Composition via Quantum Cellular Automata

Hector Miller-Bakewell, Eduardo Miranda

2021-11-19

Goal

To facilitate **musicians** using quantum computers to aid composition.

Building on existing classical techniques.

What's to come

What's in this presentation

- Cellular Automata
- **CAMUS** The **C**ellular **A**utomaton **MUS**ic generation system (Miranda)
- (Partitioned) Quantum Cellular Automata
- Combining the two approaches

Classical Cellular Automata

Cellular Automata

Discretised Spacetime

Cells form a grid

Time advances in steps

Bounded Information Propagation

Each cell's next state is dependent only on nearby cells' current states

Finite states

For example, a bit

Example Cellular Automata

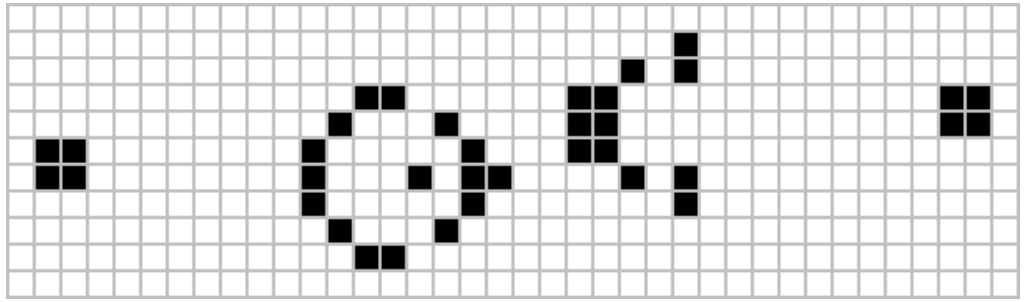


Figure 1: John Conway's Game Of Life

Space is a 2D grid of cells, each is either On or Off

A cell's next state is determined by its own state and its current level of crowding

Why Cellular Automata?

Cellular Automata originated in the 1960s (von Neumann and Ulam) as tools to examine self-replicating behaviour.

Since then they have been image processing (Preston and Duff, 1984), ecology (Hogeweg, 1988), sociology (Epstein and Axtell, 1996), etc..

They exhibit complex emergent behaviour from simple rules.

History of Cellular Automata in music

Iannis Xenakis, mid 1980s, used cellular automata "to create complex temporal evolution of orchestral clusters"

Other examples include Beyls (1989), Millen (1990), and Miranda (1990)

CAMUS

From CA to music

Input: The state of (two) 2D cellular automata, at a **single** moment in time

Output: A **sequence** of triads, of varying pitch composition, instrumentation, and timing

Choosing the intervals of a triad

The root of the triad is determined in advance[1]

```
for cell in first_automaton:
    if cell is True:
        first_interval <- cell.x
        second_interval <- cell.y
```

