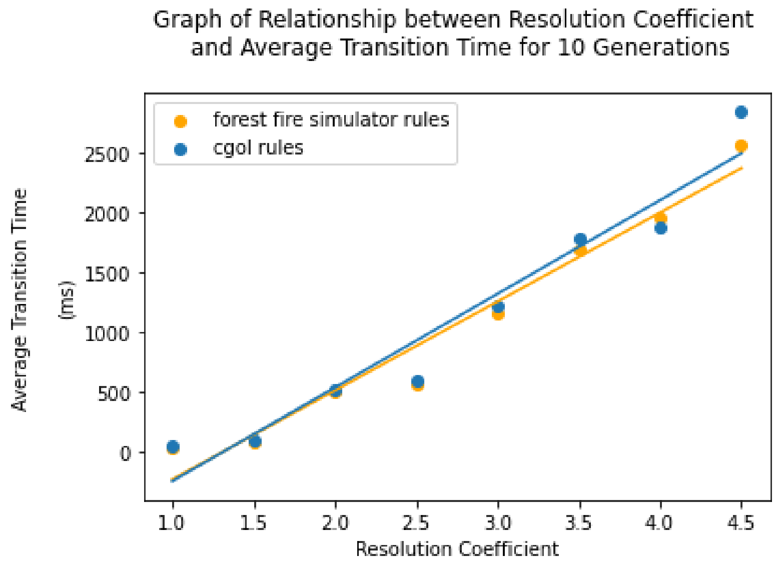


Figure 4.2: Graph showing of Resolution Coefficient vs. Avg. Transition Times

Based on the straight line and minimal signs of deviation, there is evidently a direct proportional relationship between the resolution coefficient and the average transition times. This is because the growth in number of rows and columns is directly proportional to the growth of the resolution coefficient.

## 4.2 User Acceptance Testing

In order to obtain feedback from users about this system's perceived usefulness, a qualitative interview was conducted. Two one-on-one interviews were conducted at different sittings. Both interview participants were computer science students at the University of Manchester. Each participant has also indicated familiarity with JSON and CAs (only CGOL).

No personally identifiable data is collected during the interview. Both participants have indicated their consent in participating by signing a form, and are aware that participation is entirely voluntary and could withdraw from the interview at any point.

### 4.2.1 Setup and Methodology

A document was created to act as a guide for the interviewees. This document was to be shown side-by-side the software in hand (see Appendix A). Additionally, the Think Aloud method was adopted throughout the interview process. This means that the participants continuously spoke aloud any words that appear in their mind as they complete the tasks, and even up until concluding the interview [9]. I as the researcher was present beside the participant(s) to observe each of their thought processes, take notes, and provide support if required.

Before conducting the interviews, one line of the code was changed. This change resulted in the radio button that populated the rules editor with the Map Generation CA (bottom left) to be hidden. This will prevent the participant in enabling the Map Generation CA rules, as the interview will involve the process of replicating the aforementioned rules (final part of Appendix A). The Appendix also serves as the questions asked throughout the interview.

### 4.2.2 Understanding the User Interface

This part of the interview involved identifying the rule editor, simulator, and the buttons (Appendix A). None of the interviewees experienced difficulties in identifying the visible components of the software. Both participants completed this part relatively quickly.

However, both participants showed a slight difficulty in detecting the buttons. A suggestion that was given would be to increase the sizes of the buttons at the very top, or move them downwards to increase visibility.

### 4.2.3 Understanding the Rules

In this part, at the same sitting, three sets of tasks relating to the JSON-format rules were given to the participants. The aims of each tasks are given below, in addition to their results. See the more detailed step-by-step instructions in the Appendix A.

#### 4.2.3.1 Color Changing

This section required the participants to change the color of the ’dead’ state from CGOL, from black to red.

Both participants found this step to be relatively trivial as both simply had to look for the value in the JSON rules that represents a color (in this case-black), and simply replaced it to the required ’red’ color.

#### 4.2.3.2 Changing the Totalistic Transition Rules

The participants were required in this section to make changes to the rules of CGOL by starting with the default settings. In general, both participants showed some signs of difficulty in understanding how the rules worked, despite having been exposed to JSON and CAs.

