BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

AUTOMATA APPLIED IN VISUAL ARTS

AUTOMATY APLIKOVANÉ V UMĚNÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **KRYŠTOF ALBRECHT**
AUTOR PRÁCE

SUPERVISOR **prof. RNDr. ALEXANDR MEDUNA, CSc.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

157203

Institut:	Department of Information Systems (DIFS)
Student:	**Albrecht Kryštof**
Programme:	Information Technology
Title:	**Automata Applied in Visual Arts**
Category:	Theoretical Computer Science
Academic year:	2023/24

Assignment:

1. Based upon the supervisor's instructions, study selected types of automata, such as cellular or two-dimensional automata, which are suitable for this bachelor thesis.
2. Introduce modified versions of these automata. Consult the supervisor about this modification.
3. Study the properties of these automata and the languages accepted by them. Follow the supervisor's suggestions concerning this study.
4. Based on the supervisor's instructions, select suitable fields of visual art. Based upon the automata introduced in 2, design several applications in the selected fields.
5. Implement the applications designed in 4.
6. Evaluate the achieved results of this bachelor thesis. Discuss its possible further development.

Literature:

- Rozenberg, G., Salomaa, A. (eds.): *Handbook of Formal Languages*, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1
- Brinkmann, R.: *The Art and Science of Digital Compositing: Techniques for Visual Effects, Animation and Motion Graphics*, 2nd edition, Morgan Kaufmann, 2008, ISBN 978-0-123-70638-6
- Gažo, M.: *Vícedimensionální automaty a jejich aplikace v umění*. Brno, 2021. Bachelor's Thesis. Brno University of Technology, Faculty of Information Technology. 2021-06-14. Supervised by Meduna Alexander. Available from: https://www.fit.vut.cz/study/thesis/23696/

Requirements for the semestral defence:
Points 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Meduna Alexandr, prof. RNDr., CSc.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

This thesis introduces a new programming language for composition of 2D visual effects. The language is based on a modified version of cellular automata designed for composition. The primary target platform is the Godot game engine, where the visual effects run using fragment shaders, although the compiler is platform-agnostic.

Abstrakt

Tato práce představuje nový programovací jazyk, určený ke kompozici dvourozměrných vizuálních efektů. Jazyk je založen na upravené verzi celulárních automatů navržené pro kompozici. Hlavní platformou, kde efekty mají běžet, je herní engine Godot, kde jsou efekty realizovány pomocí fragment shaderů.

Keywords

automaton, cellular automaton, effect, VFX, shader, games, art, composition, DSL, GPU acceleration

Klíčová slova

automat, celulární automat, efekt, VFX, shader, hry, umění, kompozice, DSL, GPU akcelerace

Reference

ALBRECHT, Kryštof. *Automata Applied in Visual Arts.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. RNDr. Alexandr Meduna, CSc.

Automata Applied in Visual Arts

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of prof. RNDr. Alexander Meduna, CSc. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Kryštof Albrecht
May 8, 2024

Acknowledgements

I would like to thank prof. RNDr. Alexander Meduna, CSc. for the indispensable advice and guidance, this thesis would not be possible without his help. Likewise, my thanks goes to Ing. Tomáš Milet, Ph.D. for advice about the shader implementation.

Contents

Chapter 1

Introduction

This thesis introduces a new domain-specific language, which intends to ease the addition of highly interactive visual effects to 2D computer videogames made in the Godot game engine, leveraging the power and wide-ranging applications of cellular automata.

Cellular automata have been used to simulate a wide variety of phenomena across many fields, including flow of liquids and gases, using models such as FHP and the lattice Boltzmann method, propagation of shock waves, crystal growth, forest fires, chemical reactions, light propagation and traffic simulation. There also exist many simpler automata, such as the famous Conway's Game of Life or Langton's Loops. These simulations could be used to make the world of a game feel more alive and interactive, especially when an already visually appealing simulation is combined with artistic filters.

The language is based on the idea of „laws acting on matter". A user defines the effect as consisting of matter, such as gas, water or glass, and then applies processes to the matter, such as the flow or burning of the gas, evaporation of the water or breaking of the glass. It is particularly suited to implementing the visuals for spells in videogames, such as fireballs, magical looking barriers or lightning. These effects can also accept input from the surrounding game scene and affect it in turn.

As a foundation for the language, a new variant of cellular automaton, called the cellular system, was developed. It allows multiple simpler cellular automata to interact through a unified interface, combining their effects together into a single, more complex cellular automaton. This automaton can effectively contain multiple layers of cells that can interact. It is also able to store configurations from several steps back in time and use those configurations during a state transition.

The thesis is structured as follows. First, foundational concepts are covered, such as systems, automata and cellular automata specifically. The next part explores the techniques used in cellular automata modeling and art. Based on this, the following chapter defines the new cellular systems and explores their abilities and limitations. Afterwards, a language is introduced to define these systems. Finally, a short description of the compiler implementation and testing follows.

Chapter 2 introduces the basic concepts the new model is based on, starting with system theory concepts, such as feedback, signals and stability. These play a significant role in generating the intricate and unstable patterns cellular automata are known for. The automata theory follows. Classic one-dimensional automata are machines that read an input tape from left to right, using a read head. Four-way automata are machines that read a two-dimensional tape, having the ability to move in any one of the four cardinal

directions. Cellular automata do not feature a read head at all, instead altering the entire tape at once, based on a rule governing the transition of each cell locally.

Chapter 3 presents a sampler of four cellular automata techniques. The first relates to the past configurations of an automaton, where it can store and read its past configurations or be run backwards. The second is the movement of discrete particles on the grid, such as ants or falling rocks. The Margolus neighborhood makes it easier to implement movement. Particles can then be used to simulate the flow of fluids.

Chapter 4 develops the new cellular system model. This model is based on representing a cellular automaton as a functional block diagram, where each block represents a rule and entire configurations are passed between blocks. Then, it is proven that any cellular system can be converted to a cellular automaton and vice versa, meaning they have the same expressive power.

Chapter 5 lays out the Cellular System Language (CSL) used to define these systems and the visual effects. Its syntax and design was inspired by ancient Greek Stoic physical theories. A tour of the language is given, with more comprehensive documentation available as Appendix B.

Chapter 6 acts as a high-level overview of the implementation and its key algorithms. The language is implemented by a compiler, which generates game objects that can be used in the Godot game engine. The user invokes the compiler on their script and the resulting file can be drag-and-dropped into the Godot editor. That is, it converts a `.csl` file into a `.escn` file, which is the format Godot uses to represent game objects and scenes.

Chapter 7 tests these generated files in Godot. Finally, chapter 8 concludes the thesis with a future plan and ideas for further development.

Chapter 2

Automata and systems

This chapter aims to introduce the concepts underlying the language, namely various concepts from automata theory as well as system theory. Automata theory provides cellular automata, used for the final visual effect, while system theory suggests a new perspective on these automata and how to build them.

Section 2.1 defines systems and associated concepts, like feedback and stability. The subsequent sections explore the power of several types of automata. Firstly, one-dimensional finite state automata will be covered, then two-dimensional automata, alongside their respective languages. The main focus lies on cellular automata.

Finally, the automata concepts and the system concepts are brought together by defining cellular automata as systems containing a feedback loop.

2.1 Systems

Systems are the subject of study for *system theory*. At its core, the field tries to bring tools to understand and model complex things. It studies how the parts of something interact and what is, or is not, essential to those interactions [2].

A *system* is any set of interconnected parts. This can be anything: a car, a computer network or even a university. As an example, figure 2.1 shows a simplified block diagram of a home air conditioning system.

The system accepts the desired air temperature as an input from the user. The computer then compares the desired temperature and the temperature measured by the thermometer, driving the heating and cooling system accordingly. The output of the system is a change in room temperature.

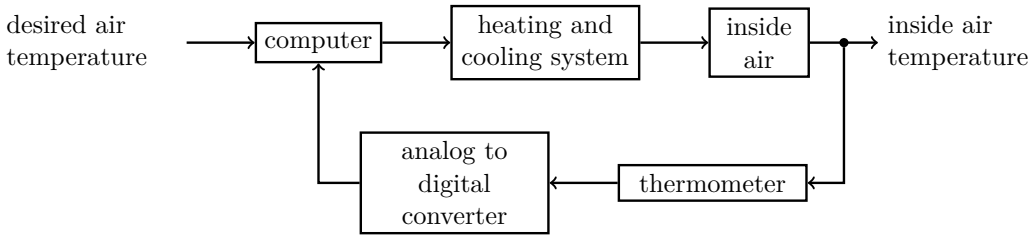
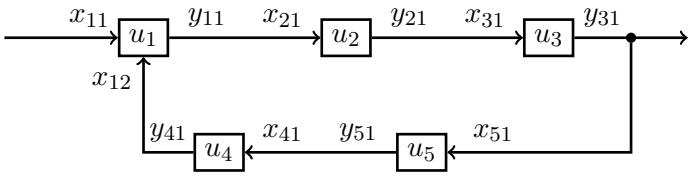


Figure 2.1: An air conditioning system [9]



$$U = \{u_1, u_2, u_3, u_4, u_5\}$$
$$R = \{(y_{11}, x_{21}), (y_{21}, x_{31}), (y_{31}, x_{51}), (y_{51}, x_{41}), (y_{41}, x_{12})\}$$

Figure 2.2: A formally defined air conditioning system $S = (U, R)$

Definition 2.1.1. A system S is a tuple $S = (U, R)$ [8], where:

- The universe $U = \{u_1, u_2, \ldots, u_n\}$ is a finite set of elements of the system,
- $u_i = (X, Y)$ is an element of the system, where X is the set of its input variables and Y the set of its output variables,
- the characteristic of the system R is the set of all connections:
  $R = \bigcup_{i,j=1}^{n} R_{ij}$
- and $R_{ij} \subseteq Y_i \times X_j$ is the connection of element u_i to element u_j.

The example system may be defined formally, as shown in figure 2.2. The following sections discuss the various concepts applicable to this basic example.

Signals

In the example, the various components are connected using arrows, representing some sort of interaction. The user enters a number into the computer. The computer outputs a control signal to drive the heating. The heating system transfers heat to or from the inside air.

In a computer simulation, these interactions can be modeled by a numerical quantity that represents them at some point in time. Heat can be represented in Joules, temperature in degrees Celsius and so on. This information either varies continuously in time (in the case of *continuous-time systems*) or changes in discrete steps (in the case of *discrete-time systems*).

Such functions of time carrying information are called *signals* [9]. For the purposes of this thesis, signals are defined as follows:

Definition 2.1.2. A signal x is a function $x : T \to L$, where:

- T is the time domain of the system,
- L is a language (sections 2.2, 2.3).

A signal with a discrete time domain (e.g. $T = \{1, 2, 3, \ldots\}$) is called a *discrete-time signal*, denoted as $x[k]$. A signal with a continuous time domain (e.g. $T = \langle 0, 1000 \rangle$) is a *continuous-time signal*, written as $x(t)$.

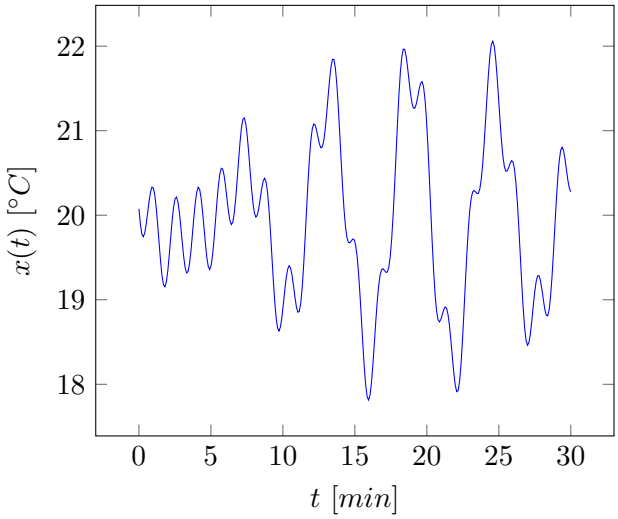


Figure 2.3: Signal from the thermometer

For example, the signal coming from the thermometer (figure 2.3) can be thought of as a mapping from a continuous time domain to a language describing the temperature. In this case, the language is equivalent to the set of real numbers.

Open and closed systems

Closed systems are systems that do not interact with their surrounding environment, being completely self-contained. Open systems have inputs and outputs, meaning they interact with surrounding elements. This interaction can be an exchange of matter, energy or information [8].

Open systems are typically parts of larger systems. The example system is open, since it accepts input from the user (x_{11}) and affects the surrounding air (y_{31}). It is also a part of a larger system, which is the entire house. Furthermore, each of the blocks of the example system is in itself an open system.

Feedback and stability

The term *feedback* refers to a situation in which two (or more) systems are connected together such that each system influences the other and their behavior is thus strongly coupled [1].

In the example system, the heating (cooling) element changes the air temperature, which is fed back into the computer controlling the heating and cooling. Alternatively, the system could have been designed around fixed parameters, such as the size of the room, how much heat flows in or out of the room and so on. The heat applied would be a function of just the temperature input by the user.

This would make the system susceptible to even the slightest change in conditions, making the room heat up or cool down too much. Utilizing feedback allows neglecting the fixed parameters, allows the system to self-regulate and makes it resilient against external conditions [1].

Feedback can also lead to *instability*, where the system veers wildly from its intended state. For example, consider a poorly designed audio setup at a concert. It may happen that the output of the speakers is fed back into the microphone. This makes the sound amplify indefinitely, which is, of course, very undesirable.

In visual art however, this kind of behavior is actually welcome, since it can produce unexpected and interesting patterns. Instability is what makes cellular automata in particular (section 2.3) so appealing.

Modeling using functional blocks

The block diagram in Figure 2.2 can be turned into a precise mathematical model. Each block has well-defined inputs and outputs, so by also formally defining the behavior for each block, it is possible to run a simulation of the shown system.

There are two types of blocks: stateless and stateful [8]. Stateless blocks are pure mathematical functions, mapping inputs to outputs. These include arithmetic functions, such as adders, and various logic gates. Stateful blocks determine their output based on their input, but also an internal state. As an example, an integrator, which is a block performing numerical integration, needs to store the value accumulated so far. In Figure 2.2 all blocks are stateless, except for u_3, which needs to store the current air temperature.

To run the model, it is necessary to know the order in which the blocks will evaluate, so that each block receives fresh inputs. This is done using topological ordering.

Definition 2.1.3. A *topological ordering* of a directed acyclic graph (DAG) $G = (V, E)$ is an ordering of the vertices V, such that if $(u, v) \in E$, then $u < v$.

Such ordering requires an acyclic system structure. Stateful blocks, however, are ignored during ordering since their output changes at the very end of a simulation step [8]. Therefore, the order of execution in Figure 2.2 is u_5, u_4, u_1, u_2, u_3, despite there being a feedback loop.

There are situations where the execution order of two or more blocks does not matter. As an example, they may require the exact same inputs. *Layering* an acyclic system groups together these blocks.

Definition 2.1.4. Given a DAG $G = (V, E)$, a *layering* of G is a partition of its node set V into disjoint subsets $V_1, V_2, \ldots, V_h$, such that if $(u, v) \in E$ where $u \in V_i$ and $v \in V_j$ then $i < j$. [5]

2.2 One-dimensional languages

In general, a *language* is a set of strings $L = \{w_1, w_2, \ldots\}$. In this chapter we consider one-dimensional languages, so a *string* w is simply a linear sequence of symbols from some alphabet Σ. For example, English is the set of all sensible sentences that can be formed using the Latin alphabet and that respect the grammatical rules of English.

Definition 2.2.1. Let Σ^* denote the set of all strings over the alphabet Σ. Every subset $L \subseteq \Sigma^*$ is a language over the alphabet Σ [10].

While there are many models for one-dimensional languages, only finite automata are presented in this chapter.

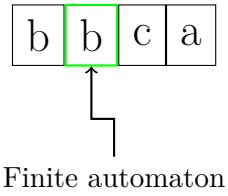


Figure 2.4: An input tape containing the string `bbca`

Finite automata

A *finite automaton*, also called a *finite state machine*, is a mathematical abstraction of an algorithm. It consists of a set of states, which the machine can transition between in response to an input.

The machine operates by reading the symbols of a string from an „input tape“ (shown in figure 2.4), moving to the right along the tape.

When from any given state, there is at most one next state to proceed to based on the current input symbol, we say that the automaton is *deterministic*.

Definition 2.2.2. A *deterministic finite automaton* (DFA) A is a quintuple $(Q, \Sigma, \delta, s, F)$ [10], where:

- Q is the finite set of states,
- Σ is the input alphabet,
- $\delta : Q \times \Sigma \to Q$ is the state transition function,
- $s \in Q$ is the starting state,
- $F \subseteq Q$ is the set of final states.

Example 2.2.1. An example DFA can be seen in figure 2.5. It is defined as $M = (Q, \Sigma, \delta, s, F)$, where:

- $Q = \{s_0, s_1, s_2\}$
- $\Sigma = \{a, b, c\}$
- $\delta = \{s_0 a \to s_1,\ s_1 a \to s_1,\ s_0 b \to s_2,\ s_2 b \to s_2,\ s_2 c \to s_1\}$
- $s = s_0$
- $F = \{s_1, s_2\}$

Let us next consider the string of letters `bbca` on the input tape. The machine always starts at the first letter of the tape (`b`) and in the state labeled as the starting state (s_0). Then, it reads the tape as illustrated in figure 2.6. The whole *run* can also be written as such:

$$s_0 bbca \vdash s_2 bca \vdash s_2 ca \vdash s_1 a \vdash s_1$$

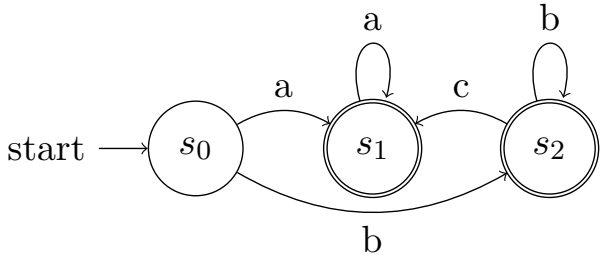


Figure 2.5: A deterministic finite automaton M

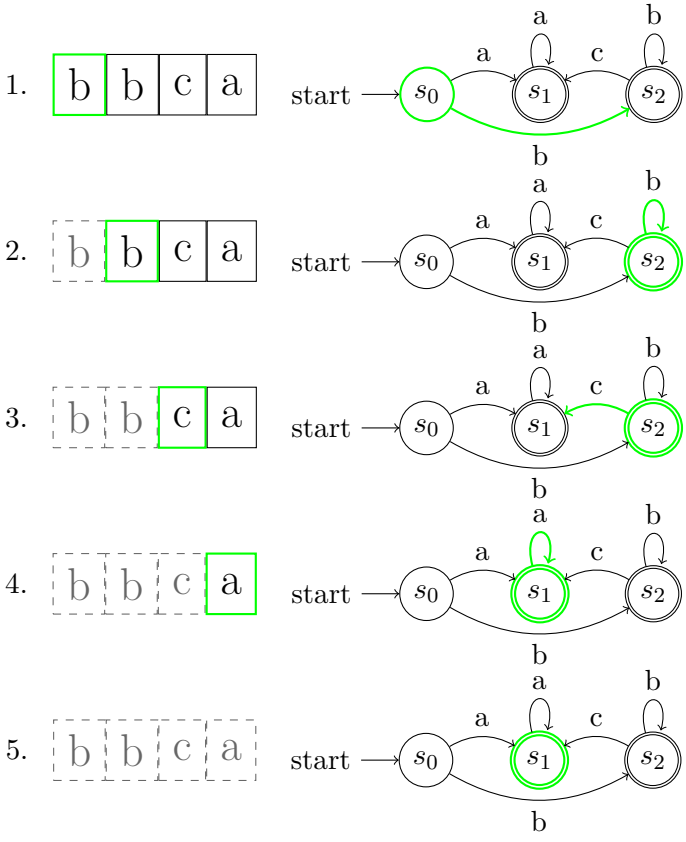


Figure 2.6: Operation of automaton M

$p_1 =$

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

$p_2 =$

a	b	c	b	a
a	b	c	b	a
a	b	c	b	a
a	b	c	b	a
a	b	c	b	a

Figure 2.7: Two-dimensional strings p_1 and p_2

2.3 Two-dimensional languages

Languages can also be generalized to two dimensions. Here, a string is not a sequence of symbols, but a grid of symbols.