[1] In the original CAMUS this was the case - we'll change this later

Choosing the intervals of a triad

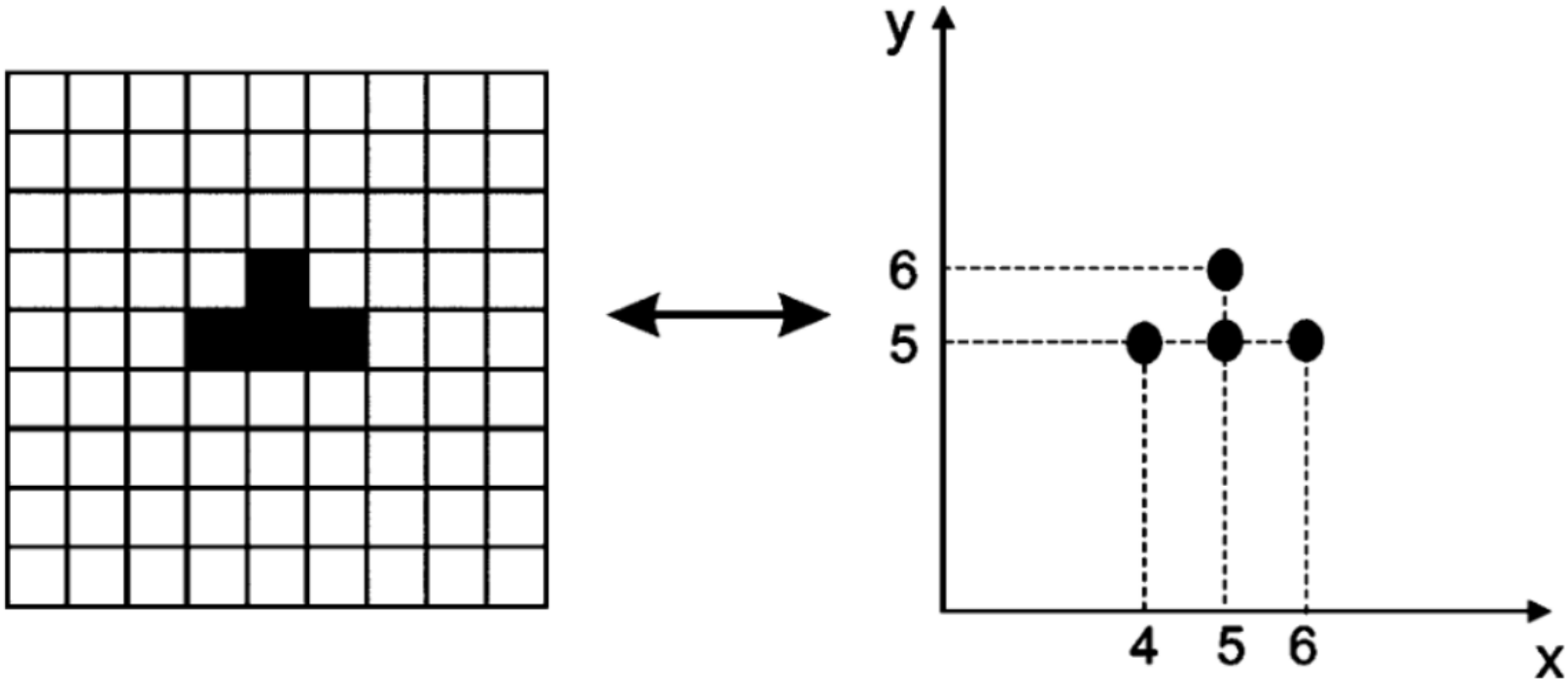
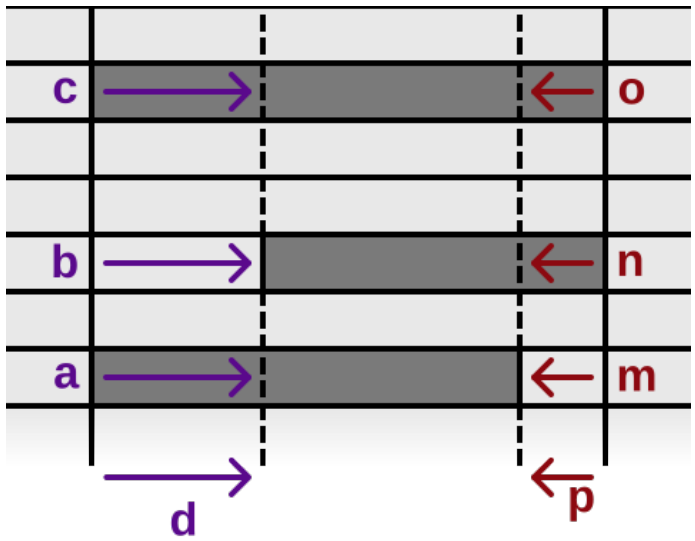


Figure 2: From cells to triads

Temporal Morphology

The timings of the voices is determined by the neighbours of the cell in the **triad** grid.