The first participant took less time to complete this task when compared to the second participant, though admitted it was considerably harder than the previous task as they (first participant) had to spend time understanding how the rules worked first. The second participant commented that ”getting past the conditionals was the hardest part”. They (second participant) however concluded that after understanding how the conditionals worked, working around it was quite simple.

It is important to note that both participants at this stage have still not gone through the rules guide. Both participants tried to understand how the rules worked by comparing the English translation of the CA’s rules to system-defined format.

#### 4.2.3.3 Changing the Probabilistic Transition Rules

In this part, participants were tasked to interface with the forest fire propagation rules and simulator. Upon starting this task, both participants had to follow a specific set of steps to set up the system for the forest fire simulation environment. This setup process took slightly longer than I initially expected it to be. I ended up providing hints and explicit guidance on the procedure to set up the rules before moving on to the main bit of the forest fire propagation CA.

Both participants showed intrigue in the forest fire propagation rules when it was running. The first task in this part (aside from setting up) was to infer what the rules in the editor meant. Both participants understood the total-probabilistic rules (`total-p`) a bit faster than previous. Additionally, both participants have at this stage started consulting the documentation. This resulted in the second task being completed much faster than expected. However, the final task was considerably harder for both.

#### 4.2.3.4 Replicating the 2D Map Generation Rules based on Provided Specifications

This final part of the interview required both participants to replicate the Game Development Map Generation rules (specified in Appendix B.3) from the CGOL rules (specified in Appendix B.1). In summary, both participants successfully completed this task and replicated the 2D Map Generation Rules as specified.

Starting from the CGOL default rules, both participants took the same approach of deleting the `0` key entirely, replacing the colors, and resorted only to using the `default` key. Both understood the specifications, but were slightly baffled at how the rules implemented the specifications given. More on this in the following section.

### 4.2.4 Analysis and Participant Suggestions

Firstly, there was a general confusion on whether which state number corresponded to what number. In the case of CGOL, the states of **Dead** and **Alive** were assigned state numbers **0** and **1** respectively. This was not made clear in the beginning, and the confusion was present in the preceding tasks. When modifying the forest fire rules, the first participant looked at the `colors` key in the `$_meta` object and inferred the relationship between state names and numbers. A similar observation was done by the same participant when interacting with the forest fire CA rules - correlating the colors red, green, and black to fire, vulnerable vegetation, and burnt/invulnerable entities. Both participants, however suggested adding a descriptor field in the rules, to make it easier to identify which numbers corresponded to which states.

Both participants showed signs of confusion when preparing the system to simulate the forest fire CA rules (subsection 4.2.3.3). This confusion was followed by a suggestion from both participants, to remove the 'number of states' text area and integrate that information to the dynamic rules `$_meta` tag.

As the issues mentioned in the two paragraphs previous are relatively straightforward to implement, I've mentioned how this could be implemented in the **Potential Extension Work** section (5.4) in the next chapter.

When interfacing with both the totalistic and total-probabilistic type of rules, the `total` argument caused confusion for both participants. As a result, I pointed both participants to the help page for the rules. They both noted as well that individually listing integers when it comes to describing the totalistic rules were somewhat impractical. An example of this would be how the 2D Map Generation rules were specified (see in section 2.2.4) and how they were translated as rules supplied to the system. When the specifications state "less than or equal to four neighbours", the rules write `[0, 1, 2, 3, 4]` and checks if the number of neighbours is in that array.

Additional suggestions that were made included syntax highlighting and auto-indentation. Furthermore, both participants quoted that learning and interacting with JSON on-the-spot was a rather daunting experience.

# Chapter 5

# Reflection & Conclusion

## 5.1 Planning and Management of Work

By using GitHub Issues and Pull Requests, planning and management of work was a relative success. Work on the project was done consistently during the allocated time frame, and when no changes were made to the repository, that meant research was still being done.

## 5.2 Project Achievements

I believe I've successfully achieved everything I aimed for in the beginning of the project. All the features, both functional and non-functional were met. Additionally, my objectives outlined in the first chapter were generally achieved. I gained a deeper understanding of CAs in general, in addition to honing my relatively forgotten skills in HTML JS. More importantly, the system was built and it scales relatively well in a linear fashion, albeit its slow times.