Definition 2.3.1. A two-dimensional string (or a picture) over Σ is a *two-dimensional rectangular array* of elements of Σ. The set of all two-dimensional strings over Σ is denoted Σ^{**}. A two-dimensional language over Σ is a subset of Σ^{**} [10].

Example 2.3.1. An example can be seen in figure 2.7. Picture p_1 is a picture over the alphabet $\Sigma = \{0, 1\}$, while p_2 is a picture over the alphabet $\Sigma = \{a, b, c\}$.

It is worth explaining some of the notation and terms associated with pictures. Let $p \in \Sigma^{**}$, $q \in \Sigma^{**}$ be pictures. Then:

Size of picture p
$l_1(p)$ denotes the number of *rows* of p and $l_2(p)$ denotes the number of *columns* of p. The pair $(l_1(p), l_2(p))$ is the *size* of picture p [10].

Symbol at position (i, j) of p
$p(i, j)$ (or $p_{i,j}$) denotes the symbol at coordinates (i, j) in picture p [10]. The very top left symbol of p is the symbol at position $(0, 0)$, while the very bottom right symbol of p is the symbol at position $(l_1(p) - 1, l_2(p) - 1)$.

Concatenation of pictures p and q
The column concatenation of p and q (denoted $p \oplus q$) is defined only if $m = m'$ (their height is equal), and it is given by [10]:

$$p \oplus q = \left[\begin{array}{ccc|ccc} p_{0,0} & \cdots & p_{0,n} & q_{0,0} & \cdots & q_{0,n'} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ p_{m,0} & \cdots & p_{m,n} & q_{m',0} & \cdots & q_{m',n'} \end{array}\right]$$

The row concatenation of p and q (denoted $p \ominus q$) is defined only if $n = n'$ (their width is equal), and it is given by [10]:

$$p \ominus q = \begin{array}{|ccc|} \hline p_{0,0} & \cdots & p_{0,n} \\ \vdots & \ddots & \vdots \\ p_{m,0} & \cdots & p_{m,n} \\ \hline q_{0,0} & \cdots & q_{0,n'} \\ \vdots & \ddots & \vdots \\ q_{m',0} & \cdots & q_{m',n'} \\ \hline \end{array}$$

Boundary of picture p

For picture p of size $(m,\ n)$, the picture $\hat{p}$ is defined as the picture of size $(m+2,\ n+2)$ obtained by surrounding p with a special boundary symbol $\# \notin \Sigma$ [10]:

$$\hat{p} = \begin{array}{|ccccc|} \hline \# & \# & \cdots & \# & \# \\ \# & p_{0,0} & \cdots & p_{0,n-1} & \# \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \# & p_{m-1,0} & \cdots & p_{m-1,n-1} & \# \\ \# & \# & \cdots & \# & \# \\ \hline \end{array}$$

This symbol can be read by two-dimensional automata to recognize the picture boundary.

A programming language, such as C or Rust, is a set of all programs that are valid in the language. The application of 2D languages in this thesis is for them to represent the set of all *valid states of a 2D world (or some part of it).*

Example 2.3.2. Consider the example of a garden. Each spot in the garden can contain one of three objects: grass (■), vines (■) or a fence (■). These can be represented with the alphabet $\Sigma = \{■, ■, ■\}$.

Since vines cannot grow in mid-air, but have to grow on a fence, all valid gardens are represented by the language

$$G = \{p \mid p \in \Sigma^{**},\ p_{i,j} = ■ \Rightarrow p_{i-1,j} = ■ \vee p_{i+1,j} = ■ \vee p_{i,j-1} = ■ \vee p_{i,j+1} = ■\}$$

Figure 2.8 contains two pictures. String g_1 is a valid garden ($g_1 \in G$), while g_2 is not a valid garden, since it contains a vine surrounded completely by grass.

In the next sections, two types of automata are presented. *Four-way finite automata* for recognizing pictures (such as those belonging to the „garden language“) and *cellular automata* for generating new valid pictures from existing pictures, simulating the world the picture represents.

Four-way finite automata

Just as a one-dimensional finite automaton contains a „head“ to read an input tape from left to right, the head of a *four-way finite automaton* reads a picture from a two-dimensional tape. In addition, the head is allowed to move in *four directions* – right, left, up and down.

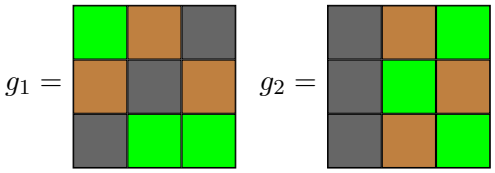


Figure 2.8: Two pictures representing a garden

Definition 2.3.2. A deterministic four-way finite automaton, *referred as 4DFA, is a* 7-*tuple* $A = (\Sigma, Q, \Delta, q_0, q_a, q_r, \delta)$ [10], where:

- Σ is the input alphabet,
- Q is a finite set of states,
- $\Delta = \{R, L, U, D\}$ is the set of „directions“,
- $q_0 \in Q$ is the „initial“ state,
- $q_a, q_r \in Q$ are the „accepting“ and the „rejecting“ state, respectively,
- $\delta : Q \setminus \{q_a, q_r\} \times \Sigma \to Q \times \Delta$ is the transition function.

The automaton always begins at position $(0, 0)$ of the input picture p and in state q_0. Then, the 4FA recognizes p, if it possibly moves around and halts in the *accepting* state q_a. If it halts in the rejecting state q_r, then the input is rejected. How this operates is illustrated in the following example.

Example 2.3.3. Consider the „garden language“ G from example 2.3.2. The 4FA to recognize this language is one that scans the picture left to right, top to bottom. If it encounters a ■, it checks if there is at least one neighboring ■. If there is not, the picture is rejected. If the automaton head moves to the bottom right corner of the picture, the picture is accepted. Figure 2.9 shows how M reads (and rejects) picture g_2 (from figure 2.8).

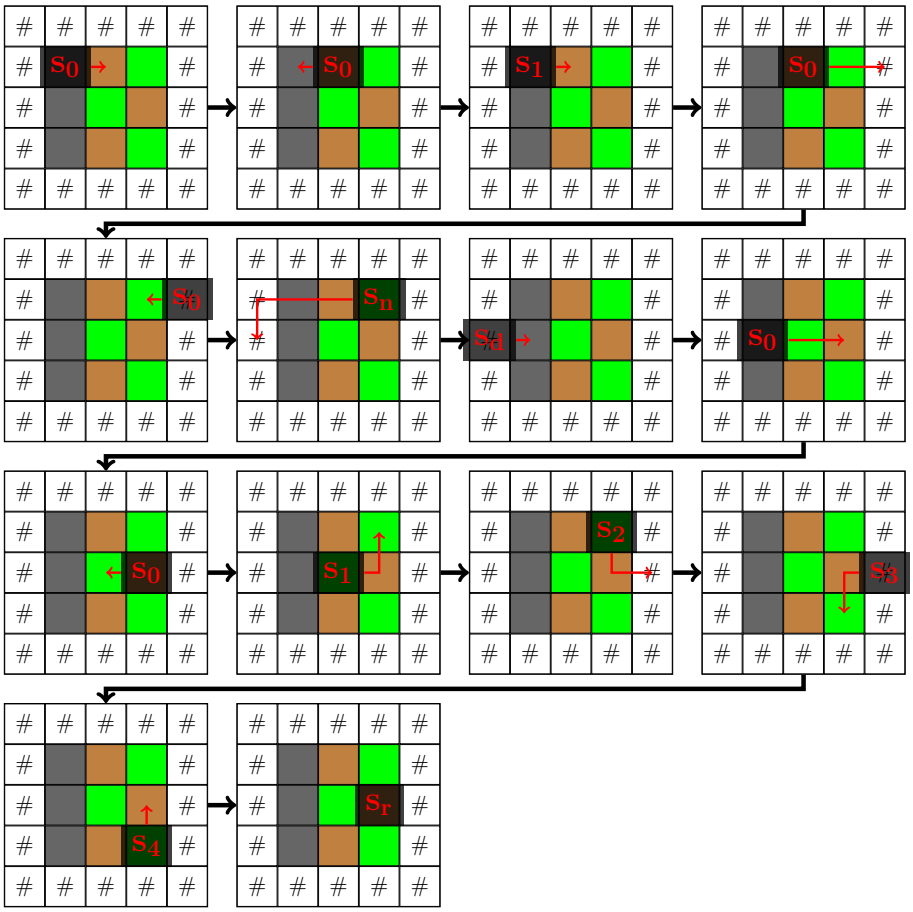


Figure 2.9: Operation of 4FA M

Cellular automata

In contrast to four-way finite automata, cellular automata (2CA) operate on the *entire* input picture concurrently at each step and there is no concept of a „head“ moving over the picture [10]. The exact operation of the automaton depends on the type used.

One of the well-known 2CA is *Conway's Game of Life*. It consists of a grid of cells, represented by a picture. Each symbol of the picture corresponds to a cell, that can either be „*alive*“ (□) or „*dead*“ (■). Whenever the 2CA performs a step, this picture (the grid) changes. Based on their neighboring cells, some cells die (□ → ■), while others come alive (■ → □).

Definition 2.3.3. Let $n \in \mathbb{N}$. A *cellular automaton* (2CA) C is a 4-tuple $C = (\Sigma, N, \delta, B)$, where:

- Σ is a finite set of symbols called the state alphabet, $\# \notin \Sigma$,
- $N \in V_1 \times V_2 \times \cdots \times V_n$, where $V_i = \mathbb{Z} \times \mathbb{Z}$, is the neighborhood,
- the total function $\delta : \Sigma'_1 \times \Sigma'_2 \times \cdots \times \Sigma'_n \to \Sigma$, where $\Sigma'_i = \Sigma \cup \{\#\}$, is the transition function,
- $B \in \{F, P, A, R\}$ is the boundary condition.

Continuing the example, *Life* can be defined formally using the above definition. Let $L = (\Sigma, N, \delta, B)$ be the *Life* 2CA. Σ is the set of all the possible states a cell can be in, so $\Sigma = \{■, □\}$. The neighborhood N tells which neighbors to examine when determining whether a given cell lives or dies.

Various neighborhoods are used in 2CA, as seen in figure 2.13. *Life* uses the so-called *Moore* neighborhood. The tuple N consists of all the relative positions of cells in the neighborhood:

$(-1,-1)$	$(-1,0)$	$(-1,1)$
$(0,-1)$	$(0,0)$	$(0,1)$
$(1,-1)$	$(1,0)$	$(1,1)$

$$N = ((0,0),(-1,-1),(-1,0),(-1,1),(0,1),(1,1),(1,0),(1,-1),(0,-1))$$

The function δ is the rule that determines how each cell changes, or how the 2CA transitions from one *configuration* to the next. The boundary condition B affects the behavior of δ for cells at the edge of the grid.

Definition 2.3.4. Let $C = (\Sigma, N, \delta, B)$ be a 2CA. A *configuration* of C is a picture $s \in \Sigma^{**}$.

A configuration of L is any picture over $\{■, □\}$. A picture containing any other symbol is not a configuration of L. This is illustrated in figure 2.10. A transition between valid configurations is called a *step* of the cellular automaton.

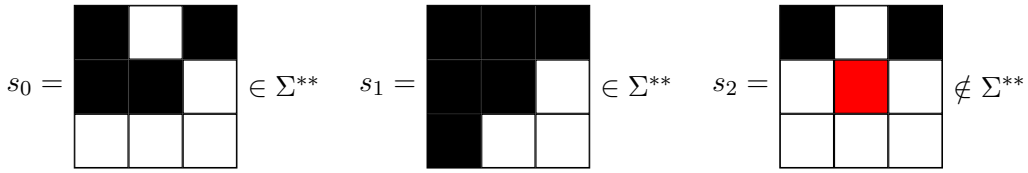


Figure 2.10: Two configurations of L and a picture (not a configuration)

Definition 2.3.5. Let $C = (\Sigma, N, \delta, B)$ be a 2CA, where $N = ((y_1, x_1), (y_2, x_2), \ldots, (y_n, x_n))$. Let s_1 and s_2 be configurations of C.

C makes a *step from* s_1 *to* s_2, written as

$$s_1 \vdash s_2$$

iff s_1 and s_2 are the same size and for all positions (i, j), such that $0 \leq i < l_1(s_1)$, $0 \leq j < l_2(s_1)$, the following holds:

$$s_2(i, j) = \delta(s_1(i + y_1, j + x_1),\ s_1(i + y_2, j + x_2),\ \ldots,\ s_1(i + y_n, j + x_n))$$

If the position $(i+y_i, j+x_i)$ is outside the bounds of s_1, then $s_1(i+y_i, j+x_i)$ is undefined by picture s_1 and is instead interpreted according to the boundary condition B: [1] [2]

$$s_1(i + y_i,\ j + x_i) = \begin{cases} \# & \text{if } B = F \\ s_1(i + y_i \bmod l_1(s_1),\ \ j + x_i \bmod l_2(s_1)) & \text{if } B = P \\ s_1(\mathit{clamp}(0, l_1(s_1) - 1, i + y_i),\ \mathit{clamp}(0, l_2(s_1) - 1, j + x_i)) & \text{if } B = A \\ s_1(i + y_i \text{ ref } l_1(s_1) - 1,\ \ j + x_i \text{ ref } l_2(s_1) - 1) & \text{if } B = R \end{cases}$$

It is δ that determines which configuration follows another. For L the function δ is defined as follows. Any dead cell with exactly 3 live neighbors comes alive, otherwise it stays dead. Any live cell with less than 2 or more than 3 live neighbors dies, otherwise it stays alive. Formally:

$$\delta(c, n_1, n_2, \ldots, n_8) = \begin{cases} \square & \text{if } c = \blacksquare \wedge |A| = 3 \\ \blacksquare & \text{if } c = \square \wedge (|A| < 2 \vee |A| > 3) \\ c & \text{otherwise} \end{cases}$$

$$A = \{n \mid n \in \{n_1, \ldots, n_8\}, n = \square\}$$

n_1	n_2	n_3
n_8	c	n_4
n_7	n_6	n_5

The way the parameters of δ map to neighboring cells is given by N, as mentioned in the definition. Taking figure 2.10 as an example, $s_0 \nvdash s_1$. Both are valid configurations, but in order for $s_0 \vdash s_1$, the center left cell of s_1 would have to be white.

The boundary conditions are illustrated in figure 2.14. The gray square represents the current cell and the blue squares cells in its neighborhood. The leftmost cell with a dashed outline is the cell outside the bounds of the 2CA configuration. For the example 2CA L, $B = F$ (*fixed*) and any out-of-bounds cells are considered dead by δ.

[1] $\mathit{clamp}(l, u, x) = \begin{cases} l & \text{if } x < l \\ u & \text{if } x > u \\ x & \text{otherwise} \end{cases}$

[2] $x \text{ ref } y = y - |(x \bmod 2y) - y|$

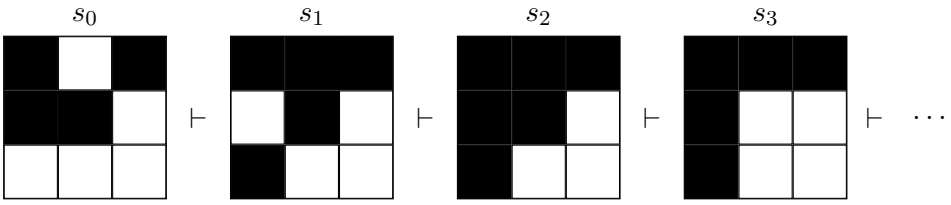


Figure 2.11: Run of 2CA L

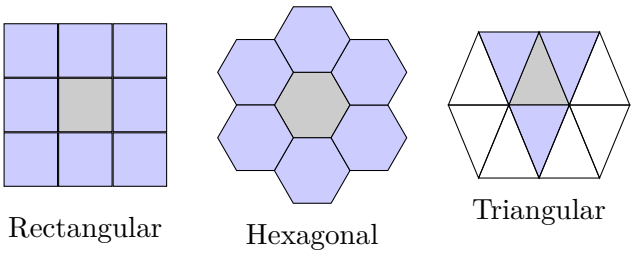


Figure 2.12: Various grid shapes

Definition 2.3.6. A *computation* (or a *run*) of 2CA C beginning from configuration s_0 is an infinite sequence of its configurations $s_0, s_1, s_2, \ldots$, where $s_i \vdash s_{i+1}$ for all $i \geq 0$.

A run of L beginning from configuration s_0 can be seen in figure 2.11.

The grid of a 2CA need not necessarily be rectangular. Any shape that can be tiled can be used as a grid, as seen in figure 2.12. A hexagonal grid, for instance, is used by the *FHP* 2CA, used for basic simulation of fluids [3]. Note, that the above definition does not directly allow for non-rectangular grids, but it is possible to emulate them using a rectangular grid.

Many other *neighborhoods* are also used (figure 2.13). Particularly of note is the *Margolus* neighborhood. Instead of a center cell examining the same neighbors every time step, the grid is partitioned into 2×2 blocks [3]. Two partitionings are possible – odd and even partitions, as shown by the blue blocks delimited by the solid and dashed lines, respectively. The neighborhood alternates between these two situations, at even and odd time steps.

Figure 2.14 also shows other boundary conditions. *Periodic* means the grid wraps around, so the value from the opposite side of the grid is used. *Adiabatic* takes the value of

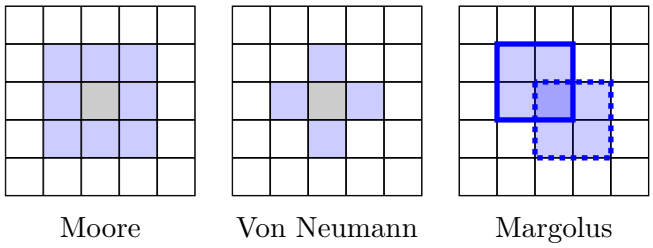


Figure 2.13: Various types of neighborhoods

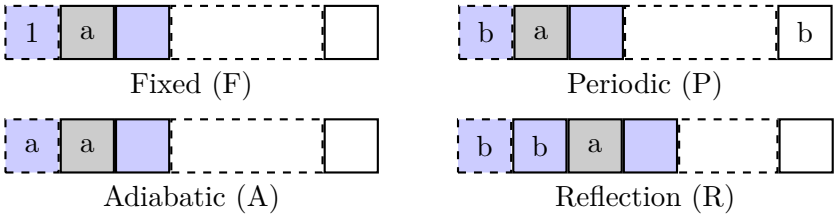


Figure 2.14: Boundary conditions

the cell on the edge of the grid. *Reflection* makes the boundary act as a „mirror“, flipping the choice of neighbor along the boundary.

2.4 Cellular automata as feedback systems

It is also possible to view 2CA from a different perspective, by interpreting a 2CA as a *system* and a computation (run) of the 2CA as a *signal*.

Let $C = (\Sigma, N, \delta, B)$ be a 2CA and $s_0, s_1, s_2, \ldots$ be its run. A run is an infinite sequence of pictures. An infinite sequence is a total function $\mathbb{N} \to X$, where X is a set. Since, then, a computation is the total function $\mathbb{N} \to \Sigma^{**}$ and a discrete-time signal is a mapping $\mathbb{N} \to L$, where L is any language, a computation is a discrete-time signal. This is illustrated in figure 2.15.

Then, C can be represented using a *functional block diagram* (figure 2.16), much like the air conditioning system from section 2.1. C is a system that accepts a previous configuration and outputs the new configuration. The new configuration is then fed back into C, repeating the cycle, advancing the automaton. This perspective forms the starting point for the new variant of 2CA introduced in chapter 4.