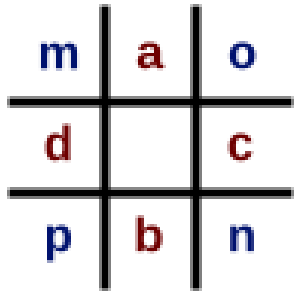


Figure 3: How the neighbours affect timings

Choosing the instruments of a triad

The instrument of a triad is determined by looking up the same `x`, `y` coordinate in a *second* cellular automaton.

```
for cell in first_automaton:
    if cell is True:
        first_interval <- cell.x
        second_interval <- cell.y
        # ... determine timings ...
        instrument <- lookup(cell.x, cell.y)
```

Choosing the instruments of a triad

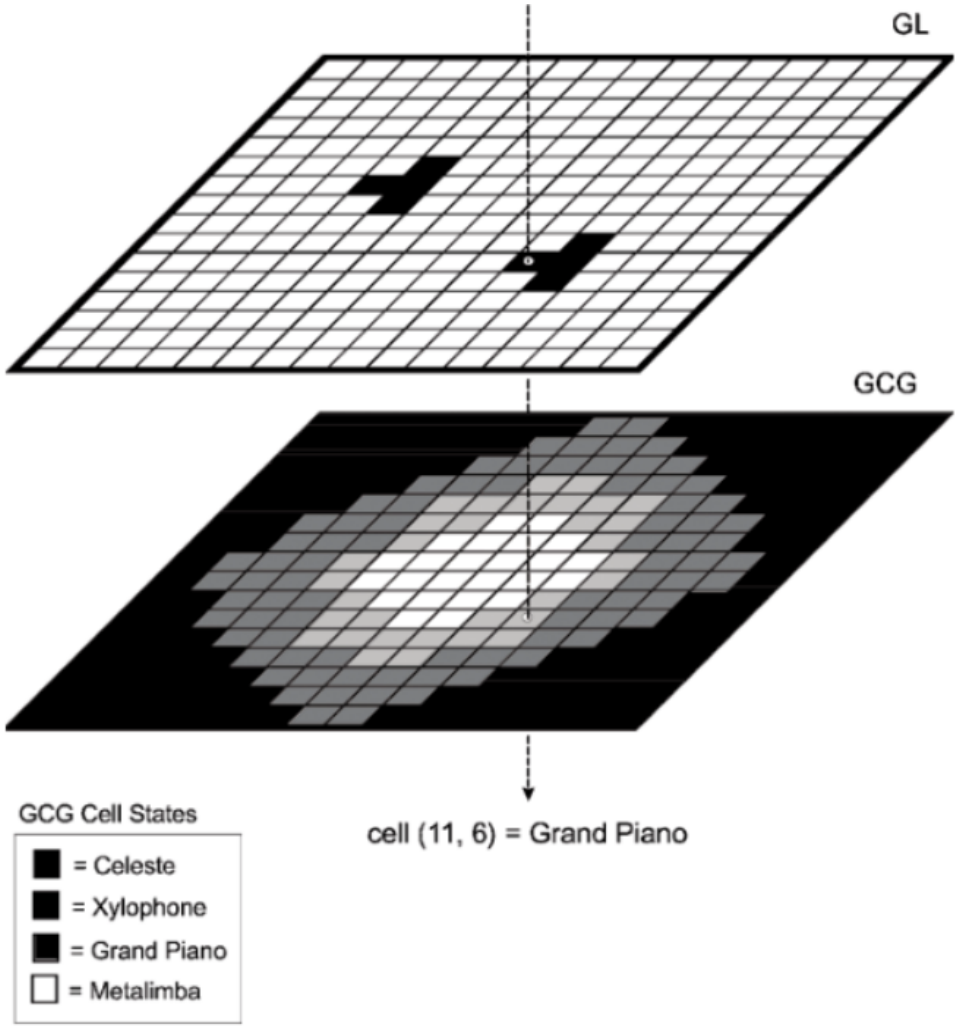


Figure 4: Using two automata to determine triad and instrument

Three Instruments

Rather than a single instrument for the triad, we assign each voice an instrument, using the neighbours of the cell in the **instrument** grid.