Through the interviews conducted, I believe I've received mostly positive feedback from the interviewees regarding my project, although the approach could have been much better. I feel a sense of achievement when the participants naturally understood the rules and how I defined them, despite the little moments of confusion that was displayed initially and oftentimes when new concepts were introduced. I am grateful to have received constructive feedback from both my supervisor, my interview participants, and everyone else I consulted along the way.

## 5.3 What Went Wrong

This section delves on to the changes towards the development process. Changes that I would make if I were to repeat the project would be:

- **Integration of Trello Boards and More Extensive Notetaking**: Planning and Management of work could have been greatly improved by Trello Boards or GitHub projects to better visualize which stage each component was at. This would have been a more streamlined approach when compared to purely using GitHub Issues, not to mention this process was a lot faster than individually creating issues. A great deal of notes were also scattered, I lacked a dedicated notebook for project note taking, and information I received through online resources only sat in my head. By having a dedicated project notebook, the planning process could have probably been done in half the time required.

- **Change in Approach for Rule Format Development**: Though the management of work was relatively flawless as were the achievements, one thing I would definitely change during the development stage was the approach taken in doing the project. My approach in developing the rules format was bottom-up. Upon starting the project, I studied extensively the rules of CGOL, and implemented features based on that functionality. As a result, the rules format were somewhat rigid. The state transition rules only gave two choices of states to transition into. If I could start over entirely, I would have taken more time to research CAs and think of a way to provide more state transition choices.

- **Consistently ask for Feedback from the Audience**: Also dealing with the rules at another angle, if I were to gather constant feedback throughout the development, then the rule format would have probably been different, when coupled with Another most The entire development process was somewhat closed-loop. The End-users (target audience of Computer Science University Students) were not at all involved in the development stage. The lack of consistent feedback from my intended end-users then resulted in a rules format that is relatively lacklustre.

- **Give more thought on Testing - Create a Test Suite**: There was also a lack of standardized testing procedure during the development of the project. The current workflow only switches between GitHub version commit hashes to identify any breaking changes if present in the main development branch. There was no unit testing, integration testing, or regression testing. Therefore, another aspect that I would change if I were to start over would be to spend some time in the beginning to design and develop a testing suite for the project. Time invested in developing a test suite will surely save the amount of time needed in creating and integrating new features.

Additionally, under mitigating circumstances, this project's deadline was extended until the end of the second semester exam period.

## 5.4 Potential Extension Work

This section outlines potential features that could be implemented in the final product, but unfortunately didn't make the final submission release. Assuming the project was started/restarted correctly with the lessons learnt in the previous section (5.3) up until this point in the final code, the following extra features are what I would like to implement in my project.

### 5.4.1 Improved Rules Documentation Page

The documentation (rules help page) was truthfully created as an afterthought. There are discrepancies in the syntax, and oftentimes they are not descriptive enough. Although both participants in the chapter previous has outlined the usefulness of the rules format, much can be done to improve the comprehension and quality of information delivered by the help page. Especially after potentially improving the rules format to add more support.

### 5.4.2 Adding a Description Field for CA Rules States

As outlined in Section 4.2.4, adding a description field to indicate what type of state is related to the number would greatly reduce confusion and complexity. The additional field, call it `state_name`, can be placed at the same level as the `next` key in every state object. In the example of CGOL rules at state 0 (Dead state):

```
"0": {
    "state_name": "dead",
    "next": {
        "conditional_requirements": [
            {
                "type": "totalling",
                "neighbour_state": 1,
                "total": [
                    3
                ]
            }
        ],
        "satisfied": 1,
        "else": 0
    }
},
```

Figure 5.1: CGOL State Transition Rule when state is 0 (dead), with description field

Notice the new key right before `next`.

Additionally, some changes to the JSON Schema will need to be made as well. In Appendix C, the following line will need to be updated in the `$defs/state/properties`:

```
state_name: {
    type: "string"
}
```

Figure 5.2: Line to be added to `state` property in the JSON Schema

### 5.4.3 Integrating Number of States Count to the Rules Only

This issue was also mentioned in Section 4.2.4. The current implementation requests the user to separately input the number of states in both the JSON rules, and the text field above it. The reason behind this implementation is that the `$_meta` field containing state number and colors were an afterthought. This discrepancy was shed to light after the code freeze.