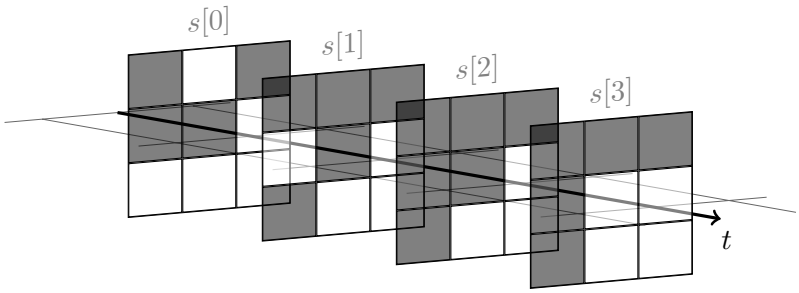


Figure 2.15: s_t interpreted as a signal $s[t]$

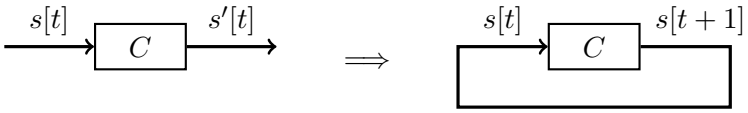


Figure 2.16: 2CA as a feedback system

Chapter 3

Art using cellular automata

The previous chapter introduced *Conway's Game of Life* as a 2CA. This chapter provides a sampler of more 2CA techniques and their potential application in visual art, with the goal of identifying key requirements for the language. Some of these 2CA are commonly used for simulation, representing „miniature worlds". Some are purely for visual spectacle. This chapter also introduces some additional terms used when talking about 2CA. All of the content in this chapter has been adapted from [11] and [3].

3.1 Using the past

Figure 3.1 shows the mesmerizing pattern produced by the „`TIME-TUNNEL`" 2CA. This 2CA is special, because it uses not only the present configuration to calculate the next state, but also the previous configuration.

Here, a cell can either be black (■) or white (□). The transition function is as follows. First, take all the cells in the Von Neumann neighborhood (north, west, east, south and center). If they are all the same color, make the center cell black, otherwise make it white. Finally, take this resulting color and compare it with the *the center color of the preceding configuration*. If they match, the center cell is black, otherwise white.

Another specialty of this 2CA is its *reversibility*. There exists a function δ^{-1} that can run the 2CA in reverse.

Definition 3.1.1. A cellular automaton is *reversible* if for every possible configuration of the automaton, there exists one and only one predecessor [11]. That is, given a 2CA $C = (\Sigma, N, \delta, B)$, C is reversible iff $\forall a \in \Sigma^{**}\ \exists b \in \Sigma^{**} : b \vdash a$ and $\forall a, b, c \in \Sigma^{**} : a \vdash c \wedge b \vdash c \implies a = b$.

There are countless uses for this technique of storing past configurations. For example, adding traces to moving patterns, as shown in figure 3.1.

3.2 Motion

A paramount 2CA technique is *motion*. Consider some „object" on the grid. Something like a grain of sand, a bullet or some other *particle*. How does it move across the grid? How does it stay intact? How can it collide with other objects?

In a 2CA, a cell has no control over, for example, its south neighbor. It cannot just „put" the particle there. The cell can only copy content from its neighbors or erase its own

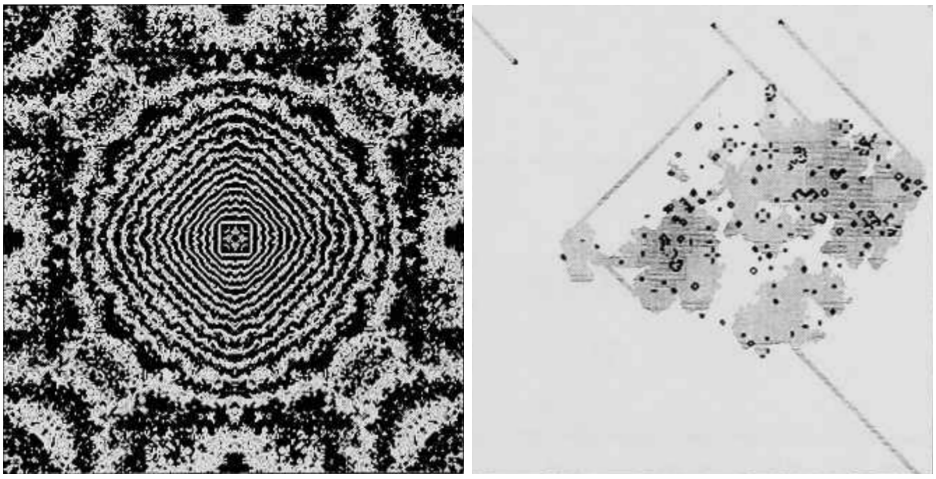


Figure 3.1: `TIME-TUNNEL` (left) and `LIFE` with traces (right) [11]

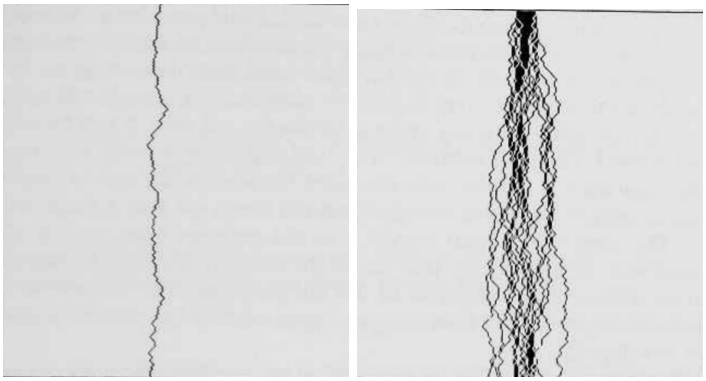


Figure 3.2: A random walk of a single particle (left) and many particles (right) [11]

content. For proper movement a sort of „handshake“ has to occur. The source cell has to be certain that the target cell can accept the particle. Similarly, the target has to know that the source can provide the particle. Then, the source can erase its particle and the target can create a particle within itself. Otherwise, particles might be lost or duplicated.

Figure 3.2 shows such a situation, a „random walk“ of particles. Please note, this 2CA is *actually one-dimensional*. All cells exist on a single line, the picture simply shows how this single-line configuration evolves over time (the run).

During a step, each particle on the line moves either left or right, at random. The left picture shows how a single particle walks the line. For a single particle, the rule is simple. If a cell has the particle, erase it, a neighbor will pick it up. Each cell also contains a random bit. An empty cell will examine its two neighbors. If the left neighbor has a particle and its random bit says „right“, copy the particle. Likewise for the right neighbor.

In the case of multiple particles, there is a problem. What if both neighbors of an empty cell contain particles that want to move into the cell? Both neighbors will erase their particle. But this cell can only hold one, leading to particles vanishing.

A solution is to partition the line into two-cell blocks. If one of the cells has the particle, it will move it to the other side within the block. If both cells have the particle, they will

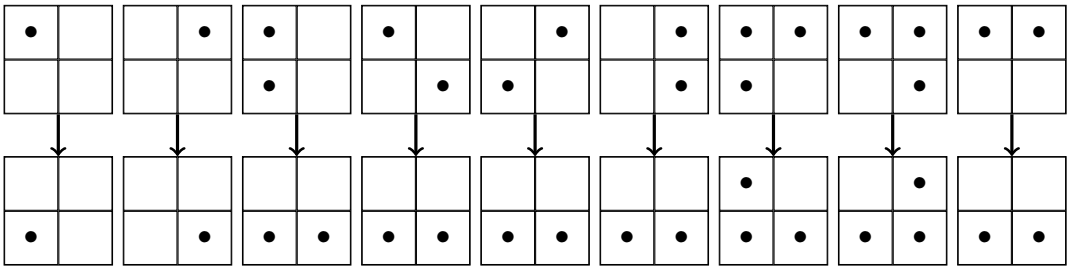


Figure 3.3: The sand rule (other configurations are unchanged) [3]

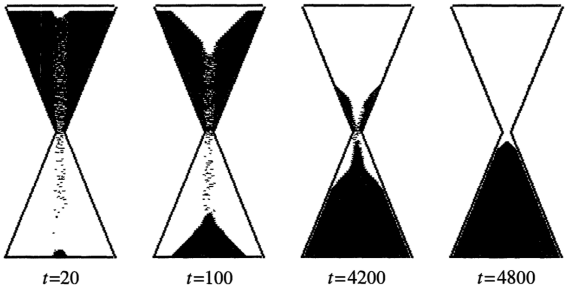


Figure 3.4: An hourglass simulation using the sand rule [3]

swap places. This way, particles will never try to occupy the same space. In practice, it is also necessary shift the partitioning back and forth every other step. Otherwise a particle would stay in these two cells forever. The right picture in figure 3.2 shows the run of this rule.

An important aspect is the *speed* at which these particles can move.

Definition 3.2.1. The *speed of light* (denoted as c) is the maximum speed at which information can propagate in a 2CA [11].

For example, the speed of light for a Moore neighborhood is $c = 1$, meaning any pattern can move at most one cell in each step of the 2CA. The particles in figure 3.2 do achieve this speed. The `LIFE` 2CA has $c = 1$ (using the Moore neighborhood), but no pattern can ever achieve it. The fastest an object can move in `LIFE` is at $c/2$ (moving one cell every two steps) [6].

3.3 The Margolus neighborhood

The Margolus neighborhood is a two-dimensional version of the partitioning scheme described above. On even time steps, the grid is partitioned into two-by-two blocks. On odd time steps, this partitioning is shifted down and to the right, as shown in figure 2.13.

One of the 2CA that makes use of this neighborhood is the *sand rule* (figure 3.3). It is a stylized simulation of how sand grains fall and pile up on top of each other.

The figure shows that it is common to write the rule for the entire block, not each cell individually.

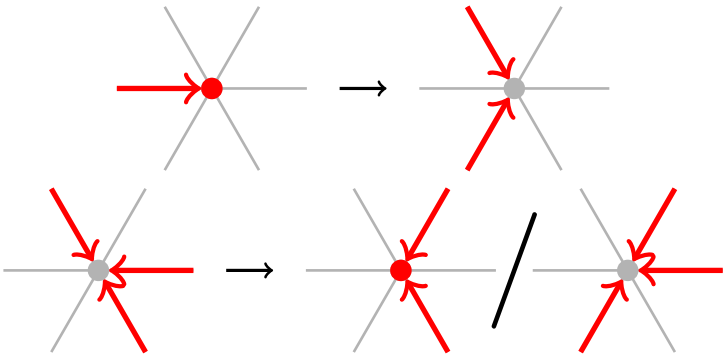


Figure 3.5: Example of a collision in FHP-III [3]

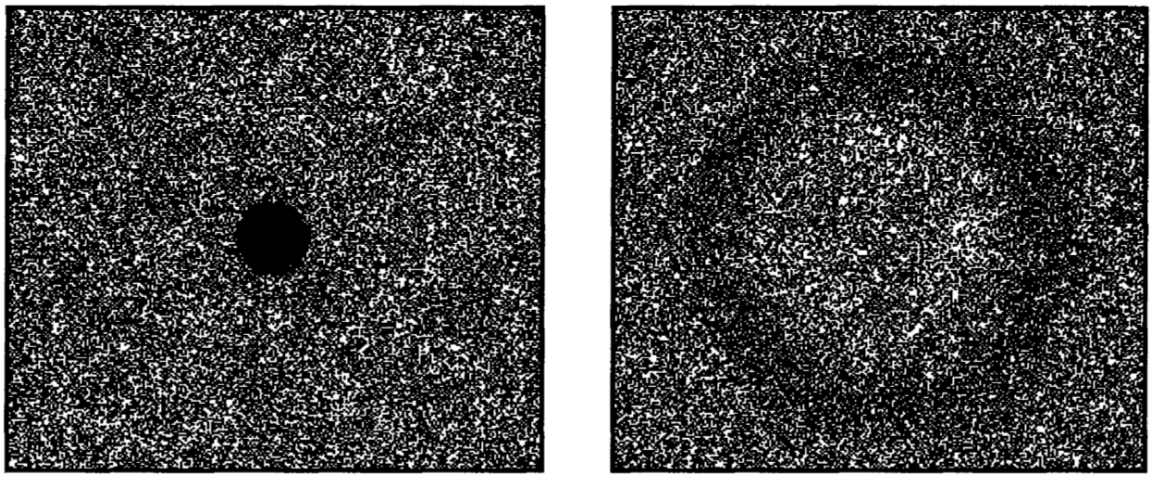


Figure 3.6: Development of a sound wave in FHP gas [3]

3.4 Liquids, gases and waves

Another appealing technique is the simulation of liquids and gases. Figure 3.6 shows how the so-called *FHP-III* 2CA can produce a „sound wave“. It simulates the travel and collision of fluid particles in a two-dimensional space, similar in spirit to the sand rule.

Unlike the sand rule, FHP-III uses a hexagonal grid. Each cell can contain up to six *moving particles* and one *particle at rest*. Each moving particle corresponds to one of the six possible directions of travel. The state of each cell is thus represented by seven bits.

The step happens in *two phases*, namely the „collision phase“ and the „movement phase“. In the collision phase, particles change direction based on the other particles in the cell, as shown in figure 3.5. The movement phase works with this new updated configuration. In the movement phase, each cell erases its content and then copies any moving particles traveling here from its neighbors. The particle at rest does not move, but can start moving during the collision phase.

Chapter 4

Cellular systems

This chapter introduces a new variant of cellular automata, which *serves as the formal semantic model* of the language and its compiler. The goal of this model is to allow a more natural way of thinking about cellular automata for the end user.

A traditional 2CA operates by applying a single transition function to its current configuration. It is usually thought of in terms of „cell state“ and a „rule that changes the cell state“. The focus is on cells and their local microscopic interactions, usually with the goal of modeling something larger, macroscopic. Specifying the interactions between particles of fluid (microscopic) to simulate flow of fluid (macroscopic), for example.

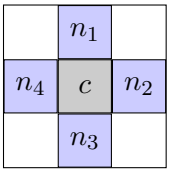
The new model, inspired by ancient Greek theories of physics, turns this focus around. It allows composing a 2CA, such that the user thinks in terms of „matter“ or „stuff“ and „processes“. The user defines the miniature world in terms of the matter that makes it up and the processes that act within it. It is still a 2CA, but defined in a different manner. This is best explained using an example.

Example 4.0.1. Consider the following visual effect for a 2D game. We have a corridor with moss growing on the walls. We want the moss to both grow and also react to the player, so that when they come near the moss with a torch, it catches fire and burns. This example will be used and gradually expanded throughout the entire chapter.

The 2CA $M = (\Sigma, N, \delta, F)$ will model this effect. Each cell can either be empty (□), contain moss (■), or contain burning moss (■). Each cell can also contain the flame of the torch (⊡ , ⊡ , ⊡). So, $\Sigma = \{□, ■, ■, ⊡, ⊡, ⊡\}$.

The neighborhood N will be the *Von Neumann* neighborhood (figure 2.13). δ will also be non-deterministic, relying on a certain amount of chance:

$$\delta(c, n_1, n_2, n_3, n_4) = \begin{cases} ■ & \text{if } c = □ \wedge |G| > 0 \wedge 10\% \text{ chance} \\ ■ & \text{if } c = ■ \wedge |B| > 0 \wedge 25\% \text{ chance} \\ □ & \text{if } c = ■ \wedge 20\% \text{ chance} \\ ⊡ & \text{if } c = ⊡ \\ c & \text{otherwise} \end{cases}$$



$$G = \{n \mid n \in \{n_1, \ldots, n_4\}, n = ■ \vee n = ⊡\}$$

$$B = \{n \mid n \in \{n_1, \ldots, n_4\}, n = ■ \vee n = ⊡\}$$

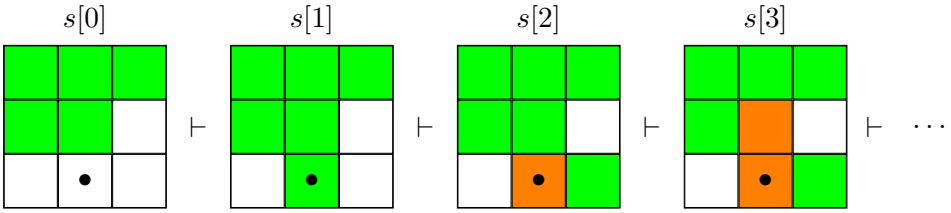
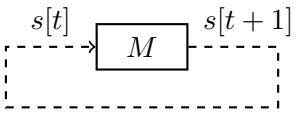


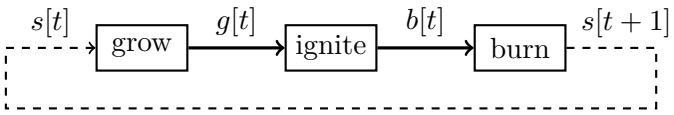
Figure 4.1: Run of 2CA M

The symbol # is interpreted as $\square$. Simply put, this is what happens in each step. Moss has a chance to spread to adjacent cells. It has a chance to catch fire if any surrounding moss is burning. Burning moss has a chance to burn up completely. Any moss touching the torch catches fire. A run of this 2CA is shown in figure 4.1.

In section 2.4 we have seen that it is possible to define a 2CA as a system with a feedback loop. Defining M in this way:



The transition rule of M is applied to $s[t]$, yielding $s[t+1]$, the next configuration. The *essence of the language* is to let the user say something along the lines of „I want green moss that grows, can burn and the player is able to ignite it with a torch.“. The *matter* is moss and fire. The *processes* are growth, ignition and burning. Splitting the above system:

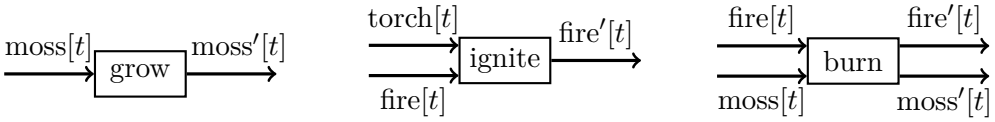


Now, M operates in three stages. First, the `grow` stage takes the picture $s[t]$ and modifies it, so that moss spreads to adjacent tiles, yielding $g[t]$. Second, `ignite` takes $g[t]$ and modifies it again, so that any moss touching a torch ignites. And finally, in `burn` the flames spread and moss burns up, yielding the next configuration $s[t+1]$. Effectively, this is the composition of multiple simpler 2CA rules.

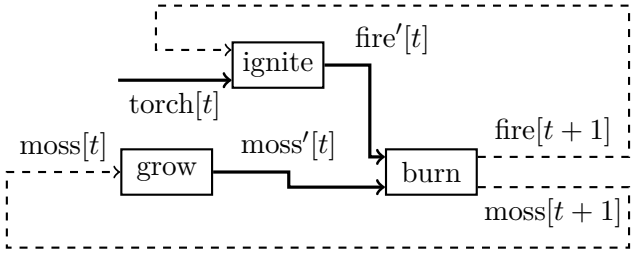
Please note, that the pictures $g[t]$ and $b[t]$ do *not represent configurations of the entire system as a 2CA*. They are merely signals that `ignite` used for communication between the blocks.

Next, note that the stages receive data they do not need. The signals $s[t]$, $g[t]$ and $b[t]$ are all pictures over Σ^{**}, so the `grow` block, for instance, receives data about which cells are burning, even though that has nothing to do with the process of growth.

Therefore, in order to limit the responsibilities of each system, it would be appropriate to split the original alphabet Σ into smaller alphabets and make each system handle only the data it needs:



Here, moss$[t] \in \{\square, \blacksquare\}^{**}$, fire$[t] \in \{\square, \blacksquare\}^{**}$ and torch$[t] \in \{\square, \boxdot\}^{**}$. Now, the responsibilities of each system are separate and clear: moss grows, torch ignites fire, fire burns moss. Reconnecting the blocks:



Notice, that there are now *two feedback connections*. In this system, `grow` and `ignite` now have well-defined responsibilities and are completely separate. They each have their own pictures to manage, moss$[t]$ and fire$[t]$. Once they have both made changes to the pictures, `burn` takes *both the pictures* and edits them *both at once*. Then, they are fed back to `grow` and `ignite`, ready for the next step. This setup is effectively equivalent to two cellular automata, that interact through the `burn` block.