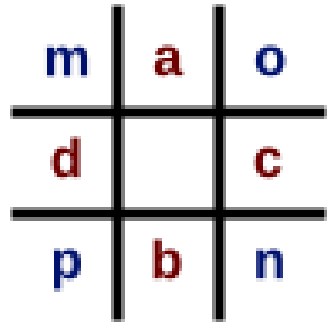


Figure 5: Labelling the neighbours

Quantum Cellular Automata

Quantum Cellular Automata[2]

Discretised Spacetime

Cells form a grid

Time advances in steps

Bounded Information Propagation

Each cell's next state is dependent only on nearby cells' current states

Quantum states

For example, a qubit

[2] A review of Quantum Cellular Automata (Farrelly, Quantum 2020)

Constructivity

These desiderata are not constructive

Simultaneity

An important part of classical cellular automata is that the update step is performed both in parallel, and simultaneously to all cells.

This is a problem when we can't copy information.

(P)QCA

The work of Arrighi and Grattage (Natural Computing, 2012) showed that the various different definitions of Quantum Cellular Automata in the literature could all be simulated by Partitioned Quantum Cellular Automata (PQCA).

PQCA, defined next slide, are constructive and universal.

PQCA

A **partition** divides the cells at a given time into tessellating supercells.

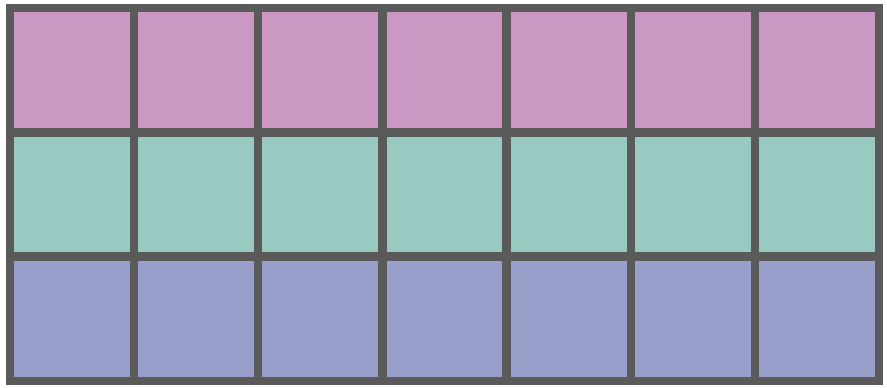


Figure 6: Horizontal Supercells

A unitary is then applied to each supercell.

The full update step is built from several such partitions and unitaries.

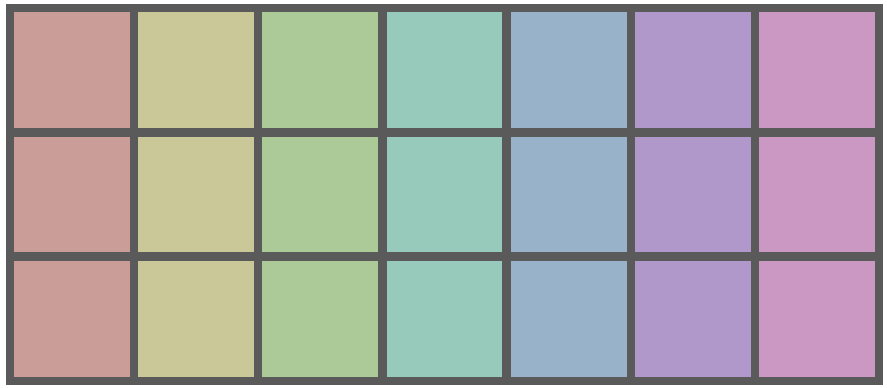


Figure 7: Vertical Supercells

Building a PQCA

Example Code

This code describes two tessellations of a 9 by 4 grid using the pqca package.

```
import pqca
tes_1 = pqca.tessellation.n_dimensional([9,4],[1,2])
tes_2 = pqca.tessellation.n_dimensional([9,4],[3,1])
```

Example 2-qubit Unitary

We can then use `qiskit` (or anything that exports to `.qasm`) to build a circuit.

```
two_qubit_circuit = qiskit.QuantumCircuit(2)
two_qubit_circuit.cx(0,1)
two_qubit_circuit.h(0)
two_qubit_circuit.draw()
```