Changes to the implementation would be to simply erase the textarea input field, and assign the value from `num_states` to the variable that was supposed to contain the value obtained from the previously removed input field.

### 5.4.4 Validator Enhancement: Check Parse Errors/Failures

In most use cases, the validator provides the feedback to the user-specified rules. However, there have been cases during arbitrary utilization involving the rules that the simulator simply stopped working due to an error in the validator. The current implementation simply parses values from the text editor to the as a JSON object value from the textarea to the validator, with a general assumption the syntax is correct. However, that is oftentimes rarely the case.

In the text area, the opening and closing curly braces (first and final lines) can be removed to provide the users a more aesthetic feel and prevent accidental deletion from them. These two characters can then be pre-pended and post-pended before being parsed as a rule. Additionally, a simple try-catch statement can be integrated at the appropriate component (i.e. the Validator component, see 3.5.4) to check its parse quality.

### 5.4.5 Reversible Cellular Automata

The current implementation of the simulator only covers totalistic and probabilistic CAs. Reversible CAs, on the other hand, state that for every configuration of the grid, there is a unique predecessor to its configuration [12]. Reversible CAs are often used in physics, particularly in gas and fluid dynamics [34]. A special rule of the CA called *Critters* developed by Tofolli [34] is reversible. Implementing this feature will enable more universality as it allows additional simulation styles.

### 5.4.6 Undo Command

Taking inspiration from reversible CAs from the subsection previous, and ”undo” button can be implemented to clarify if the user-defined rules behave as specified. This functionality allows testing the randomness of the implemented transition rules, or simply debugging a rule when the user wants to keep the initial state (or previous state) of the entire grid.

In terms of implementation, a separate finite list (Array) containing the current CA’s grid (as a 2D array) and also the grid from a few steps before can be used as a temporary memory space.

Alternatively, a stack can be used in place of the Array for the same purpose. Undoing the applying of the transition rule to the grid (i.e. going back one generation) can mean a pop of the stack, and assigning that value to it. ”Undo” commands commonly use the stack [17].

### 5.4.7 Support for Defining CA Grid Dimensions

The system currently generates a grid based off the user’s screen size and resolution. This therefore doesn’t allow for larger scale simulations. Two basic text fields accepting numbers can be used as user’s input to specify its row and column numbers, then be utilized in the simulation and models.

Cell sizes should also have the capability of being automatically scaled down, in order to fit more cells in the provided square. This functionality can be added to the `sketcher` bit of my written library, scaling down the square’s (cell’s) size by the grid dimensions and provided simulator frame size.

### 5.4.8 Support for User-Defined Initial States and Clickable Grid

For some CAs that are used for prediction (e.g. population dynamics, ecological growth), initial states may be crucial to test the robustness of the defined CA rules, after a few generations. For example, the implementation of CAs for population dynamics outlined by Mavroudi [23] rely on a certain city-style grid.

Activating these presets can be done by the click of a button, similar to how the current implementation of `Firesim` works. The only difference would be instead of calling a function to randomly generate states of the cells, the button loads a predetermined two-dimensional array with the appropriate number of states.

### 5.4.9 Support for Variable Grid Density

The `random` button in the program purely allocates states randomly. This means that each state in the *state space* (see section 2.1) all have an equal chance of being considered in the randomization process. Minor changes to the actuator class can be made in order to provide support for grid density.

By using the predefined probability calculator function in the Function Builder Component (see 3.5.5.2), instead of randomly allocating states to the cells, some states may have a larger probability of being assigned to a cell. Input can be taken by creating a new textarea where the value should be a floating point number between 0 and 1.

### 5.4.10 Support for more Complex Probability Rules

The current implementation of rules only allows for two types of probability, i.e. `probability` and `total-p`. The latter when implemented on the forest fire propagation simulation briefly touched upon the idea of a probability distribution. This means that the more of a cell's neighbours is on fire, the more likely the cell in test will burn.

A method of specifying a custom probability function can be implemented by allowing the user to write probability functions in JavaScript, then pass that written function to the anonymous function builder. However, this might add an additional layer of complexity towards the users. Therefore an additional user-defined rules format specifically for writing probabilities may need to be developed again.