We have arrived at what I call the *cellular system*. Such a system is made up of *blocks* (e.g. `ignite` or `grow`):

Definition 4.0.1. Let $m, n \in \mathbb{N}$. A *block* is a triple $B = (C, I, O)$, where:

- C is the *internal object* of the block, which describes its behavior,
- $I = \{(x_1, \Sigma_1), (x_2, \Sigma_2), \ldots, (x_n, \Sigma_n)\}$ is the set of *inputs*, where x_i is an identifier and Σ_i is an alphabet (the alphabet of the input picture),
- $O = \{(y_1, \Sigma'_1), (y_2, \Sigma'_2), \ldots, (y_m, \Sigma'_m)\}$ is the set of *outputs*, where y_i is an identifier and Σ_i is an alphabet (the alphabet of the output picture).

What the block does is given by its *internal object*. The simplest such object is a *rule*, which expands on the standard definition of 2CA. It has a neighborhood and a boundary condition, just like a traditional 2CA, but it has multiple alphabets as well as multiple transition functions. This is what allows the rule to affect multiple pictures at once.

Definition 4.0.2. Let $m, n \in \mathbb{N}$. An *elementary rule* is a 4-tuple $R = (S, N, T, B)$, where:

- $S = (\Sigma_1, \Sigma_2, \ldots, \Sigma_m)$, where Σ_i is an alphabet, $\# \notin \Sigma_i$,
- $N \in V_1 \times V_2 \times \cdots \times V_n$, where $V_i = \mathbb{Z} \times \mathbb{Z}$, is the neighborhood,
- $T = (\delta_1, \delta_2, \ldots, \delta_m)$,
- the total function $\delta_i : \Sigma_1^1 \times \Sigma_2^1 \times \cdots \times \Sigma_n^1 \times \Sigma_1^2 \times \Sigma_2^2 \times \cdots \times \Sigma_n^2 \times \cdots \times \Sigma_n^m \to \Sigma_i$, where $\Sigma_k^j = \Sigma_j \cup \{\#\}$, is a transition function,
- $B \in \{F, P, A, R\}$ is the boundary condition.

Each alphabet corresponds to some kind of „matter“ in the modeled effect. For each of these alphabets there is a transition function, which is a function of *all* the alphabets. For example, the behavior of the `burn` block can be defined as $R = (S, N, T, F)$, where:

- $S = (\Sigma_f, \Sigma_m)$, where $\Sigma_f = \{\square, \blacksquare\}$, $\Sigma_m = \{\square, \blacksquare\}$,
- $N = ((0,0), (-1,0), (0,1), (1,0), (0,-1))$ is the *Von Neumann* neighborhood,
- $T = (\delta_f, \delta_m)$,
- $$\delta_f(c^f, n_1^f, \ldots, n_4^f, c^m, n_1^m, \ldots, n_4^m) = \begin{cases} \blacksquare & \text{if } c^m = \blacksquare \wedge |B| > 0 \wedge 25\% \text{ chance} \\ \square & \text{if } c^f = \blacksquare \wedge c^m = \square \\ c^f & \text{otherwise} \end{cases}$$
$$B = \{n \mid n \in \{n_1^f, \ldots, n_4^f\}, n = \blacksquare\}$$
- $$\delta_m(c^f, n_1^f, \ldots, n_4^f, c^m, n_1^m, \ldots, n_4^m) = \begin{cases} \square & \text{if } c^f = \blacksquare \wedge c^m = \blacksquare \wedge 20\% \text{ chance} \\ c^m & \text{otherwise} \end{cases}$$

Definition 4.0.3. Let $R = (S, N, T, B)$ be an elementary rule, where $S = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$. Then, the block $(R, \{(1, \Sigma_1), (2, \Sigma_2), \ldots, (n, \Sigma_n)\}, \{(1, \Sigma_1), (2, \Sigma_2), \ldots, (n, \Sigma_n)\})$ is R *as a block*.

Essentially, an index for each alphabet is added, so that it is possible to identify the inputs and outputs of the transition functions. Therefore, the rule of `burn` as a block can be defined as $B = (R, \{(1, \Sigma_f), (2, \Sigma_m)\}, \{(1, \Sigma_f), (2, \Sigma_m)\})$. The larger system consists of multiple blocks and builds upon the definition of systems in section 2.1:

Definition 4.0.4. Let $n \in \mathbb{N}$. A *cellular system (CS)* is a 5-tuple $C = (U, B, \Lambda, R, F)$ where:

- $U = \{u_1, u_2, \ldots, u_n\}$ is a finite set of objects called *elements*,
- B is a finite set of blocks (defined above),
- $\Lambda : U \to B$ is a total function, assigning to each element a block,
- the partial injective function[1]
$$R : \bigcup_{i=1}^{n} \{(u_i, y) \mid y \in \text{outputs of } \Lambda(u_i)\} \to \bigcup_{j=1}^{n} \{(u_j, x) \mid x \in \text{inputs of } \Lambda(u_j)\}$$
such that $R((u, (y, \Sigma_o))) = (v, (x, \Sigma_i)) \implies \Sigma_o = \Sigma_i$, is the connection function,
- $F \subseteq R$ is the set of feedback connections,
- $\forall u \in U : (u, u) \notin P^+$, where $P = \{(u_i, u_j) \mid ((u_i, y), (u_j, x)) \in R \setminus F, \text{ for some } y, x\}$, in other words, no element can be connected to a previous element, unless it is a feedback connection.[2]

[1] A function is a binary relation f, such that $(x, y) \in f \wedge (x, z) \in f \implies y = z$. A binary relation is any set of 2-tuples.

[2] P^+ denotes the transitive closure of P.

The example system, shown in the above diagram, can then be defined as $M = (U, B, \Lambda, R, F)$ where:

- $U = \{\texttt{ignite}, \texttt{grow}, \texttt{burn}\}$,
- $B = \{I, G, B\}$, where B is the `burn` block defined earlier and I, G can be defined in a similar manner,
- $\Lambda = \{\texttt{ignite} \mapsto I, \texttt{grow} \mapsto G, \texttt{burn} \mapsto B\}$,
- $R = \{\ (\texttt{ignite}, (1, \Sigma_f)) \mapsto (\texttt{burn}, (1, \Sigma_f)), \quad (\texttt{grow}, (1, \Sigma_m)) \mapsto (\texttt{burn}, (2, \Sigma_m)),$
  $(\texttt{burn}, (1, \Sigma_f)) \mapsto (\texttt{ignite}, (1, \Sigma_f)), \quad (\texttt{burn}, (2, \Sigma_m)) \mapsto (\texttt{grow}, (1, \Sigma_m))\ \}$
- $F = \{(\texttt{burn}, (1, \Sigma_f)) \mapsto (\texttt{ignite}, (1, \Sigma_f)), (\texttt{burn}, (2, \Sigma_m)) \mapsto (\texttt{grow}, (1, \Sigma_m))\}$.

R says that the first output of `ignite` is connected to the first input of `burn`, the second output of `burn` is fed back into the first input of `grow` (because it is in F) and so on.

To complete the definition, a proof that no element is connected to a previous one is needed. First, construct the set $P = \{(\texttt{ignite}, \texttt{burn}), (\texttt{grow}, \texttt{burn})\}$ by going through each element of $R \setminus F$ and removing the input/output indices. Then, the transitive closure $P^+ = P$. P^+ is anti-reflexive and so M is a valid cellular system.

4.1 Abstraction and refinement

In order for the user to have the ability to compose more complex effects, it is mandatory to build up larger systems out of smaller subsystems. To achieve this, a CS can act as a block:

Definition 4.1.1. Let $C = (U, B, \Lambda, R, F)$ be a CS. Then, the block (C, I, O) is C *as a block*, where:

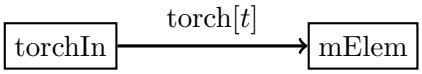
- $I = \{((u, i), \Sigma_i) \mid u \in U,\ (i, \Sigma_i) \in \text{inputs of } \Lambda(u),\ (u, (i, \Sigma_i)) \notin img(R)\}$[3]
- $O = \{((u, o), \Sigma_o) \mid u \in U,\ (o, \Sigma_o) \in \text{outputs of } \Lambda(u),\ (u, (o, \Sigma_o)) \notin dom(R)\}$[4]

In other words, a CS as a block has inputs that correspond to the unused inputs of its internal blocks. The outputs correspond to the unused outputs of internal blocks. The input/output is identified by the name of the internal element and the name of the input/output of its assigned block. If a CS has unused inputs or outputs, then it is *open*, otherwise it is *closed*.

Definition 4.1.2. Let $C = (U, B, \Lambda, R, F)$ be a CS. C is *closed* iff R is bijective, otherwise it is *open*.

The block corresponding to the example system M, as defined in the above section, would be $S = (M, \{((\texttt{ignite}, 2), \{\square, \boxdot\})\}, \emptyset)$. M as a block has an input that links to the second input (unused) of its internal `ignite` element, allowing it to be embedded inside another CS:

[3] $img(f) = \{y \mid \exists x : f(x) = y\}$ denotes the values function f can produce (the image).
[4] $dom(f) = \{x \mid \exists y : f(x) = y\}$ denotes the values function f is defined for (the domain of definition).



Formally, this CS would be written as

$$(\{\texttt{torchIn}, \texttt{mElem}\}, \{R, S\}, \{\texttt{torchIn} \mapsto R, \texttt{mElem} \mapsto S\},$$
$$\{(\texttt{torchIn}, (1, \{\square, \boxdot\})) \mapsto (\texttt{mElem}, ((\texttt{ignite}, 2), \{\square, \boxdot\}))\}, \emptyset)$$

where R is a block sensing where the torch is (explained later in section 4.4).

Definition 4.1.3. Let $C_1 = (U_1, B_1, \Lambda_1, R_1, F_1)$, $C_2 = (U_2, B_2, \Lambda_2, R_2, F_2)$ be cellular systems. C_1 is an *abstraction* of C_2 and C_2 is a *refinement* of C_1, denoted as $C_1 \preceq C_2$ iff

$$\begin{aligned}
&\exists u \in U_1 : \Lambda_1(u) = (C, I, O), \text{ such that } C = (U, B, \Lambda, R, F) \text{ is a CS and } U_1 \cap U = \emptyset \\
&\quad \wedge U_2 = U_1 \setminus \{u\} \cup U \\
&\quad \wedge B_2 = B_1 \cup B \\
&\quad \wedge \Lambda_2 = \Lambda_1 \setminus \{u \mapsto b \mid b \in B_1\} \cup \Lambda \\
&\quad \wedge R_2 = R_1 \setminus \{(u, y) \mapsto (v, x) \mid y \in \text{outputs of } \Lambda_1(u),\ v \in U_1,\ x \in \text{inputs of } \Lambda_1(v)\} \\
&\qquad\qquad \setminus \{(v, y) \mapsto (u, x) \mid x \in \text{inputs of } \Lambda_1(u),\ v \in U_1,\ y \in \text{outputs of } \Lambda_1(v)\} \\
&\qquad\qquad \cup R \\
&\qquad\qquad \cup \{(j, y) \mapsto (k, x) \\
&\qquad\qquad\quad \mid j, k \in U_2, \\
&\qquad\qquad\quad\ y \in \text{outputs of } \Lambda_2(j),\ x \in \text{inputs of } \Lambda_2(k), \\
&\qquad\qquad\quad\ (\exists o \in O : o = ((j, n_o), \Sigma_o) \wedge y = (n_o, \Sigma_o) \wedge R_1((u, o)) = (k, x)) \\
&\qquad\qquad\quad\ \text{or } (\exists i \in I : i = ((k, n_i), \Sigma_i) \wedge x = (n_i, \Sigma_i) \wedge R_1((j, y)) = (u, i)) \\
&\qquad\qquad\quad\ \text{or } (\exists i \in I\, \exists o \in O : i = ((k, n_i), \Sigma_i) \wedge o = ((j, n_o), \Sigma_o) \\
&\qquad\qquad\qquad\qquad\qquad \wedge y = (n_o, \Sigma_o) \wedge x = (n_i, \Sigma_i) \\
&\qquad\qquad\qquad\qquad\qquad \wedge R_1((u, o)) = (u, i))\} \\
&\quad \wedge F_2 = (F_1 \cap R_2) \cup F \\
&\qquad\qquad \cup \{(j, y) \mapsto (k, x) \in R_2 \setminus R_1 \setminus R \\
&\qquad\qquad\quad \mid (j, y) \in dom(F_1) \vee (k, x) \in img(F_1) \vee (j \in U \wedge k \in U)\}
\end{aligned}$$

Figure 4.2 shows what refining a CS looks like. Essentially, to obtain a refinement of C_1 means to „expand“ or „unfold“ one of its elements that contains a CS:

1. Pick one of its elements u whose block contains a CS C,
2. Disconnect u from the other elements in C_1,
3. Replace u with the elements of C, keeping the blocks assigned to the elements,
4. Add all the connections R and feedback connections F from C to C_1,
5. Connect any two elements, such that one both of the elements correspond to an input/output of u (C as a block) and that output/input was connected to the other element in C_1,

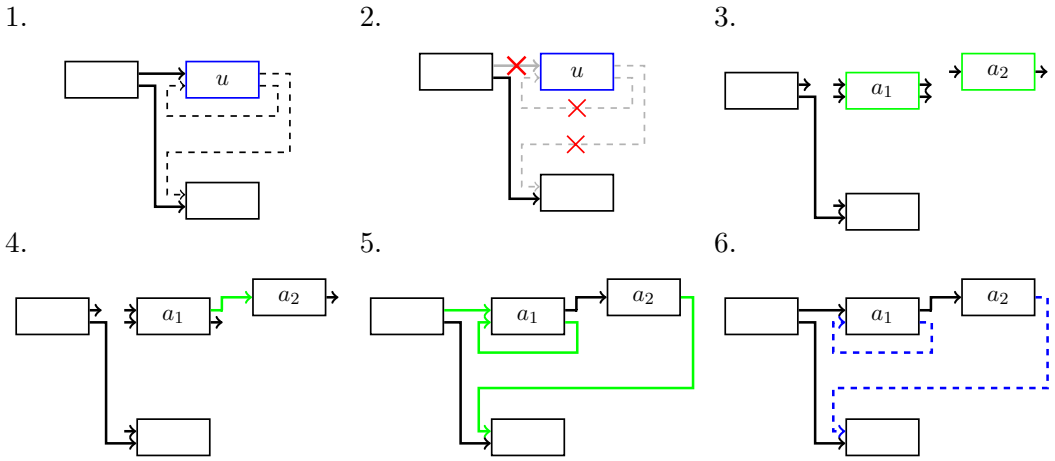


Figure 4.2: Refining a cellular system

6. Finally, consider the connections added in the previous step ($R_2 \setminus R_1 \setminus R$). Of those connections, mark a connection as a feedback if one of its endpoints took part in a feedback connection originally. Also mark those connections that connect a newly added element to another newly added one, since that corresponds to a connection from u to u in C_1.

4.2 Feedback loops and configuration

This section starts describing the actual semantics of a CS. The feedback connections in a CS work by introducing a one-step delay between the output of one block and the input of another. A block that outputs to a feedback connection stores a picture there for the next step.

A CS „stores“ its configuration in its feedback connections. A configuration of a CS is a function that assigns a picture over the appropriate alphabet to each feedback connection.

Definition 4.2.1. Let $C = (U, B, \Lambda, R, F)$ be a CS without a refinement. A *configuration* of C is any total function

$$\chi : F \to \bigcup_{f \in F} \Sigma_f^{**}$$

such that $\forall f \in F : \chi(f) \in \Sigma_f^{**}$, where Σ_f is the alphabet associated with the input/output of blocks that f connects, and all pictures in $img(\chi)$ are of the same size.

It is possible for a CS to have no feedback connections. Such a system has only one possible configuration, which assigns no pictures to nothing (an empty function).

Likewise, a CS can have *only* feedback connections ($R = F$). This is useful for storing a history of configurations and implementing time delays.

4.3 Step and run

Only a closed CS is executable. The compiler (chapter 6) will refuse to compile a CS that is open. Before running the system, it also has to be fully refined, so that the CS only contains elementary rules as blocks.

The operation of a CS that meets these conditions is simple. Consider a CS $C = (U, B, \Lambda, R, F)$. The elements of C execute one by one, according to the topological ordering of the *execution order graph* of C.

Definition 4.3.1. Let $C = (U, B, \Lambda, R, F)$ be a CS. The *execution order graph* of C is the directed acyclic graph $O = (U, P)$, where $P = \{(u, v) \mid ((u, y), (v, x)) \in R \setminus F, \text{ for some } y, x\}$.

An element's input can either come from a regular connection or a feedback connection. Inputs from regular connections are provided by the preceding elements and inputs from feedback are provided by the current configuration χ_1. The first element to execute in a step only has feedback as input.

Definition 4.3.2. Let C be a CS. Let χ_1 and χ_2 be configurations of C. C makes a *step from* χ_1 *to* χ_2, written as $\chi_1 \vdash \chi_2$ iff χ_2 can be constructed using the following algorithm:

Input : Cellular system $C = (U, B, \Lambda, R, F)$, configuration χ_1.
Output: Configuration χ_2.

$\chi_2 \leftarrow \emptyset$
$\phi \leftarrow \emptyset$ `// A function that assigns pictures to` ***`non-feedback`*** `connections`
while *there is a next element to evaluate* **do**
  $u \leftarrow$ the next element from U to evaluate
  Let (S, N, T, B) be the elementary rule of $\Lambda(u)$, where $N = ((y_1, x_1), (y_2, x_2), \ldots, (y_n, x_n))$.
  Let p_1, p_2, …, p_m be the input pictures, where for each of the input connections c_1, c_2, …, c_m of u: $p_i = (\chi_1 \cup \phi)(c_i)$.
  foreach *transition function* $\delta \in T$ **do**
    Construct the picture r of the same size as p_1 symbol by symbol, such that

$$
\begin{aligned}
r(i,j) = \delta(\;& p_1(i+y_1, j+x_1),\ p_1(i+y_2, j+x_2),\ \ldots,\ p_1(i+y_n, j+x_n),\\
& p_2(i+y_1, j+x_1),\ p_2(i+y_2, j+x_2),\ \ldots,\ p_2(i+y_n, j+x_n),\\
& \vdots\\
& p_m(i+y_1, j+x_1),\ p_m(i+y_2, j+x_2),\ \ldots,\ p_m(i+y_n, j+x_n)\\
)&
\end{aligned}
$$

    If a position is outside the bounds of an input picture, interpret the missing symbol according to the boundary condition B as described in Definition 2.3.3.
    $o \leftarrow$ the output connection of u corresponding to δ
    if $o \in F$ **then**
      $\chi_2(o) \leftarrow r$
    else
      $\phi(o) \leftarrow r$
return χ_2

Definition 4.3.3. A *run* (or a *computation*) of CS C beginning from configuration χ_0 is an infinite sequence of its configurations χ_0, χ_1, χ_2, …, where $\chi_i \vdash \chi_{i+1}$ for all $i \geq 0$.

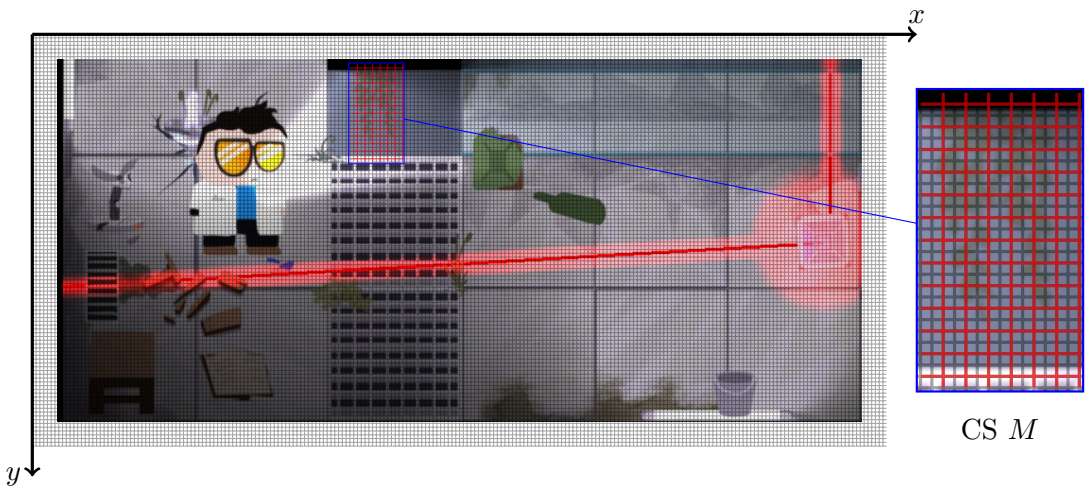


Figure 4.3: A CS embedded in a game scene

4.4 Interactions with the target platform

This section investigates how a CS is actually used. This thesis focuses on applications in art and videogames, but a CS can be used for any simulation involving cellular automata in general. The language compiler is built to support multiple target platforms, so apart from videogames, the CS defined by the language could run in a built-in simulator, be exported to C++ and so on. The rest of this section will focus on videogames only, describing the precise way in which a CS is used in a 2D game. That is, what it means to *embed a CS-based simulation within a game world*.