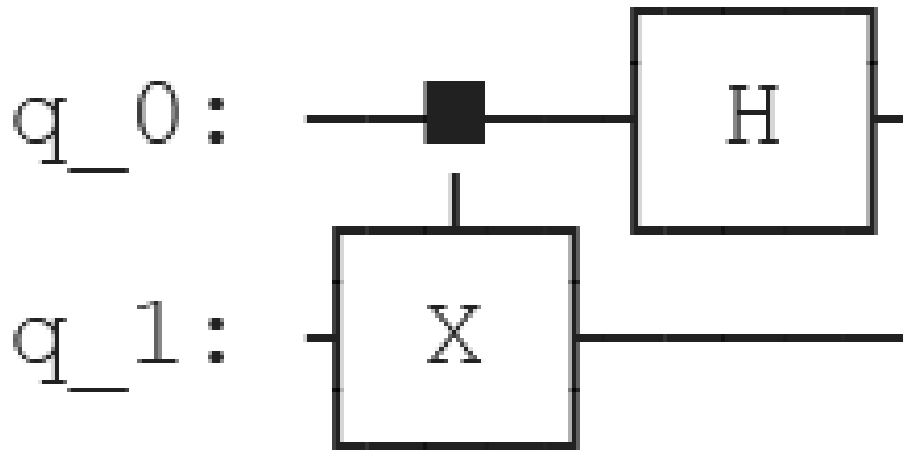


Figure 8: Example 2-qubit circuit

Example 3-qubit Unitary

```
three_qubit_circuit = qiskit.QuantumCircuit(3)
three_qubit_circuit.cx(0,1)
three_qubit_circuit.x(1)
three_qubit_circuit.cx(1,2)
three_qubit_circuit.draw()
```

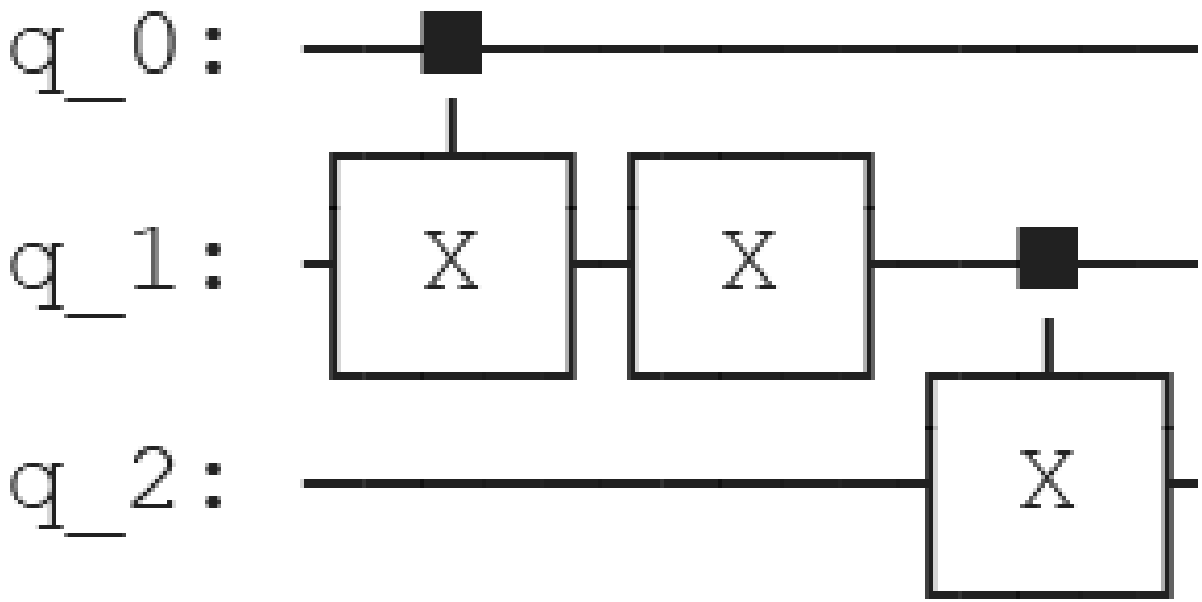


Figure 9: Example 3-qubit circuit

Update Frames

An *Update Frame* is a partition along with the unitary to run on it.

```
u1 = pqca.UpdateFrame(tes_1,
        qiskit_circuit=two_qubit_circuit)

u2 = pqca.UpdateFrame(tes_2,
        qiskit_circuit=three_qubit_circuit)
```