### 5.4.11 Support for Different Neighbourhood Types

In addition to the Moore-type neighbourhood defined in 2.1.3, another commonly used neighbourhood type is the Von-Neumann Neighbourhood, shown below:

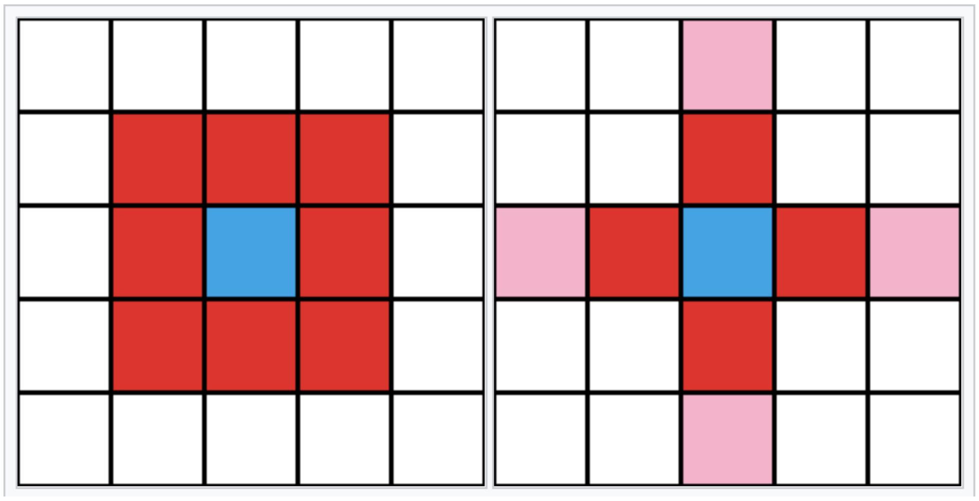


Figure 5.3: Moore Neighbourhood (left) and Von-Neumann Neighbourhood (right) [34]

In contrast to the Moore Neigbourhood, the Von Neumann Neighbourhood only considers the adjacently placed cells of a cell in the center. Like the Moore Neighbourhood, the Von-Neumann neighbourhood may also consider the search depth of the neighbours, also defined as the manhattan distance from the center cell. On the right-hand side of the Figure 5.3 above, for the Von-Neumann neighbourhood, the red cells are Von-Neumann neighbours of manhattan distance $r = 1$, whereas the pink ones are of $r = 2$.

In terms of changes towards the implementation, the JSON Schema fed to the validator will need to add a required neighbour-type field in the `conditional_requirements` property, particularly in the `reqs_list` to validate the neighbourhood type. In the actuator component, another function (e.g. `count_vn_neighbours()`) can be written beside the default `countNeighbours()` (change to perhaps `count_m_neighbours()`). Once these functionalities are implemented, this will allow the function builder component to return a new anonymous function implementing the functionality of the Von-Neumann Neighbourhood.

Once implemented, the Von-Neumann Neighbourhood may serve as a basic building block of defining the notion of 4-connected pixels in the field of computer graphics [37], adding more levels of universality to the simulator.

## 5.5 Conclusion and Personal Achievements

Considering the amount of time that was spent researching and developing for the project, I learned a great deal throughout the entire process. Revisiting HTML and JS was a rewarding experience for me as it has been quite a while since I've developed using those languages. I gained invaluable experience as well, particularly in conducting qualitative interviews to evaluate. Overall, this learning experience was extremely rewarding for me as I got to push my boundaries and nurture my resourcefulness. So many things that I didn't expect to learn by studying Computer Science. This experience and lessons learnt are invaluable, and I will certainly keep pushing my boundaries and abilities as I did from this project's start to finish.

# Bibliography

[1] A. Alexandridis, L. Russo, D. Vakalis, G. V. Bafas, and C. I. Siettos. Wildland fire spread modelling using cellular automata: Evolution in large-scale spatially heterogeneous environments under fire suppression tactics. *International Journal of Wildland Fire*, 20(5):633, Oct 2011.

[2] Daniel Ashlock. Evolvable fashion-based cellular automata for generating cavern systems. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 306–313. IEEE, 2015.

[3] J Banks, J Carson, BL Nelson, and D Nicol. Discrete-event system simulation"prentice hall. *Inc. EnglewoodCliffs, NewJersey*, 198(4), 1996.