The game scene

A game is always played in some type of *space*. Most modern videogames use three-dimensional space, the rest use two-dimensional space. Higher-dimensional spaces are also sometimes used, but usually only to show off the technology.

Each game engine works differently, but in general, the *game scene* represents the game space and the objects within it. *Game objects* are usually of varying types. There are graphical objects that render to the screen, such as sprites or meshes. There are also physics objects that represent bodies and solid obstacles, such as characters, walls, moving platforms or vehicles. These objects have a position and a size in the game space, together making up the entirety of the player's experience.

Varying measurement units are used for positions and sizes. The Godot Game Engine simply expresses coordinates using pixels, while Unity uses an arbitrary „world unit“. Importantly, it is up to the user to decide the scale of their objects. For example in the game this thesis uses for testing, 1 meter corresponds to 64 pixels and doors are 128 pixels – or 2 meters – wide.

In the context of a game scene, a CS is simply another type of game object, with a position occupying a predetermined rectangular region of the scene. It is able to react to surrounding objects and also affect them in turn. This allows the user to embed a 2CA-based simulation into their game.

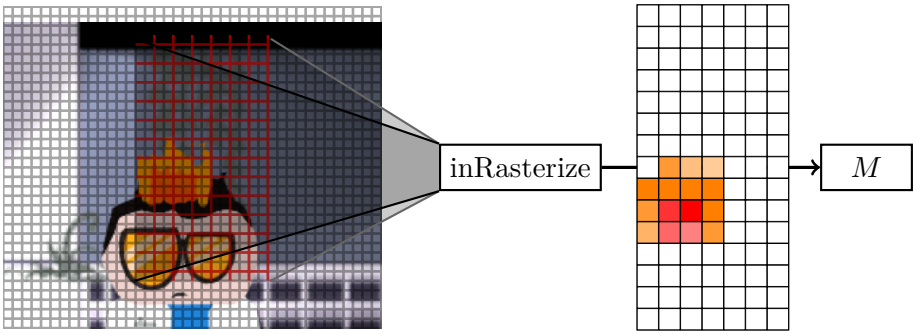


Figure 4.4: Behavior of the `inRasterize` block

Figure 4.3 illustrates the situation. It shows a scene along with a few objects, such as the character on the left and the mirror reflecting a laser on the right. We will focus on the highlighted wall, which accommodates our example moss-growing CS M defined earlier.

The user first programs the behavior of the CS using the provided language, exports it as an object and then can add it to the scene at the desired position. The size of the CS is set in advance by the user. Importantly, the size of the configurations of the CS (the size of the grid in cells) is separate from the space it takes up in the scene.

The gray grid in the figure shows the scene's pixel coordinate grid[5] and the red grid shows how the cell grid of the CS maps onto it. The size of the CS is 7 cells wide and 14 cells tall, meaning all its configurations will be of that size. However, it occupies a region 14 pixels wide and 28 pixels tall.

Input and output

There are two basic internal objects for blocks – elementary rules and cellular systems. These can be defined by the user. To interact with other game objects, a CS makes use of special internal objects, which aren't user-defined, but instead hard-coded into the compiler.

The first kind are *input blocks* that bring information into the CS from the scene. The `inRasterize` block monitors what game objects intersect the region occupied by the CS and *rasterizes* the intersecting part as a picture at the output. Not all objects are picked up by the block, the user can set a filter.

Figure 4.4 shows how the `inRasterize` block can gather information about objects that are on fire. The block fills the role of the `torchIn` block, whose function was promised will be explained.

A character with their hair on fire walks into the region of the CS. The pixels of the flame overlapping the region are detected by `inRasterize`, which then averages and outputs their color. Other input blocks, such as `inPresence` look merely for the presence of an object, outputting a picture consisting of boolean values. Input blocks gather data about the scene every step and, from the point of view of the CS, usually have no inputs themselves.

Conversely, *output blocks* are meant to affect the surrounding scene. The `outDisplay` block, shown in Figure 4.5, takes in a picture over an alphabet of all possible colors (just a normal image) and displays it in the CS region. Note that neither the configuration of a CS, nor any of its internal signals are visible implicitly. An output block is always required

[5]The grid is not to scale for clarity.

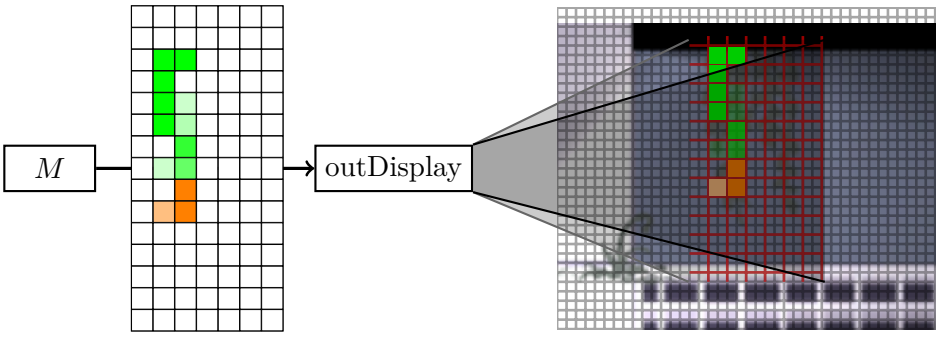


Figure 4.5: Behavior of the `outDisplay` block

to get any kind of visual output. Similarly to input blocks, output blocks execute their effect every step and have no outputs of their own.

For other platforms, special blocks of this kind can be defined in a similar manner.

Limitations

It is also important to discuss the limitations of a CS within a game, in order for the language design to mitigate them. There are several aspects that can significantly affect the visual quality of an effect produced by the CS, or even make it impossible to practically model certain effects, even though a cellular automaton would be ideal in different circumstances.

First is the number of steps the CS makes in a single second. In a game, the frames per second (FPS) greatly affect how smooth it feels to play. This is largely a matter of personal preference, but generally around 60 to 100 frames per second make for a good experience.

Since a CS runs in real time, it is subject to this condition too. For an effect that runs continuously, at least 60 steps per second is needed. For slower effects, such as the example used in this chapter, a slower rate is fine.

Fragment shaders, which can be used to implement a CS, execute once per game frame. Depending on the hardware, a game can run at 50, 100 or 500 FPS. Depending on the implementation, the steps per second are limited by a multiple of the FPS. It would be inappropriate to run a step each game frame, since that would tie the speed of the effect to the FPS. To keep the steps per second constant, some frames need to be skipped.

The second aspect to consider relates to the speed of light of the CS, which is the theoretical maximum number of cells a pattern can travel in a single step. Multiplying the speed of light by the steps per second yields the speed limit in cells per second.

As mentioned before, each cell of the CS can correspond to multiple pixels in the scene and each pixel translates to a length in meters. As such it possible to calculate the speed of light of the CS in meters per second, which we will call the *real speed of light*.

Definition 4.4.1. The *real speed of light* of a CS, denoted as c_r, is the theoretical maximum distance in meters any pattern can travel in one second. It is given by:

$$c_r = \frac{c \cdot \text{steps per second} \cdot \text{pixels per cell}}{\text{pixels per meter}}$$

If a CS runs at 60 steps per second, with $c = 1$, 1 pixel per cell and 64 pixels per meter, then the resulting limit is $c_r = \frac{1 \cdot 60 \cdot 1}{64} = 0.9375\ m/s$. This isn't particularly useful

for modeling, say, an explosion, which propagates at roughly 340 m/s, the speed of sound in air. Therefore, a CS is more suitable for subtler, slower effects.

4.5 Equivalence to cellular automata

Cellular systems are meant to ease the process of constructing 2CA, not surpass them in power. This is similar to multi-tape Turing machines compared to normal Turing machines. This section will prove that it is possible to convert between 2CA and CS, such that their behavior is equivalent. The explained conversion techniques will form the core of the compiler's code generator.

Section 2.4 and the beginning of this chapter claim, that any 2CA can be converted to a CS. Intuitively, perhaps, this claim makes sense. The two models „behave the same“, so they are equivalent. However, no formal criterion of what „behaving the same“ means has been defined.

One such definition could be that if both models start from the same configuration, then they will have *the same runs*, performing the same steps. That is, the corresponding configurations in their runs are equivalent and the run of one model can be used to obtain the run of the other model. As such, there isn't a run that the other model isn't able to mimic. But, the configuration of a 2CA is just a picture and the configuration of a CS is a function producing multiple pictures, so comparing them directly is not possible. It is possible, however, to establish a bijection between the possible configurations of the two models, pairing together equivalent configurations.

A problem arises with regards to the boundary conditions. When converting a CS to a 2CA, how does one emulate the different boundary conditions used by each block with just a single condition permitted by the 2CA? A solution is to simply limit the conversion to those cases where all blocks of the CS use the same condition. A CS making use of multiple boundary conditions is hard to reason about, both for the user and for the compiler. Therefore, the language will forbid setting boundary conditions for rules and open CS altogether and permit the setting only for a closed CS, where the condition will be propagated to all contained rules.

Theorem 4.5.1. Any given closed CS S, whose rules all share the same boundary condition, can be converted to a 2CA C and vice versa, such that there exists a total bijective function $\sigma : X \leftrightarrow P$, where X and P are the sets of all possible configurations of S and C respectively and σ fulfills the following condition.

Let $s_0 \vdash s_1 \vdash s_2 \vdash \dots$ be a run of C and $\chi_0 \vdash \chi_1 \vdash \chi_2 \vdash \dots$ be a run of S. If $s_0 = \sigma(\chi_0)$, then $s_1 = \sigma(\chi_1)$, $s_2 = \sigma(\chi_2)$, ...

Proof. Converting a 2CA to a CS. Let $C = (\Sigma, N, \delta, B)$ be a 2CA. First we will convert C to a CS S using the same method as used at the start of this chapter. Then, we will find σ and prove by induction it can be used to convert between runs of C and S.

Construct the corresponding elementary rule $E = ((\Sigma), N, (\delta), B)$. Then, E as a block is the tuple $E_B = (E, \{(1, \Sigma)\}, \{(1, \Sigma)\})$. Construct the supposedly equivalent CS $S = (\{e\}, \{E_B\}, \{e \mapsto E_B\}, \{f\}, \{f\})$, where $f = (e, (1, \Sigma)) \mapsto (e, (1, \Sigma))$. S is clearly closed and contains only elementary rules as blocks sharing the same boundary condition.

Now, we will find the function σ. Since S essentially uses the function δ of C to update the picture assigned to f, it would appear that $\sigma(\chi) = \chi(f)$ for any configuration χ of S.

First we prove that σ is a total bijection by proving that it is total and its inverse is total. σ is total, because any given χ must accept f and only f. For any configuration p of C, the

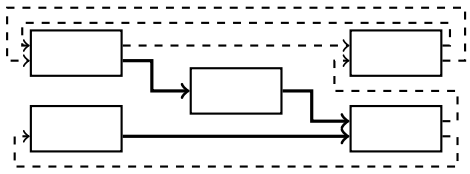
inverse function is $\sigma^{-1}(p) = \{f \mapsto p\}$, since $\sigma^{-1}(\sigma(\chi)) = \sigma^{-1}(\chi(f)) = \{f \mapsto \chi(f)\} = \chi$. It is also total, since $p \in \Sigma^{**}$ and Σ corresponds to f, so p can always be assigned to f.

Second we prove by induction that σ can indeed be used to convert between runs of S and C. Let $s_0 \vdash s_1 \vdash \dots$ be a run of C and let $\chi_0 \vdash \chi_1 \vdash \dots$ be a run of S. Assume that $s_0 = \sigma(\chi_0)$, forming the base case. If $s_0 \neq \sigma(\chi_0)$, the two runs are unrelated.

For the induction step we prove $s_i = \sigma(\chi_i) \implies s_{i+1} = \sigma(\chi_{i+1})$ for all $i \geq 0$. We can rewrite this as $s_i = \chi_i(f) \implies s_{i+1} = \chi_{i+1}(f)$. The configuration s_{i+1} is obtained from s_i by constructing it symbol by symbol using the function δ according to Definition 2.3.5. Since S is so simple, the step algorithm from Definition 4.3.2 simplifies too. There is only one element e, it only has the one input f and one transition function δ. Only one resulting picture is constructed using $\chi_i(f)$ and in the same manner as for normal 2CA. The algorithm boils down to a single assignment of this resulting picture to $\chi_{i+1}(f)$. Per the induction hypothesis, s_i and $\chi_i(f)$ are the same picture and since the same procedure is applied to both to obtain s_{i+1} and $\chi_{i+1}(f)$, the new pictures are also equal. This completes the induction step, proving that σ can convert between the runs. C and S behave the same.

Converting a CS to a 2CA. This is a somewhat more complex process, which involves gradually merging all the blocks and elements of the CS into one element, easily convertible to a 2CA by reversing the procedure above. The conversion will happen in six stages, each stage bringing the workings of the CS closer to a traditional 2CA.

Let $S_1 = (U, K, \Lambda, R, F)$ be a CS. First, we will merge all elements of S_1 that can execute simultaneously (the ones in the same layer of the layering of S_1), so that we are left with only elements connected linearly in series. The parallel elements will be merged by simply moving all their transition functions into a new block, assigning that block to a new element and wiring back the connections. Below is an example of an S_1:



Let $L_1, L_2, \dots, L_h$ be the layering of the execution order graph of S_1. We will construct a sequence of h elements $l_1, l_2, \dots, l_h$ and blocks $B_1, B_2, \dots, B_h$ to later connect in series. To obtain the i-th block B_i, containing the elementary rule E_i, take the set $L_i = \{e_1, e_2, \dots, e_m\}$ and merge the rules $(A_1, N_1, T_1, C), (A_2, N_2, T_2, C), \dots, (A_m, N_m, T_m, C)$ of blocks $\Lambda(e_1), \Lambda(e_2), \dots, \Lambda(e_m)$. The result is $E_i = (A, N, T, C)$. Its components are as follows.

The alphabet tuple A is a simple concatenation of the tuples $A_1, \dots, A_m$. That is, let $A_j = (\Sigma_1^j, \Sigma_2^j, \dots, \Sigma_{t_j}^j)$. Then $A = (\Sigma_1^1, \Sigma_2^1, \dots, \Sigma_{t_1}^1, \Sigma_1^2, \Sigma_2^2, \dots, \Sigma_{t_2}^2, \dots, \Sigma_{t_m}^m)$.

The neighborhood N is the *smallest neighborhood*, which contains all the neighbor positions from $N_1, \dots, N_m$ plus the neighbor $(0, 0)$, which will be required for adding identity functions to rules in later stages. It is a concatenation of $((0, 0)), N_1, \dots, N_m$, with only the first duplicate neighbor kept. For example, given $N_1 = ((1, 1), (1, 2))$ and $N_2 = ((1, 1), (0, 1))$, $N = ((0, 0), (1, 1), (1, 2), (0, 1))$.

The tuple T is made by transforming the transition functions from $T_1, \dots, T_m$, so that they take the new neighbors as inputs and ignore them. Let $T_1 = (\delta_1^1, \delta_2^1, \dots, \delta_{t_1}^1)$, $T_2 = (\delta_1^2, \delta_2^2, \dots, \delta_{t_2}^2), \dots, T_m = (\delta_1^m, \delta_2^m, \dots, \delta_{t_m}^m)$.

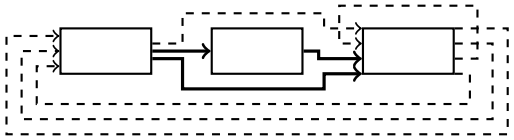
Then T is defined as $T = (\Delta_1^1, \Delta_2^1, \dots, \Delta_{t_1}^1, \Delta_1^2, \Delta_2^2, \dots, \Delta_{t_2}^2, \Delta_1^3, \dots, \Delta_{t_m}^m)$, where Δ_j^k behaves the same as δ_j^k, but ignores the new parameters: those corresponding to the input

connections of the other elements in this layer and those corresponding to the new neighbors in N, which aren't a part of N_k. Finally, the boundary condition C is the same for all the original rules and is used in E_i too.

Now, we are left with the rules E_1, $E_2, \ldots, E_h$. To obtain the entire block B_i with its inputs and outputs, we simply take E_i *as a block*, per Definition 4.0.3. To reconnect the elements $l_1, \ldots, l_h$, we construct the new set of connections R' and feedback connections F'.

R' is made by transforming each connection $(u, (y, \Sigma)) \mapsto (v, (x, \Sigma)) \in R$, where $u \in L_j$ and $v \in L_k$, into $(l_j, (x + n_x, \Sigma)) \mapsto (l_k, (y + n_y, \Sigma))$, where n_x (n_y) is the total combined number of alphabets of the elements, which came before u (v) in the layer L_j (L_k) during the merge. Since R is bijective and the merge preserves the total number of inputs and outputs (the block B_i has $t_1 + t_2 + \cdots + t_m$ inputs and outputs), we can afford this. F' is obtained in the exact same manner.

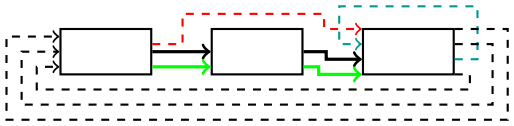
The CS $S_2 = (\{l_1, l_2, \ldots, l_h\}, \{B_1, B_2, \ldots, B_h\}, \{l_1 \mapsto B_1, \ldots, l_h \mapsto B_h\}, R', F')$. The system S_2 has an important property required by the next stage. Its execution graph has the layering $\{l_1\}, \{l_2\}, \ldots, \{l_h\}$, which means that $(l_j, y) \mapsto (l_k, x) \in R' \setminus F' \implies k > j$. Below is an example of an S_2, obtained by applying the procedure to the previous example:



In the second stage, we will ensure that each element is connected only to the very next one via non-feedback connections. That is, we will construct new connections R'', such that $(l_j, y) \mapsto (l_k, x) \in R'' \setminus F' \implies k = j + 1$. This makes the next stages simpler.

For each connection $(l_j, y) \mapsto (l_k, x) \in R' \setminus F'$, where $k > j+1$, we will remove the connection and „pass it through“ the elements l_{j+1}, $l_{j+2}, \ldots, l_{k-1}$ by appending an identity function to the transition functions of the blocks B_{j+1}, $B_{j+2}, \ldots, B_{k-1}$ and also appending the appropriate alphabet corresponding to the removed connection. Let x_{j+1}, $x_{j+2}, \ldots, x_{k-1}$ be the newly introduced inputs and y_{j+1}, $y_{j+2}, \ldots, y_{k-1}$ the new outputs. Now, add the connections $(l_j, y) \mapsto (l_{j+1}, x_{j+1})$, $(l_{j+1}, y_{j+1}) \mapsto (l_{j+2}, x_{j+2}), \ldots, (l_{k-1}, y_{k-1}) \mapsto (l_k, x)$.

Let B'_1, $B'_2, \ldots, B'_h$ be the new blocks obtained during the process. The resulting CS is $S_3 = (\{l_1, l_2, \ldots, l_h\}, \{B'_1, \ldots, B'_h\}, \{l_1 \mapsto B'_1, \ldots, l_h \mapsto B'_h\}, R'', F')$. The following is obtained from the previous example:

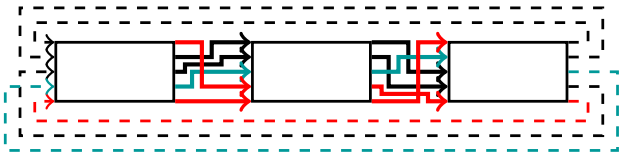


Stage three handles the feedback connections. Right now, any given feedback connection $f = (l_j, y) \mapsto (l_k, x)$ with a corresponding alphabet Σ can exist between any two elements of the chain, so either $j \geq k$ or $j < k$. We would like to move all feedback connections, so that $j = h$ and $k = 1$, connecting the last and first element, bringing us closer to having just one feedback connection in the whole system. The idea is that l_1 can pass the picture $\chi_t(f)$ through any intermediate elements to l_k and l_j can pass the updated picture $\chi_{t+1}(f)$ to l_h.

First, remove f. Add an identity function and Σ to the blocks B'_1, $B'_2, \ldots, B'_{k-1}$. Let x_1, $x_2, \ldots, x_{k-1}$ be the new inputs and y_1, $y_2, \ldots, y_{k-1}$ the new outputs. Add the con-

nections $(l_1, y_1) \mapsto (l_2, x_2), \ldots, (l_{k-1}, y_{k-1}) \mapsto (l_k, x)$. This is the path through which l_k will receive $\chi_t(f)$ from l_1. Then, add an identity function and Σ to B'_{j+1}, $B'_{j+2}, \ldots, B'_h$. Let x'_{j+1}, $x'_{j+2}, \ldots, x'_h$ be the new inputs and y'_{j+1}, $y'_{j+2}, \ldots, y'_h$ the new outputs. Add the connections $(l_j, y) \mapsto (l_{j+1}, x'_{j+1}), \ldots, (l_{h-1}, y'_{h-1}) \mapsto (l_h, x'_h)$, which will pass $\chi_{t+1}(f)$ from l_j to l_h. Finally, add the feedback connection $(l_h, y'_h) \mapsto (l_1, x_1)$. If $l_j = l_1$ ($l_k = l_h$), then use $x_1 = x$ ($y'_h = y$).

The result is CS $S_4 = (\{l_1, l_2, \ldots, l_h\}, \{B''_1, \ldots, B''_h\}, \{l_1 \mapsto B''_1, \ldots, l_h \mapsto B''_h\}, R''', F'')$. Below is an example of how to transform the red and blue feedback connections from the example above:

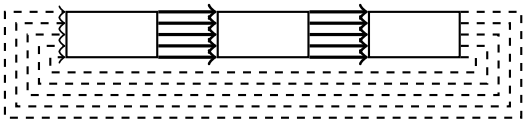


At this point feedback connections exist only between l_1 and l_h and non-feedback connections only between l_i and l_{i+1}. All blocks have the same number of inputs and outputs. Stage four makes it possible to merge all connections between two elements into a single connection, carrying the entire state. For this, we need to ensure that all blocks share the same alphabet tuple and $(l_j, y) \mapsto (l_k, x) \in R'''' \implies x = y$, so that input values are not swapped. We can use the block B''_i to reorder the alphabets of B''_{i+1}, starting with $i = 1$.

Given a connection $c = (l_i, y) \mapsto (l_{i+1}, x)$, remove c from R'''. Reorder the alphabets of B''_{i+1}, such that if y is the a-th output of B''_i and x is the b-th input of B''_{i+1}, we move the b-th alphabet of B''_{i+1} so that its the a-th alphabet. Also reorder the transition functions and inputs/outputs.

Given a feedback connection $f = (l_h, y) \mapsto (l_1, x)$, we remove f, add $(l_h, y) \mapsto (l_1, y)$ and remap the parameters of the transition functions of B''_1 to correspond to the input y instead of x.

This gives the CS $S_5 = (\{l_1, l_2, \ldots, l_h\}, \{B'''_1, \ldots, B'''_h\}, \{l_1 \mapsto B'''_1, \ldots, l_h \mapsto B'''_h\}, R'''', F'''')$.



In stage five, we can assume that $A'''_1 = A'''_2 = \cdots = A'''_h = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$, where A'''_k is the alphabet tuple of B'''_k. Replace each tuple with (Σ), where $\Sigma = \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n$. Given the transition function tuple $(\delta_1, \delta_2, \ldots, \delta_n)$ of a block, replace it with (Δ), where:

$$
\begin{aligned}
\Delta(x_1, x_2, \ldots, x_m) = (\\
&\delta_1(u^1_1, u^1_2, \ldots, u^1_m, u^2_1, u^2_2, \ldots, u^2_m, \ldots, u^n_m),\\
&\delta_2(u^1_1, u^1_2, \ldots, u^1_m, u^2_1, u^2_2, \ldots, u^2_m, \ldots, u^n_m),\\
&\vdots\\
&\delta_n(u^1_1, u^1_2, \ldots, u^1_m, u^2_1, u^2_2, \ldots, u^2_m, \ldots, u^n_m)\\
&)\\
x_k = (u^1_k, u^2_k, \ldots, u^n_k)
\end{aligned}
$$

Also replace all connections between elements with a single connection, since each block now only has one input and output.

In the sixth stage, we merge the entire chain into a single block. We can merge two blocks B_i'''' and B_{i+1}'''' by creating a new neighborhood N, that contains the neighborhood of B_{i+1}'''', but also the neighbors obtained by overlaying the neighborhood of B_i'''' over each neighbor of B_{i+1}''''. The merged block has the same alphabet and boundary condition as the two blocks. The new transition function Δ works by simulating what B_i'''' would have done to the neighborhood of B_{i+1}'''' and then applying the transition function of B_{i+1}'''' as usual. We can construct the final 2CA as $M = (\Sigma, N, \Delta, C)$.

The bijection σ works by simply stacking the pictures produced by a configuration χ of S_1 on top of each other. To determine the order of stacking, consider stage three of the conversion. Each original feedback connection is moved so that it connects the k-th output of l_h to l_1. Let f_1, $f_2, \ldots,$ f_n be the ordered connections, where f_1 is the connection assigned to the first output, f_2 the one assigned to the second and so on. Then $\sigma(\chi)$ is a picture, assembled as follows:

$$\sigma(\chi)(i,j) = (\chi(f_1)(i,j),\, \chi(f_2)(i,j), \ldots,\, \chi(f_n)(i,j))$$

The function σ is total, because any χ is always defined for $f_1, \ldots, f_n$. Its inverse $\sigma^{-1}(p)$ is defined as:

$$\sigma^{-1}(p) = \{f_1 \mapsto p_1,\, f_2 \mapsto p_2, \ldots,\, f_n \mapsto p_n\}$$

where each symbol $p_k(i,j)$ is the k-th component of the tuple $p(i,j)$. The inverse is also total, since $p(i,j)$ is always an n-tuple, where the k-th component always exists. We prove by induction that σ establishes a correspondence between runs of S_1 and M.

Let $\chi_0 \vdash \chi_1 \vdash \ldots$ be a run of S_1 and $s_0 \vdash s_1 \vdash \ldots$ a run of M. Assume $s_0 = \sigma(\chi_0)$ as the base case and prove that $s_i = \sigma(\chi_i) \implies s_{i+1} = \sigma(\chi_{i+1})$. First, note that S_1 and S_5 behave the same, as we are merely reordering connections and filling the gaps with identity functions. Therefore, any configuration (run) of S_1 is also a configuration (run) of S_5, providing we take into account the reassigned feedback. Second, the function σ can be used to convert the run of S_5 to a run of the CS produced in stage five and six by converting χ_i to $\{f \mapsto \sigma(\chi_i)\}$, where f is the single feedback connection of the final CS. Finally, M is obtained using the same procedure as when converting a 2CA to a CS, but reversed. The first part of this proof proves that σ can convert between runs of the final CS and the 2CA M. □

Chapter 5

The CSL language

This chapter introduces a new language, called the Cellular System Language (CSL), that is used to define the cellular systems presented in the previous chapter. This chapter provides the design philosophy behind the language and introduces a few basic constructs from the perspective of the user. For more comprehensive documentation, see Appendix B.

Section 5.1 explores the various metaphors the language uses, explaining the terms the user thinks in during programming and how these translate to defining cellular systems. Section 5.2 gives a tour of the language.

5.1 Metaphors

As alluded to earlier, the idea for the new variant of 2CA came from ancient Greek Stoic physical theories. While wildly inaccurate today, their simplified model of the world works for the stylized universe that runs in a 2CA. This section examines how useful of a metaphor it could be for the language itself, that is, whether it is something the user can think in.

The Stoic philosophers divided the universe into two parts, the active and the passive principles. The passive principle represents matter, lying dormant until acted on by the active principle [4].

The Active Principle (The Universal Reason)

The equivalent expression in today's terms would be „the laws of physics". It referred to the driving forces of the world. Back then, it was believed that it was the various Greek gods that drove its many aspects. Helios pulled the sun along using his fiery chariot. Poseidon commanded the waters and seas with his trident. In this way, the entire pantheon would take turns painting the canvas of the universe, guiding and shaping its material substrate. Presiding over all these deities was Zeus.

A cellular system is an analog to this „pantheon of gods", each block contributing to the overall picture.

The Passive Principle (Substance)

The passive principle was referred to as „primary matter (οὐσια)" by the Greek philosopher Zeno and later on as „essentia" by Seneca, a Roman. The English translation is „substance" or „essence". It could be equated to physical matter, fields and also information that surround us. All of these would have no behavior of their own, if it weren't for the laws of physics.

The configuration of a CS is an analog of this passive principle, representing a snapshot of the world.

It would perhaps be grossly inappropriate to use this type of religious metaphor directly. People are unfamiliar with the jargon and, in a technical context, it could be seen as somewhat deranged. It can however provide inspiration for general features of the language as well as its syntax.

5.2 Scripting

To create their simulated effect, the user specifies it in terms of *processes*. This construct, declared using the `process` keyword, specifies the behavior of the „stuff", or the *substance*, that makes up the simulation, determining the transition rule all cells will use. The most basic substances correspond to the basic data types found in any language, such as `Int`, `Float` or `Bool`.

```
process main:
       White <- not White
       outDisplay (boolToColor White)
```

This script will create an effect that alternates between black and white every step. When invoked, the compiler searches for a process called `main` and uses it to construct the main effect. Processes are defined similarly to functions in other languages, but they work very differently. The first line assigns the result of *another process* called `not` to a substance named `White`. Substances are declared implicitly whenever they are used as a parameter for a process and are initialized to a default value. Since `not` expects the `Bool` type, `White` is a `Bool` substance and has the value `false`. The process `outDisplay` displays a `Color` substance and `boolToColor` converts a `Bool` to a `Color`.

The value that a substance had at the end of a step (the end of `main`) is preserved for the start of the next step. Therefore, `White` will keep alternating between `true` and `false`.

```
process main:
       White <- not White[-1,0]
       outDisplay (boolToColor White)

process main2 with reflective boundary:
       ...
```

By using the `[x,y]` operator, the neighbor at the given offset from the current cell can be accessed. Above, `main` will create a series of black and white stripes, growing from the left. An out-of-bounds neighbor's value is determined by the boundary condition, set using `with CONDITION boundary`.

```
process main:
       AliveCount <- (boolToInt Alive)[moore by (+)]
       Alive <- if Alive then (AliveCount == 2 || AliveCount == 3)
                         else (AliveCount == 3)
       outDisplay (boolToColor Alive)
```

Above is an implementation of Life. The `[moore by +]` is a *fold operator* and it can accumulate together results of a binary process applied to a neighborhood. In this case, it

adds together the live cells within the `moore` neighborhood. The `if-then-else` expression works as expected from other languages.

```
process last(In: Bool, Out: Bool):
        Out <- Last[0,0]
        Last <- In[0,0]

process main:
        Trace <- last Alive -- Equivalent to "last Alive Trace"
        life Alive -- declared same as above, but as an open process
        outDisplay (toColor Alive Trace)
```

A process that is declared with parameters is *open* and can act on substances given from the outside. That it can „act on“ means it can *write to* as well as read from its parameters. The script above adds a trace effect to Life.

```
process main:
        mossEffect

process ignite(Torch: Bool, Fire: Bool):
        Fire <- if Torch then True else Fire[0,0]

process grow(Moss: Bool):
        randomBool Rand
        Moss <- Moss[0,0] || (Moss[moore by (||)] && Rand)

process burn(Fire: Bool, Moss: Bool):
        Burnup <- randomBool && Fire
        Moss <- Moss && (not Burnup)
        SpreadFire <- Fire[0,0] || (Fire[moore by (||)])
        Fire <- Moss && SpreadFire

process mossEffect:
        inPresence Torch
        ignite Torch Fire
        grow Moss
        burn Fire Moss
        outDisplay (toColor Fire Moss)
```

This is an implementation of the effect presented in Chapter 4. Note the use of `randomBool` and `inPresence`, processes that bring in input from the game scene (Section 4.4).

Chapter 6

Implementation

This chapter provides an overview of the compiler architecture and the technologies used within it. The compiler operates by accepting a `.csl` file as the input, written in Cellular System Language, and produces a Godot Text Scene File (`.escn`). This file can then be used as a game object within the Godot Game Engine, where it runs the defined effect using a multi-pass fragment shader.

The target platform is Godot version 3.5[1]. As of the time of writing, the latest version is 4.2, but the game used for testing is implemented in the older version. Godot 4.2 supports compute shaders, which make the task of implementing cellular automata vastly simpler, but version 3.5 does not support compute shaders, only fragment shaders.[2]

6.1 Technologies

The language of choice is Haskell. It lends itself well to writing compilers, thanks to the availability of parser libraries, purely functional programming, good pattern matching and monads. Below is a list of libraries used. These are all available on Hackage[3].

megaparsec, parser-combinators
: Megaparsec is a fork of the Parsec library. It is a library for implementing parsers in a modular fashion using monadic combinators, where the final parser is composed of many smaller parsers.

mtl
: MTL is the Monad Transformer Library. It allows composing the effects of multiple monads together, making it easy to combine a stateful computation with one that requires exception handling, for example.

lens
: Lenses are composable getters and setters, allowing one to write functions that „target“ a part of some data structure and then map functions over those parts. For example,

[1] https://docs.godotengine.org/en/3.5/

[2] The difficulty of implementing any but the most simple 2CA using fragment shaders is a significant part of the reason why this language and compiler exist in the first place. Setting up the required nodes and shaders in the Godot Editor is a rather menial task, as described in Section 6.7. Even then, the section gives only a small tasting of the laundry list of details the user needs to take care of to make 2CA function. CSL transforms this task into just a few lines of code.

[3] https://hackage.haskell.org/

targeting all the identifiers in an abstract syntax tree, or all the connection endpoints of a cellular system.

matrix, set-monad, algebraic-graphs
These are libraries providing data structures for matrices, sets and graphs.

6.2 Architecture

The compiler executes in five stages run sequentially. Each stage outputs a data structure that is passed to the next stage.

1. **Parser** – Performs lexical and syntax analysis of a `.csl` file, producing an abstract syntax tree (AST),
2. **Semantic analyzer** – Traverses and analyzes the AST, producing a cellular system,
3. **Cellular system processor** – Transforms the cellular system into a linear sequence of blocks (see Theorem 4.5.1),
4. **Scene graph builder** – Converts the linearized system into a graph representing the Godot implementation of the system,
5. **Scene file generator** – Serializes the scene graph, generating a Godot Text Scene File (TSCN, `.escn`).

This division makes it possible to separate the language from the target platform entirely. It is also possible to swap out the language frontend for a graphical frontend, where a user can define the CS using a graphical editor.

6.3 Parser

The scanner and parser are both implemented in the `CSL.Frontend.Parser` module. Their role is to parse the input file and produce an abstract syntax tree (`CSL.Frontend.AST`). The full grammar is available in Appendix B.

6.4 Semantic analyzer

The semantic analyzer traverses the `AST` and constructs a `CellularSystem` (implemented in the `CSL.Model.CellularSystem` module) as the intermediate representation. The implementation of `CellularSystem` adheres to its definition from Chapter 4. The analyzer is implemented in three analyzer modules (`CSL.Frontend.Semantics.Analyzers`) and three monad modules (`CSL.Frontend.Semantics.Monads`) providing common functions.

The *script analyzer* analyzes top-level items in the script and uses the `ScriptAnalyzer` monad. Definitions of neighborhoods and substances are processed directly by the script analyzer. Definitions of processes are handed off to the process analyzer.

The *process analyzer* constructs a block containing the cellular system defined by a process, using the `ProcessAnalyzer` monad. Each process consists of one or more expressions on separate lines. The analyzer takes these lines one by one and calls the final analyzer, the *expression analyzer*, on them. The expression analyzer returns a fragment of a cellular

system, along with information about how to connect it to the rest of the system being built. This information comes in the form of *signals*.

A *signal* consists of an identifier (such as `Fire`) paired with two markers, one assigned to the input of an element, called the *start marker*, and the other to an output of an element (the same element or a different one), the *end marker*. Whenever a signal identifier is used as an argument of an open process, a signal is created with that identifier. The placement of the start marker and end marker is defined by the parameters of the open process being used.

The fragment defined by an expression is connected to the system being built by connecting the end markers of the already existing signals to the start markers of the signals in the fragment. The exact way the *expression analyzer* assembles system fragments defined by an expression is described in Appendix B.

If the analyzed process is open, its parameters are associated with the signals that share the same identifier and those signals are deleted. Any leftover signals will have a feedback connection added, connecting the end marker and start marker.

In the end, the whole analysis returns the cellular system defined by the process called `main`.

6.5 Cellular system processor

Before the cellular system is sent to the scene builder, it is passed through the `refine`, `mergeSequential` and `linearize` functions in order. These are located in the model, in the module `CSL.Model.CellularSystem`.

1. `refine` – Flattens a system, exactly as described in Section 4.1.

2. `mergeSequential` – Optimizes a system by merging together elements connected in sequence, which do not need to run sequentially. Whenever element A reads only the center neighbor of a picture it receives from element B, A will absorb element B and inline the transition functions of B into its own.

3. `linearize` – Implements the conversion algorithm from the proof of Theorem 4.5.1, except for the last stage, where the chain of blocks is merged into one block.

The output is therefore a series of blocks, which the scene builder considers as just a list of expressions that make up the transition functions. This makes it more convenient to implement the system using a multi-pass shader.

6.6 Scene graph builder

Before describing the scene builder, it's best to introduce how a CS works with the Godot Game Engine, starting with a brief introduction to the basic concepts of Godot 3.5[4]. There are five terms, important to the implementation: *node*, *scene*, *scene tree*, *viewport* and *shader*. In Godot a game is a *tree* of *nodes* that the user can group together into *scenes*. The nodes are rendered using *viewports*. A *shader* can be attached to a node to change the way it is drawn.

[4]For more details, see https://docs.godotengine.org/en/3.5/getting_started/introduction/key_concepts_overview.html.

Scene

A scene is a reusable part of a game. This includes characters, levels, vehicles, user-interface elements, such as buttons and menus, or any other object within the game. Even the game itself is a scene. Scenes can be executed by the engine and are always saved to a file[5].

Node

A scene is a tree of nodes. For example, a character can have the `KinematicBody2D` node as its root. This node can move around the scene and collides with walls and floors. Adding children to this node adds functionality to the character. A `Sprite2D` node attaches a sprite to the character and the `AudioStreamPlayer2D` node can play sounds, such as footsteps.

Scene tree

When the main scene of the game is executed, it is added to the scene tree. As all scenes are trees of nodes, the scene tree is also a tree of nodes.

Viewport

A `Viewport` is a node that renders its children. A scene tree always has a `Viewport` as its root node, rendering the whole game. The user can also add `Viewport` nodes to render anything they want to a texture and optionally display it.

Shader

Shaders can be attached to a node. It is a program that recalculates the color of each pixel of a node. A pixel in the resulting texture consists of four floating point numbers, one for each of the color channels: red, green, blue and alpha. They can also accept *textures rendered by Viewports* as their inputs.