[4] Lindsay Bassett. *Introduction to JavaScript object notation: a to-the-point guide to JSON.* " O'Reilly Media, Inc.", 2015.

[5] Michael Batty. Cellular automata and urban form: a primer. *Journal of the American planning association*, 63(2):266–274, 1997.

[6] Michael Batty. Urban evolution on the desktop: simulation with the use of extended cellular automata. *Environment and planning A*, 30(11):1943–1967, 1998.

[7] Michael Batty, Yichun Xie, and Zhanli Sun. Modeling urban dynamics through gis-based cellular automata. *Computers, environment and urban systems*, 23(3):205–233, 1999.

[8] browserify. browserify. 2010.

[9] Elizabeth Charters. The use of think-aloud methods in qualitative research an introduction to think-aloud methods. *Brock Education Journal*, 12(2), 2003.

[10] J. G. Freire and C. C. DaCamara. Using cellular automata to simulate wildfire propagation and to assist in fire management. *Natural Hazards and Earth System Sciences*, 19(1):169–179, 2019.

[11] Martin Gardner. The fantastic combinations of john conway's new solitaire game" life". 1970. *URL: http://ddi. cs. uni-potsdam. de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican. htm (visited on June 10, 2015).*

[12] Howard Gutowitz. *Cellular automata: Theory and experiment.* MIT press, 1991.

[13] Zia Ul Haq, Gul Faraz Khan, and Tazar Hussain. A comprehensive analysis of xml and json web technologies. *New Developments in Circuits, Systems, Signal Processing, Communications and Computers*, pages 102–109, 2013.

[14] David Harris and Sarah L Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[15] Andrew Ilachinski. *Cellular automata: A discrete universe*. World Scientific, 2002.

[16] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–4, 2010.

[17] Elshad Karimov. Stacks. In *Data Structures and Algorithms in Swift*, pages 27–32. Springer, 2020.

[18] Lemont B Kier, Paul G Seybold, and Chao-Kun Cheng. *Modeling chemical systems using cellular automata*. Springer Science & Business Media, 2005.

[19] Klayton Kowalski. Game development tutorial — cellular automata and procedural map generation. 2020.

[20] Pierre-Yves Louis and Francesca R. Nardi. *Probabilistic Cellular Automata: Theory, applications and future perspectives*. Springer International Publishing, 2018.

[21] Geraint Luff. Tiny validator (for v4 json schema). 2018.

[22] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Spicing up map generation. In *European Conference on the Applications of Evolutionary Computation*, pages 224–233. Springer, 2012.

[23] Antonia Mavroudi. Simulating city growth by using the cellular automata algorithm. 2007.

[24] Lauren McCarthy, Casey Reas, and Ben Fry. *Getting started with P5. js: Making interactive graphics in JavaScript and processing*. Maker Media, Inc., 2015.

[25] Guy H Orcutt. Simulation of economic systems. *The American Economic Review*, 50(5):894–907, 1960.

[26] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273, 2016.

[27] Paul Rendell. Turing universality of the game of life. In *Collision-based computing*, pages 513–539. Springer, 2002.

[28] Paul Rendell. A simple universal turing machine for the game of life turing machine. In *Game of Life Cellular Automata*, pages 519–545. Springer, 2010.

[29] James F. Robeson. *Logistics Handbook*. Free Press, 2011.

[30] Diego Ruiz-Moreno, Paula Federico, and Graciela Canziani. Population dynamics models based on cellular automata that includes habitat quality indices defined through remote sensing. 06 2002.

[31] Joel L Schiff. *Cellular automata: a discrete view of the world*. John Wiley & Sons, 2011.

[32] Richard Southwell and Jianwei Huang. Complex networks from simple rewrite systems. *CoRR*, abs/1205.0596, 2012.

[33] Bharath Srinivasan. Words of advice: teaching enzyme kinetics, 2021.

[34] Tommaso Toffoli and Norman Margolus. *Cellular automata machines: a new environment for modeling*. MIT press, 1987.

[35] Ajit Kumar Verma, Srividya Ajit, Durga Rao Karanki, et al. *Reliability and safety engineering*, volume 43. Springer, 2010.

[36] Eric W. Weisstein. Moore neighborhood.

[37] Joseph N Wilson and Gerhard X Ritter. *Handbook of computer vision algorithms in image algebra*. CRC press, 2000.

[38] S Wolfram. Statistical mechanics of cellular automata. *Rev. Mod. Phys.; (United States)*, 55, 7 1983.

[39] S. Wolfram. *A new kind of science*. Wolfram, 2002.

[40] Fulong Wu. Calibration of stochastic cellular automata: the application to rural-urban land conversions. *International journal of geographical information science*, 16(8):795–818, 2002.

## Appendix A

## Interview Questionnaire

## Part 1: Understanding the User Interface

Take a look around the user interface. I'll ask you here if you can identify some things.

1. Were you able to identify the simulation window? **(Y/N)**
2. Were you able to identify the rules editor? **(Y/N)**
3. Were you able to identify the controls for the system? **(Y/N)**

***Question:*** **Did you have any difficulty in identifying any of these elements? Explain.**

## Part 2: Understanding the Rules

This section will test your understanding of the rules and will make you interface with the simulator through those rules. You might want to consult the help page for the rules.

### Part 1: Simple Color Changing

1. Refresh the page. By default you should see the rules for Conway's Game of Life Cellular Automata, the most popular cellular automata.
2. You can see their colors clearly. Black or white. Try changing the color of the black cells to red and make it appear on the simulator
3. Don't forget to apply the rules by pressing the button.
4. How easy/hard was this to you?

### Part 2: Changing the Criteria - Totalistic

For reference, the rules for CGOL are as follows;

1. Any live cell with **two** or **three** live neighbours **survives**.
2. Any dead cell with **three** live neighbours becomes a **live** cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

And notice there is a help page for the rules

Now change the rules a bit, to this:

1. Any live cell with two or three live neighbours **dies**.
2. Any dead cell with **four** or **five** live neighbours becomes a **live** cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

***Question:*** **Were you able to do this? How easy/hard was it?**

## Part 3: Probabilistic CA

Now refresh the page again. Let’s try enabling the fire forest propagation mode. Follow the instructions:

1. In the enter state toolbox, enter 3
2. Press the ‘Fire Sim’ radio button on top of the rules editor
3. Apply the rules
4. Press the ‘Firesim’ button on top of the page.

***Question:*** **Honest feedback, what did you think of this process of changing?**

Okay, now let’s try playing with the probabilistic rules:

1. Take a minute to read the prescribed rules in the editor. What do you think it means?
2. Try changing the probabilities of spreading to 0 in all cases. Did you manage to do this?
3. Let’s just make everything burn. Regardless of how many neighbours any unburnt cell has, make sure it has a 100% chance of burning down. (hint: consult the rules help)

***Question:*** **Was there anything hard that you had to do in this process? Please Explain**

## Part 4: Making your Own Rules!

This is the final part of the interview. Refresh the page. I’m going to make you create a map generator for Game Development. You can take inspiration from the Conway’s Game of Life Rules to work your way around this. NB: You only need to write the rules.

The specifications are as follows:

There are two types of cells: Sea and Land. Set their colors appropriately.

1. If any cell has ≤ 4 neighbours that are of type 'Sea', then the state of the next cell should be a 'Sea'.
2. Otherwise, it should be 'Land'.

Translate the specifications above to the rules textarea.

***Question:*** **How easy/hard was this for you?**

# Appendix B

# Default Rules

## B.1 Conway's Game of Life

```
{
    "$_meta": {
        "num_states": 2,
        "colors": {
            "0": "black",
            "1": "white"
        }
    },
    "0": {
        "next": {
            "conditional_requirements": [
                {
                    "type": "totalling",
                    "neighbour_state": 1,
                    "total": [
                        3
                    ]
                }
            ],
            "satisfied": 1,
            "else": 0
        }
    },
    "default": {
        "next": {
            "conditional_requirements": [
                {
                    "type": "totalling",
                    "neighbour_state": 1,
                    "total": [
                        2,
                        3
                    ]
                }
            ],
            "satisfied": 1,
            "else": 0
        }
    }
}
```