The gist of the CS implementation is as follows. Each element of the linearized system is implemented as one `Viewport` with one `ColorRect` node as its child. A `ColorRect` node is just a colored rectangle. This node will take up the entirety of the `Viewport`'s size and will have a shader attached to it. The shader will *implement the transition function* of the element.

Therefore, the `Viewport` will output a texture corresponding to the picture the element outputs. A cell of the automaton corresponds to a pixel in the texture. This texture will be the input of the next element's shader, which will again produce a texture by rendering to the `Viewport` of that next element.

The `outDisplay` output block renders any texture it receives to the screen. Input blocks such as `inPresence` will simply produce their output texture by rendering whatever `Sprite2D` nodes enter the area.

The problem is that while the state of a cell can be of potentially arbitrary size and format, a pixel can only hold the four floats. Therefore, the back end of the compiler also handles bit packing arbitrary data into the color channels as well as adding additional shaders to handle the data that does not fit in the channels one pixel.

The scene builder constructs a graph, where each node corresponds to one of these `Viewport` nodes. It is implemented in the `CSL.Backend.Godot.SceneGraphBuilder` module and runs the following functions on the linearized system, in sequence.

[5]These files have the `.tscn` extension. Files ending in `.escn` are exactly the same format, but indicate that this file was generated by an external program and the Godot editor should not write to it.

1. `extractExprs` – Extracts the expression tree of each transition function in the linearized system. This produces a sequence of ASTs.

2. `removeTupleConstructors` – The Godot shader language supports neither structs nor tuples. This function replaces any expressions producing tuples with multiple expressions, each producing one component of the tuple.

3. `createBaseGraph` – Transforms the sequence of ASTs into a graph, where each node corresponds to a node in Godot. This graph is a linear path.

4. `eliminateUselessNodes` – Eliminates identity functions left behind by the `linearize` function.

5. `splitRuleNodes` – Splits the nodes in the graph, so that their output fits in one pixel. Please note, this step means that nodes can now *receive their inputs from multiple other nodes, not just one*, as the graph is no longer a path.

The resulting graph structure represents the final scene.

6.7 Scene file generator

The role of the scene generator is to serialize the structural representation of the scene produced by the scene builder into a file.

Any scene created in the Godot Editor is saved in the *TSCN (text scene) file format*[6]. This format is the target language of the generator. Besides a description of the scene tree, a `.tscn` (`.escn`) file can also contain embedded GDScript code, shader code and even binary files like images, meaning it can contain everything necessary for a game object.

A file in the TSCN format consists of four main sections: declaration of external resources, declaration of internal resources, a description of the scene tree and signal connections.

External resources

These are external files that the object depends on, such as scripts or images. These are not used by the code generator, since it only exports a single self-contained file.

Internal resources (subresources)

Internal resources (also called subresources) are the same as external resources, but built into the file. The generator puts shader code and GDScript code in this section.

Scene tree

This section contains all of the nodes in the scene and their parameters. Nodes often contain references to subresources, for example a GDScript subresource attached to a player node. The generator puts the description of `Viewport` nodes here.

Signal connections

Signals are a means of communication between nodes. This section is unused by the generator.

[6] https://docs.godotengine.org/en/3.5/development/file_formats/tscn.html

The file generator resides in `CSL.Backend.Godot.SceneFileGenerator` and uses the `Generator` monad implemented in `CSL.Backend.Godot.SceneFileGeneratorMonad`. The main module handles generating shader code, while the monad generates the surrounding TSCN, in which the shader code is embedded. A file is generated as follows.

1. `runGenerator` – Initializes the `Generator` monad, which contains a table describing the nodes and subresources to be generated. Then, it calls `processNode` on each node in the scene graph.
2. `processNode` – Processes a node (see below) and populates the node and subresource table.
3. Once all nodes are processed, `runGenerator` takes the final table and serializes it, putting the appropriate text in each section of the TSCN file.

To process a node means to generate the shader code that implements it and register the node in the node table:

1. Generate the preamble of the shader, which consists of `uniform`s declaring the texture inputs of this shader and code that unpacks data from the pixels of the inputs. One `uniform sampler2D` declaration is generated for each node read by the processed node. Multiple cells (pixels) can be read from each texture and the code to unpack the data from a pixel is generated by calling the `generateReaders` function.
2. Generate the main body of the shader. The processed node can output multiple values and code is generated for each of them. Each of these values is a function of the unpacked data from the preamble.
3. Generate the output code. By calling the `generateWriters` function, each value produced in the previous step is packed into a color channel. Multiple values can be packed into the same channel. For example, the red channel can fit 8 `Bool` values.

The bitpacking functions, that is `generateReaders` and `generateWriters`, are part of the `CSL.Backend.Godot.BitpackingGenerator` module. These functions work by constructing a `StorageMap`, which is a mapping between the bits of a to-be-packed variable and the bits of the color channel. This is used to calculate the bit masks and bit offsets for the reader/writer code.

Chapter 7

Testing

The generated scene file can be drag-and-dropped into the Godot Editor. The generated object inherits the `Control` class and its size is set in the same way as for any other `Control` node.[1] When the object is selected in Godot, the Inspector panel offers options for input and output processes, such as `inPresence`.

This chapter presents four CSL scripts and the visuals produced by their generated game objects. Note that the code examples use a `colorize` function, but exclude its definition for the sake of brevity.

7.1 Life with traces

Figure 7.1 shows the `life_with_traces` effect, introduced in Chapter 3. The script makes use of CSL's ability to store past configurations.

```
process life(Alive: Bool):
       AliveCount <- (boolToInt Alive)[moore by (+)]
       Alive <- if Alive
                then (AliveCount ==i 2) || (AliveCount ==i 3)
                else (AliveCount ==i 3)

process main:
       life Alive
       Alive <- Alive || inPresence

       -- 'lastb' returns the configuration that
       -- its argument had in the last step
       AliveP1 <- lastb Alive
       AliveP2 <- lastb AliveP1

       outDisplay (colorize Alive AliveP1 AliveP2)
```

[1] See https://docs.godotengine.org/en/3.5/tutorials/ui/size_and_anchors.html

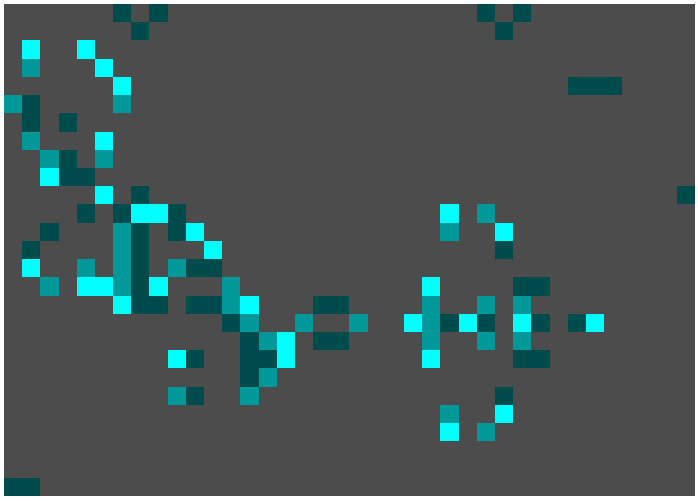


Figure 7.1: The `life_with_traces.csl` effect

7.2 Moss burner

Figure 7.2 shows the implementation of the moss-burning effect from Chapter 4.

```
process ignite(Lighter: Bool, Fire: Bool):
	Fire <- Fire || Lighter

process plant(Planter: Bool, Moss: Bool):
	Moss <- Moss || Planter

process grow(Moss: Bool):
	Moss <- Moss[0,0] || Moss[moore by (||)]

process burn(Fire: Bool, Moss: Bool):
	SpreadFire <- Fire[0,0] || Fire[vonNeumann by (||)]
	Fire <- Moss && SpreadFire
	Moss <- Moss && (not Fire)

process main:
	-- Two 'inPresence' processes allow input of both moss and fire
	inPresence Planter
	inPresence Torch

	ignite Torch Fire
	plant Planter Moss

	grow Moss
	burn Fire Moss

	outDisplay (colorize Fire Moss)
```

Figure 7.2: The `moss_burner.csl` effect

7.3 One of eight

Figure 7.3 shows an automaton from [11], called `ONE_OF_EIGHT`. It produces ring-like patterns that I just found interesting.

```
process main:
        Alive <- (boolToInt Alive)[moore by (+)] ==i 1
        Alive <- Alive || inPresence

        outDisplay (colorize Alive)
```

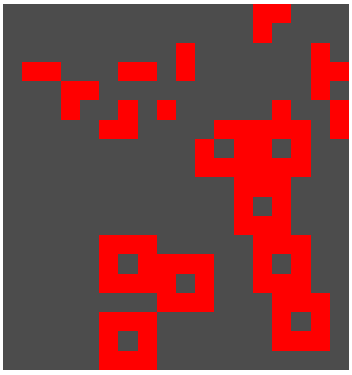


Figure 7.3: The `one_of_eight.csl` effect

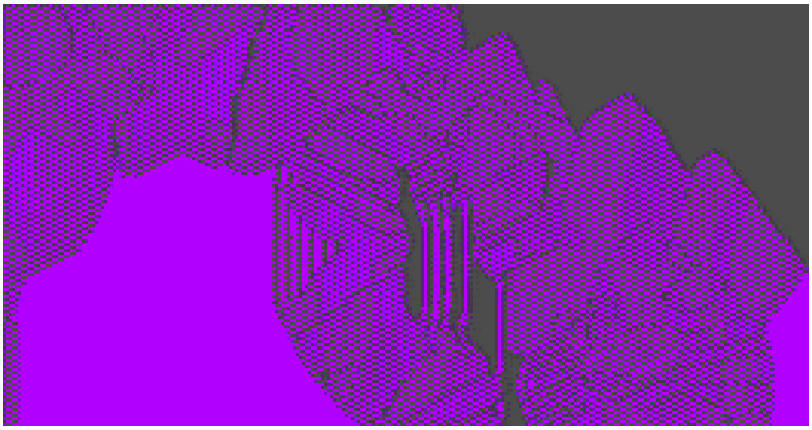


Figure 7.4: The `forcefield.csl` effect

7.4 Forcefield

Figure 7.4 shows a so-called *neural cellular automaton*, producing an effect with organic appearance. The cell state in these 2CA is continuous rather than discrete and they work by first performing a convolution of the neighborhood and then passing the resulting value through an activation function. These concepts, inspired by machine learning, can too be implemented in CSL.

```
process convolution(In: Float, Out: Float):
	Out <- (In[-1,-1] *f (-0.452)) +f (In[0,-1] *f 0.344)
		+f (In[1,-1] *f 0.807) +f (In[-1,0] *f 0.292)
		+f (In[0,0] *f 0.393) +f (In[1,0] *f (-0.447))
		+f (In[-1,1] *f (-0.816)) +f (In[0,1] *f 0.645)
		+f (In[1,1] *f 0.878)

process activation(In: Float, Out: Float):
	Powed <- In[0,0] *f In[0,0]
	Dimmed <- 0.5 *f Powed[0,0]
	Clamped <- if Dimmed[0,0] >f 1.0 then 1.0 else Dimmed[0,0]

	Out <- Clamped[0,0]


process forcefield(Field: Float):
	Field <- activation(convolution Field[0,0])

process main:
	Input <- if inPresence then 1.0 else 0.0
	Field <- Field +f Input

	forcefield Field
	outDisplay (colorize Field)
```

Chapter 8

Conclusion

The goal of this thesis was to introduce a modified version of cellular automata, study its properties and apply this automaton to a suitable field of visual art. The result is a language for embedding composable cellular automata into videogames.

In the beginning chapter, a few concepts from system theory and automata theory were described. These included systems and the very basics of modeling using functional blocks. Deterministic finite automata, four-way automata and cellular automata were covered, including examples of neighborhoods and boundary conditions. In the next section, the key idea of representing a cellular automaton in a functional block diagram was introduced.

The next chapter further explored cellular automata. Four techniques useful in modeling were presented, along with the visuals they can produce. Accessing past configurations can be used to add a trace effect to an automaton or used in other ways in the transition rule. The sand rule can be used to implement an hourglass animation. And the FHP cellular automaton can produce a wave effect.

Following this came the new variant of cellular automaton, or rather, a new method of composing them. This new model of representation brings two advantages. First, it provides a unified interface through which effects of multiple automata may be composed, both in series and in parallel. Second, it simplifies working with past configurations in the automaton. Wherever a past configuration is needed, a feedback loop is added. Multiple feedback connections allow for multi-step time delays. The way this automaton works in a game was described. This automaton was shown to be equivalent in power to normal cellular automata, as one model can mimic any move of the other, albeit this was proven only for the case where all rules share the same boundary condition.

Then a new domain-specific language was introduced as a way to define these systems and embed them in videogames. The original inspiration was given and a few basic example scripts in the language were presented. Full documentation was given in an appendix. A high-level overview of the compiler implementation was given in the next chapter.

Several CSL scripts were tested, producing interactive visual effects. The first is an implementation of Life with traces, showcasing the ability of CSL to store past configurations. The second is the burning moss effect, demonstrating the use of multiple inputs from the game scene and the composition of two separate cellular automata, one for growth and one for burning. The third is just an effect I found interesting. The final effect is a so-called neural cellular automaton, combining concepts from machine learning and automata.

I have a few ideas for various extensions to cellular systems. For instance, the model could be combined with pattern recognition. Areas of the grid that form a certain pattern could have a different transition function applied. Or, they could me made so that they

are not just limited to a certain region of the game scene, but can automatically expand to cover any area. This expansion could take place when a cell reaches the edge of the automaton. There could also be output blocks that synthesize sound as well, where the float value within a cell would be considered the amplitude of the sound wave at that location and time. Currently, the Godot implementation runs these automata at around 60 steps per second, which would yield a maximum frequency of 60 Hz. There might be a way to run them faster or somehow interpolate the missing samples in the sound wave, to achieve audible frequencies.

The systems could also have impact on the gameplay. For instance, calling a method on any game object that enters a cell in a certain state. This would, for instance, allow modeling toxic gas, not only visually, but also have it damage the player by calling methods on the character in a cell containing gas. The cells in the automaton could be given a collision box to make them interact with physics objects. The automata could be used not only in-game, but also in algorithms for procedural level generation, such as generating dungeons for roguelike videogames.

The language could use polymorphism and better type inference. It could also have better support for modeling the movement of particles. Constructs such as a Rust-like `match` expression and enumerator types would be useful. Higher-order processes would allow the user of a process to fine-tune its behavior, much like higher-order functions in other languages. Another extension could be better handling of time. Currently, there is the `lasti/lastf/lastb` process. But besides introducing a time delay, perhaps there could be a keyword to make the automaton run backwards in time. Or a way to selectively speed up or slow down time in certain areas of the grid. Or to introduce an operator similar to the neighbor access operator, that would simply allow the user to access past configurations with an index. Similarly there could a time-themed variant of the neighborhood fold.

The compiler is also designed to be able to support multiple front ends and back ends. It can be extended with a graphical node-based editor on the front end, similar to many of the graphical shader editors available today. On the back end, a built-in simulator could be added, maybe even with a more scientific focus, such as for biology simulations. Or, it could generate code that would run as a live wallpaper on the desktop. Or just generate a static image.

Bibliography

[1] Åström, K. and Murray, R. *Feedback Systems: An Introduction for Scientists and Engineers, Second Edition*. Princeton University Press, 2021. ISBN 9780691193984.

[2] Bertalanffy, L. von. *General System Theory: Foundations, Development, Applications*. George Braziller, Inc., 1968.

[3] Chopard, B. and Droz, M. *Cellular Automata Modeling of Physical Systems*. January 1998.

[4] Durand, M., Shogry, S. and Baltzly, D. Stoicism. In: Zalta, E. N. and Nodelman, U., ed. *The Stanford Encyclopedia of Philosophy*. Spring 2023th ed. Metaphysics Research Lab, Stanford University, 2023 [cit. 2023-12-10]. Available at: https://plato.stanford.edu/archives/spr2023/entries/stoicism/.

[5] Healy, P. and Nikolov, N. S. How to Layer a Directed Acyclic Graph. In: *International Symposium Graph Drawing and Network Visualization*. 2001 [cit. 2024-01-20]. Available at: https://api.semanticscholar.org/CorpusID:206628397.

[6] Johnson, N. *Spaceship Speed Limits in „B3“ Life-Like Cellular Automata* [online]. October 2009 [cit. 2024-01-27]. Available at: http://www.njohnston.ca/2009/10/spaceship-speed-limits-in-life-like-cellular-automata/.

[7] Marlow, S. *Haskell 2010 Language Report*. 2010. Available at: https://www.haskell.org/onlinereport/haskell2010/.

[8] Peringer, P. and Hrubý, M. *Lecture slides for Modelling and Simulation*. September 2022 [cit. 2023-11-09]. Available at: https://www.fit.vutbr.cz/study/courses/IMS/public/prednasky/IMS.pdf.

[9] Priemer, R. *Introductory Signal Processing*. World Scientific, 1991. Advanced series in electrical and computer engineering. ISBN 9789971509194.

[10] Rozenberg, G. and Salomaa, A. *Handbook of Formal Languages*. January 1997.

[11] Toffoli, T. and Margolus, N. *Cellular Automata Machines: A New Environment for Modeling*. January 1987. ISBN 9780262291019.

Appendix A

SD card contents

- `app/` – application source code (CSL compiler)
- `app/README.md` – instructions on usage
- `report/` – LaTeX source code for the thesis text
- `godot_test/` – CSL source code and the Godot project used for testing
- `bin/` – CSL compiler executable
- `xalbre05.pdf` – thesis text

Appendix B

CSL documentation

B.1 Introduction

A CSL script defines the structure of a single cellular system and consists of *processes* and *substances*. A substance defines an alphabet (product data type, a struct) using other substances. A process defines a block containing a CS using other processes. Built-in substances and processes are provided as a starting point for implementing new ones. When invoked, the compiler looks for a process called `main`, which is used as the resulting CS of the script.

A process is written similarly to a function in other languages, but instead of containing a sequence of statements, it contains a sequence of expressions. An expression consists of processes applied to other expressions, constants or signals and connects one or more new blocks to the system of the process. Signals are names given to inputs and outputs of processes to connect their blocks together.

The document uses Backus-Naur form to define the lexemes and syntax of the language. Empty strings are denoted by λ. Semantics of various constructs are described in terms of their analogous definition using a CS, using a more high-level description. Exact mathematical definitions are used where needed.

B.2 Lexical analysis

Comments

Comments can be located anywhere within the document, start with a double dash (`--`) and run to the end of the line:

```
-- This is a comment!

neighborhood center: [0,0] -- This is also a comment!
```

Numeric literals

CSL has literals for integers and floats, which are defined the same way as in [7]:

⟨digit⟩ ::= ⟨ascDigit⟩ | ⟨uniDigit⟩
⟨ascDigit⟩ ::= `0` | `1` | `...` | `9`
⟨uniDigit⟩ ::= *any Unicode decimal digit*
⟨octit⟩ ::= `0` | `1` | `...` | `7`
⟨hexit⟩ ::= ⟨digit⟩ | `A` | `...` | `F` | `a` | `...` | `f`
⟨decimal⟩ ::= ⟨digit⟩ | ⟨digit⟩⟨decimal⟩
⟨octal⟩ ::= ⟨octit⟩ | ⟨octit⟩⟨octal⟩
⟨hexadecimal⟩ ::= ⟨hexit⟩ | ⟨hexit⟩⟨hexadecimal⟩
⟨exponent⟩ ::= `e`⟨decimal⟩ | `e-`⟨decimal⟩ | `e+`⟨decimal⟩
⟨intLit⟩ ::= ⟨decimal⟩ | `0o`⟨octal⟩ | `0x`⟨hexadecimal⟩
⟨floatLit⟩ ::= ⟨decimal⟩`.`⟨decimal⟩ | ⟨decimal⟩⟨exponent⟩
| ⟨decimal⟩`.`⟨decimal⟩⟨exponent⟩

Identifiers and keywords

⟨char⟩ ::= *alphanumeric Unicode character*
⟨upperChar⟩ ::= *an uppercase Unicode character*
⟨lowerChar⟩ ::= *a lowercase Unicode character*
⟨symbolChar⟩ ::= *a Unicode symbol character*
⟨chars⟩ ::= ⟨char⟩⟨chars⟩ | λ
⟨symsOrChars⟩ ::= ⟨symbolChar⟩⟨symsOrChars⟩ | ⟨char⟩⟨symsOrChars⟩ | λ
⟨prefixProcessId⟩ ::= ⟨lowerChar⟩⟨chars⟩
⟨neighborhoodId⟩ ::= ⟨lowerChar⟩⟨chars⟩
⟨substanceMemberId⟩ ::= ⟨lowerChar⟩⟨chars⟩
⟨infixProcessId⟩ ::= ⟨symbolChar⟩⟨symsOrChars⟩
⟨substanceId⟩ ::= ⟨upperChar⟩⟨chars⟩
⟨signalId⟩ ::= ⟨upperChar⟩⟨chars⟩
⟨keyword⟩ ::= `if` | `then` | `else` | `by` | `with`
| `process` | `substance` | `neighborhood`
| `fixed` | `periodic` | `adiabatic` | `reflective`
| `boundary` | `true` | `false`

Identifiers are of six types. Identifiers for processes used in prefix form, identifiers for neighborhoods and names for substance members begin with a lowercase letter. Instead of operators, CSL has „infix processes“. Their identifier starts with a symbol character. Signals and substances begin with a capital letter. Keywords are reserved and cannot be used as identifiers.

Indentation

CSL is indentation sensitive. Top level items in a script must *not* be indented, while their body *must* be indented. Both spaces and tabs can be used for indentation, as long as all lines in the body are indented at the same level. Expressions cannot span multiple lines, unless specified otherwise. There are no nested scopes.

B.3 Full grammar

⟨EOL⟩ ::= *the appropriate end of line specifier*
⟨IND⟩ ::= *an indented line of a body as described above*
⟨script⟩ ::= ⟨topItem⟩ | ⟨topItem⟩ ⟨script⟩
⟨topItem⟩ ::= ⟨substance⟩ | ⟨neighborhood⟩ | ⟨process⟩
⟨substance⟩ ::= `substance` ⟨substanceId⟩ `:` ⟨EOL⟩⟨memberLines⟩
⟨memberLines⟩ ::= ⟨memberLine⟩ | ⟨memberLine⟩⟨memberLines⟩
⟨memberLine⟩ ::= ⟨IND⟩⟨substanceMemberId⟩ `:` ⟨substanceId⟩⟨EOL⟩
⟨neighborhood⟩ ::= `neighborhood` ⟨neighborhoodId⟩ `:`⟨neighborPosList⟩
⟨neighborPosList⟩ ::= ⟨neighborPos⟩ | ⟨neighborPos⟩ ⟨neighborPosList⟩
⟨neighborPos⟩ ::= `[`⟨intLit⟩`,` ⟨intLit⟩`]`
⟨processId⟩ ::= ⟨prefixProcessId⟩ | `(`⟨infixProcessId⟩`)`
⟨process⟩ ::= ⟨closedProcess⟩ | ⟨openProcess⟩
⟨closedProcess⟩ ::= `process` ⟨processId⟩ ⟨boundarySpec⟩ `:` ⟨EOL⟩⟨exprLines⟩
⟨boundarySpec⟩ ::= `with` ⟨boundaryCond⟩ `boundary` | λ
⟨boundaryCond⟩ ::= `fixed` | `periodic` | `adiabatic` | `reflective`
⟨openProcess⟩ ::= `process` ⟨processId⟩ `(`⟨signalList⟩`)` `:` ⟨EOL⟩⟨exprLines⟩
⟨signalList⟩ ::= ⟨annotatedSignal⟩ | ⟨annotatedSignal⟩ `,` ⟨signalList⟩
⟨annotatedSignal⟩ ::= ⟨signalId⟩ `:` ⟨substanceId⟩
⟨exprLines⟩ ::= ⟨exprLine⟩ | ⟨exprLine⟩⟨exprLines⟩
⟨exprLine⟩ ::= ⟨IND⟩⟨expr⟩⟨EOL⟩
⟨expr⟩ ::= `pass` | `if` ⟨expr⟩ `then` ⟨expr⟩ `else` ⟨expr⟩
| ⟨prefixProcessId⟩ | ⟨prefixProcessId⟩ ⟨exprList⟩
| ⟨expr⟩ ⟨infixProcessId⟩ ⟨expr⟩ | ⟨substanceConstructor⟩
| ⟨neighborAccess⟩ | ⟨memberAccess⟩ | ⟨signalId⟩
| ⟨neighborhoodFold⟩ | ⟨assignment⟩ | `(`⟨expr⟩`)`
| ⟨floatLit⟩ | ⟨intLit⟩
⟨exprList⟩ ::= ⟨expr⟩ | ⟨expr⟩ ⟨exprList⟩
⟨substanceConstructor⟩ ::= ⟨substanceId⟩ `{` ⟨constructorItems⟩ `}`
⟨constructorItems⟩ ::= ⟨substanceMemberId⟩ `<-` ⟨expr⟩
| ⟨substanceMemberId⟩ `<-` ⟨expr⟩ `,` ⟨constructorItems⟩
⟨neighborAccess⟩ ::= ⟨expr⟩`[`⟨intLit⟩`,` ⟨intLit⟩`]`
⟨memberAccess⟩ ::= ⟨expr⟩`.`⟨substanceMemberId⟩
⟨neighborhoodFold⟩ ::= ⟨expr⟩`[` ⟨neighborhoodId⟩ `by` ⟨processId⟩ `]`
⟨assignment⟩ ::= ⟨signalId⟩⟨optMember⟩ `<-` ⟨expr⟩
⟨optMember⟩ ::= `.`⟨substanceMemberId⟩ | λ

B.4 Items

Top-level items can be defined in the source code in any order and can be used before they are defined, as long as they are defined somewhere within the script.

Substances

From a user perspective, a substance represents matter, information or energy in the miniature world being modeled. Formally, a substance defines an alphabet that is the Cartesian product of other alphabets. It is analogous to a `struct` in other languages. For example:

```
substance Color:
        r: Float
        g: Float
        b: Float
```

This snippet defines a new substance named `Color` with an alphabet $\Sigma = \Sigma_{float} \times \Sigma_{float} \times \Sigma_{float}$, where Σ_{float} represents all valid encodings of a float. A substance can be constructed using a constructor expression (Section B.5) and members can be accessed using an access expression (Section B.5).

Neighborhoods

A neighborhood defines a list of relative cell positions that make up the neighbors of a center cell:

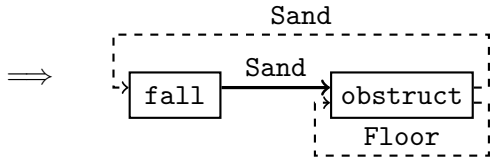
```
neighborhood vonNeumann: [-1, 0] [1, 0] [0, -1] [0, 1]
```

They are used in neighborhood fold expressions (Section B.5) to implement totalistic rules, which are rules that depend only on some aggregate property of the neighbors of a cell, such as their sum.

Processes

Processes model behavior by defining a block containing a cellular system. Each line in the body of a process contains an expression, which adds blocks to the system and connects them:

```
process test:
        fall Sand
        obstruct Sand Floor
```



The process `test` contains two *process application* expressions (Section B.5). Let `fall` be a process defining a block with one input/output and `obstruct` be a process defining a block with two inputs/outputs. The expression `fall Sand` *adds* the block defined by `fall` and *binds* its first input/output to a signal named `Sand`. A signal is similar to a „thread“ that is run through different processes to connect them. Formally, it is a name assigned to the *input of the process* where this signal identifier was *first* used and the *output of the process* where it was *last* used.

The expression `obstruct Sand Floor` uses the existing `Sand` signal to bind the output of `fall` to the first input of `obstruct` and binds the second input/output to a new `Floor`

signal. Feedback connections are automatically introduced between the output where a signal ended and the input where it started. `Sand` started at the first input of `fall` and ended at the first output of `obstruct`. `Floor` started at the second input of `obstruct` and ended at its second output.

There are situations where the process ends up with unused inputs, where no signal started, or unused outputs, where no signal ended. Such unused inputs are assigned default values and unused outputs are voided.

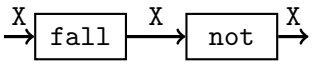
A process can be either *closed* or *open*, defining a closed or open cellular system respectively. A closed process is defined without a parameter list, like the `test` process above. It is also possible to specify a different boundary condition for the closed system:

```
process test with periodic boundary:
    ...
```

This will set the *periodic* boundary condition for the system defined by `test`. An open process is defined with a list of parameter signals:

```
process myOpen(X: Bool):
    fall X
    not X
```

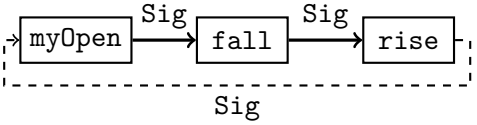
$\Longrightarrow$



Parameter signals must have a substance (type) specified. In the `myOpen` process, `fall` binds its input to `X`, `X` is passed on to `not` and finally `not` binds its output to `X`. It can then be used in any other process like so:

```
process main:
    myOpen Sig
    fall Sig
    rise Sig
```

$\Longrightarrow$



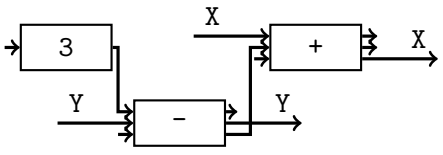
In a process application, the alphabet of newly created signals is deduced from the parameter list of the open process being applied. The signal `Sig` is first used as the first parameter of `myOpen`, which assigns the `Bool` substance to it, since that is the type of its first parameter. Note that `fall` and `not` are also open processes.

B.5 Expressions

Expressions construct and connect together one or more blocks. For example:

```
X <- X + (3 - Y)
```

$\Longrightarrow$



CSL does not have arithmetic or logical operators. Instead of being operators, `+` and `-` are simply processes in infix form. The signature of `+`, for example, is `process (+)(A: Int, B: Int, Res: Int)`. It reads `A` and `B` as operands and overwrites `Res` with the result of addition. The rationale behind this is to unify the concept of functions (operators) and

processes and avoid having two similar constructs in the language. It also allows defining operators that have their own *internal state*.

When an open process, such as `+`, is used in an expression, the last signal in its parameter list is used as the output. The output of an expression can be assigned to a signal using the assignment operator `<-` (Section B.5).

The only special operator is the assignment operator (Section B.5). Infix processes take precedence over assignment. All infix processes are left-associative.

Assignment expression

⟨assignment⟩ ::= ⟨signalId⟩ `<-` ⟨expr⟩

An assignment assigns an expression to a signal. If the signal exists and its type matches the type of the expression's output, the signal's end is moved to this output. If the signal does not exist, a new signal is created, that starts nowhere and ends at the expression's output.

The output of the assignment expression is the output of its assigned expression.

```
process sigExists:
        -- MyBoolSig starts at the input of 'not' ...
        not MyBoolSig
        -- ...and ends at the output of the 'true' expression
        MyBoolSig <- true

        -- A feedback connection is made for MyBoolSig, since
        -- it has a start and an end

process sigNew(A: Int, B: Int):
        -- MyNum is created, starting nowhere (there
        -- are not even unused inputs for it start somewhere)
        -- and ending at the output of 'A + B'
        MyNum <- A + B

        -- A feedback connection is not made for MyNum, since
        -- it starts nowhere, and the output of 'A + B' is voided
```

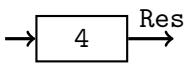
Constant expression

⟨constant⟩ ::= ⟨intLit⟩ | ⟨floatLit⟩

A constant expression introduces a new block, with a single input/output that outputs the value of the constant. The output of the expression is the output of this block.

```
process main:
        Res <- 4
```

⟹



Signal identifier expression

⟨signalId⟩ ::= ⟨upperChar⟩⟨chars⟩

If a signal identifier of an existing signal is used in an expression, the output where this signal ended is considered the input of the expression (if the type matches). If the signal does not exist, it is automatically declared, as long as the type it should have is known.

```
process main:
       -- Known is automatically declared,
       -- as + expects an Int
       Res <- Known + 3

       -- Unknown cannot be automatically declared,
       -- as its type cannot be deduced
       Res2 <- Unknown[1,2]
```

Process application expression

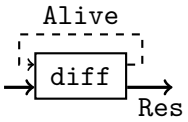
⟨prefixProc⟩ ::= ⟨prefixProcessId⟩ ⟨exprList⟩
⟨infixProc⟩ ::= ⟨expr⟩ ⟨infixProcessId⟩ ⟨expr⟩

A process application expression adds the block defined by the used process and connects the outputs of the subexpressions to the inputs of the block. The output of the expression is the last output of the block.

```
process main:
       -- 'diff' is applied to 'Alive'.
       Res <- diff Alive

-- The 'Out' signal is used as the output of
-- the above expression
process diff(In: Bool, Out: Bool):
       Out <- Last[0,0]
       Last <- In[0,0]
```

⟹



If-then-else expression

⟨ifThenElse⟩ ::= `if` ⟨expr⟩ `then` ⟨expr⟩ `else` ⟨expr⟩

An if-then-else expression adds a block that outputs its third input if its first input is `true` otherwise it outputs its second input. The second and third input must be of the same type. If expressions can span multiple lines.

```
process main:
       Res <- if Alive then 3 * 4 else 20
```

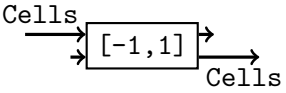
Neighbor access expression

⟨neighborAccess⟩ ::= ⟨expr⟩`[`⟨intLit⟩`,` ⟨intLit⟩`]`

A neighbor access expression inserts a block that reads the given neighbor of the subexpression's output and outputs the result.

```
process shift(Cells: Bool):
        Cells <- Cells[-1,1]
```

$\Longrightarrow$



Which contains the elementary rule $R = ((\Sigma_{bool}, \Sigma_{bool}), ((0,0),(-1,1)), (\delta_{id}, \delta_g), F)$, where $\delta_{id}(i_1, i_2, x_1, x_2) = i_1$ and $\delta_g(i_1, i_2, x_1, x_2) = i_2$.

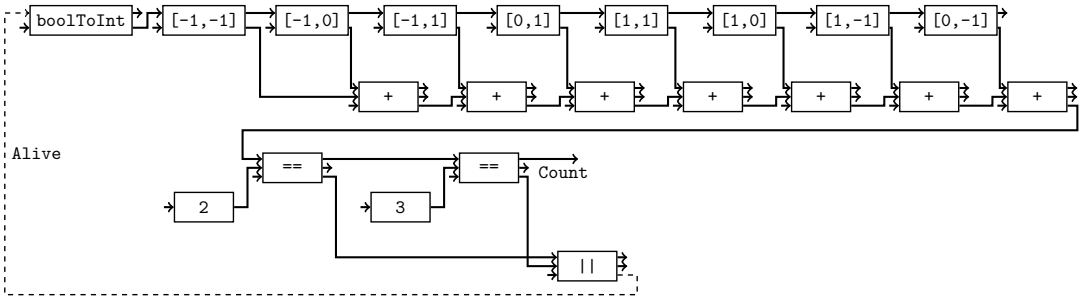
Neighborhood fold expression

⟨neighborhoodFold⟩ ::= ⟨expr⟩`[` ⟨neighborhoodId⟩ `by` ⟨processId⟩ `]`

A neighborhood fold gets the values of the subexpression neighbors given by the neighborhood and sequentially applies the given process to the results from left to right. The following code runs Conway's Game of Life (a slightly broken version):

```
process gameOfLife:
        Count <- (boolToInt Alive)[moore by (+)]
        Alive <- (Count == 2) || (Count == 3)
```

And produces:



Substance constructor expression

⟨substanceConstructor⟩ ::= ⟨substanceId⟩ `{` ⟨constructorItems⟩ `}`

⟨constructorItems⟩ ::= ⟨substanceMemberId⟩ `<-` ⟨expr⟩

| ⟨substanceMemberId⟩ `<-` ⟨expr⟩ `,` ⟨constructorItems⟩

A substance constructor adds a block that constructs a tuple at the output from its inputs. This output tuple is an element of the Cartesian product defined by the substance. Constructors can span multiple lines.

```
substance Color:
       r: Float
       g: Float
       b: Float

process main:
       -- +f is addition of floats
       R <- R +f 1.0
       G <- G *f 0.8

       -- Color has the type of Color
       Color <- Color {
              r <- R,
              g <- G,
              b <- R *f 2.0
       }
```

Substance member access expression

$$\langle \text{memberAccess} \rangle \quad ::= \quad \langle \text{expr} \rangle \, . \, \langle \text{substanceMemberId} \rangle$$

Components of substances can be accessed using the member access expression. It adds a block that extracts a component of a tuple.

```
substance Test:
       a: Int
       b: Bool

process main:
       S <- Test { a <- 3, b <- true }
       B <- S.b
```

B.6 Built-in neighborhoods

`moore`
: The Moore neighborhood, excluding the center cell.

`vonNeumann`
: The Von Neumann neighborhood, excluding the center cell.

B.7 Built-in substances

`Int`
: A basic integer type (8-bit for Godot).

`Float`
: A basic float type (16-bit for Godot).

`Bool`
: A basic boolean type (1-bit for Godot).

`Color`
: Represents a color in RGBA format.

```
substance Color:
        r: Float
        g: Float
        b: Float
        a: Float
```

B.8 Built-in processes

`inRasterize(Res: Color)`
: Rasterizes objects that enter the region of the system within the game scene and outputs their color. In Godot, it currently only rasterizes `Sprite` nodes.

  Godot Inspector options:

  groups
  : The names of the node groups this process should detect.

`inPresence(Res: Bool)`
: Detects objects that enter the region of the system within the game scene. In Godot, it currently detects only `Sprite` nodes.

  Godot Inspector options:

  groups
  : The names of the node groups this process should detect.

`outDisplay(In: Color)`
: Displays the given color substance in the region of the game scene occupied by the system. Using multiple `outDisplay` processes overlays them on top of each other.

`lasti(In: Int, Out: Int)`
`lastf(In: Float, Out: Int)`
`lastb(In: Bool, Out: Int)`
: Overwrites `Out` with the value that `In` had in the last step.

`boolToInt(In: Bool, Res: Int)`
: Writes `1` to `Res` if `In` is `True`, otherwise writes `0` to `Res`.

`boolToColor(In: Bool, Res: Color)`
: Writes a white color to `Res` if `In` is `True`, otherwise writes a black color to `Res`.

`(+)(A: Int, B: Int, Res: Int)`
`(+f)(A: Float, B: Float, Res: Float)`
: Performs addition of `A` and `B` and overwrites `Res` with the result.

`(-)(A: Int, B: Int, Res: Int)`

`(-f)(A: Float, B: Float, Res:  Float)`
Performs subtraction of `A` and `B` and overwrites `Res` with the result.

`(*)(A: Int, B: Int, Res:  Int)`
`(*f)(A: Float, B: Float, Res:  Float)`
Performs multiplication of `A` and `B` and overwrites `Res` with the result.

`(/)(A: Int, B: Int, Res:  Int)`
`(/f)(A: Float, B: Float, Res:  Float)`
Performs division of `A` and `B` and overwrites `Res` with the result.

`(&&)(A: Bool, B: Bool, Res:  Bool)`
Performs the AND operation on `A` and `B` and overwrites `Res` with the result.

`(||)(A: Bool, B: Bool, Res:  Bool)`
Performs the OR operation on `A` and `B` and overwrites `Res` with the result.

`not(A: Bool, res:  Bool)`
Performs the NOT operation on `A` and overwrites `Res` with the result.

`(==i)(A: Int, B: Int, Res:  Bool)`
`(==f)(A: Float, B: Float, Res:  Bool)`
`(==b)(A: Bool, B: Bool, Res:  Bool)`
`(>i)(A: Int, B: Int, Res:  Bool)`
`(>f)(A: Float, B: Float, Res:  Bool)`
`(<i)(A: Int, B: Int, Res:  Bool)`
`(<f)(A: Float, B: Float, Res:  Bool)`
Overwrites `Res` with the result of the comparison of `A` and `B`.