## B.2 Forest Fire Simulator

```
{
    "$_meta": {
        "num_states": 3,
        "colors": {
            "0": "black",
            "1": "green",
            "2": "red"
        }
    },
    "1": {
        "next": {
            "conditional_requirements": [
                {
                    "type": "total-p",
                    "neighbour_state": 2,
                    "total": [
                        1,
                        2,
                        3,
                        4
                    ],
                    "p": 1
                },
                {
                    "type": "total-p",
                    "neighbour_state": 2,
                    "total": [
                        5, 6, 7, 8
                ],
                    "p": 0.8
                }
            ],
            "satisfied": 2,
            "else": 1
        }
    },
    "default": {
        "next": {
            "conditional_requirements": [
                {
                    "type": "probability",
                    "p": 1
                }
            ],
            "satisfied": 0,
            "else": 0
        }
    }
}
```

## B.3 Game Development Map Generation

```
{
    "$_meta": {
        "num_states": 2,
        "colors": {
            "0": "blue",
            "1": "green"
        }
    },
    "default": {
        "next": {
            "conditional_requirements": [
                {
                    "type": "totalling",
                    "neighbour_state": 1,
                    "total": [
                        1, 2, 3, 4
                    ]
                }
            ],
            "satisfied": 0,
            "else": 1
        }
    }
}
```

# Appendix C

# JSON Schema

```
{
type: "object",
properties: {
    $_meta: {
        type: "object",
        additionalProperties: false,
        properties: {
            num_states: {
                type: "integer"
            },
            colors: {
                type: "object",
                patternProperties: {
                    "^.*$": {
                        type: "string"
                    }
                }
            },
        },
        required: ["num_states"]
    },
},
patternProperties: {
    "^.*$": {
        $ref: "#/$defs/state",
    },
},
$defs: {
    state: {
        type: "object",
        additionalProperties: true,
        properties: {
            next: {
                type: "object",
                properties: {
                    conditional_requirements: {
                        $ref: "#/$defs/reqs_list"
                    },
                    satisfied: {
                        type: "integer"
                    },
                    else: {
                        type: "integer"
                    },
                },
                required: ["conditional_requirements", "satisfied", "else"],
            },
        },
    },
    reqs_list: {
        type: "array",
        items: [{
            $ref: "#/$defs/reqs"
        }],
    },
    reqs: {
        additionalProperties: true,
        type: "object",
        properties: {
            type: {
                enum: ["totalling", "total-p", "expression", "probability"],
            },
            neighbour_state: {
                type: "integer",
            },
            total: {
                type: "array",
                items: [{
                    type: "integer"
                }],
            },
        },
        allOf: [{
                if: {
                    required: ["type"],
                    properties: {
                        type: {
                            const: "totalling",
                        },
                    },
                },
        then: {
            required: ["neighbour_state", "total"],
            properties: {
                neighbour_state: {
                    type: "integer"
                },
                total: {
                    type: "array",
                    items: [{
                        type: "integer",
                    }, ],
                },
            },
        },
    },
    {
        if: {
            properties: {
                type: {
                    const: "total-p",
                },
            },
            required: ["type"],
        },
        then: {
            required: ["neighbour_state", "p", "total"],
            properties: {
                neighbour_state: {
                    type: "integer"
                },
                p: {
                    type: "number"
                },
                total: {
                    type: "array",
                    items: [{
                        type: "integer",
                    }, ],
                },
            },
        },
    },
    {
        if: {
            properties: {
                type: {
                    const: "expression",
                },
            },
            required: ["type"],
        },
        then: {
            required: ["lhs", "rhs", "cmp"],
            properties: {
                lhs: {
                    type: "object",
                    properties: {
                        neighbour_states: {
                            type: "array",
                            items: [{
                                type: "integer"
                            }],
                        },
                    },
                },
                rhs: {
                    type: "object",
                    properties: {
                        neighbour_states: {
                            type: "array",
                            items: [{
                                type: "integer"
                            }],
                        },
                    },
                },
                cmp: {
                    enum: ["<", ">", "="],
                },
            },
        },
    },
    {
        if: {
            properties: {
                type: {
                    const: "probability",
                },
            },
            required: ["type"],
        },
        then: {
            required: ["p"],
            properties: {
```

```
                            p: {
                                type: "number"
                            },
                        },
                    },
                },
            ],
        },
    },
}
```