Automaton

A `pqca.Automaton` is

- An initial state
- An update step determined by a sequence of update frames
- A *backend* that simulates the circuit

```
init = [0]*9*4
back = pqca.backend.qiskit()
automaton = pqca.Automaton(init, [u1, u2], back)
```

The Update Circuit

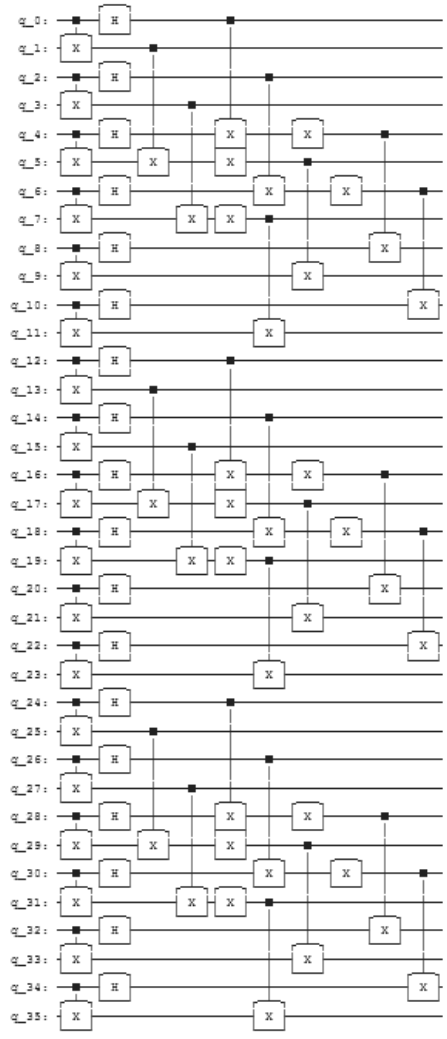


Figure 10: The tessellated circuit

Circuit Complexity

This allows us to build large circuits from tessellations of small circuits.

Measurement

In order to feed the results of the PQCA back into CAMUS we need to **measure** the states of the qubits after the update circuit.

Some results

Figure 11: The initial state

Some results

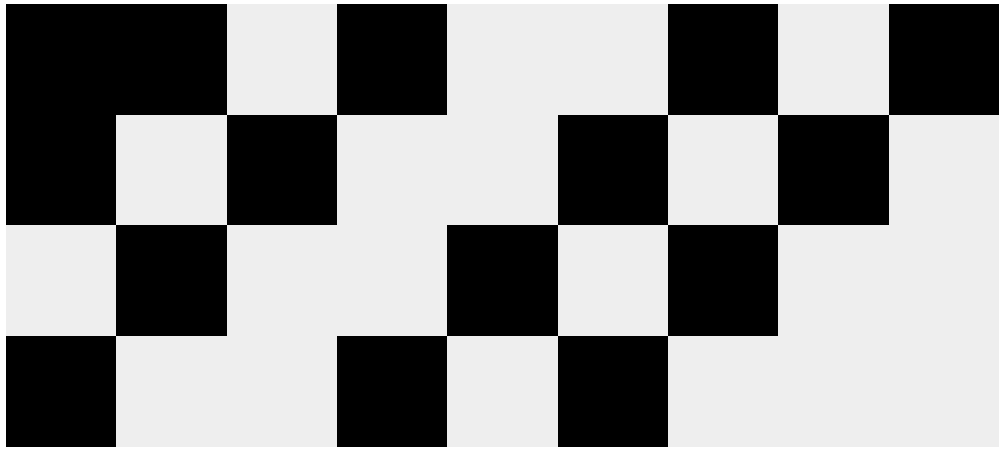


Figure 12: After the first step

Some results

Figure 13: After the second step

Some results

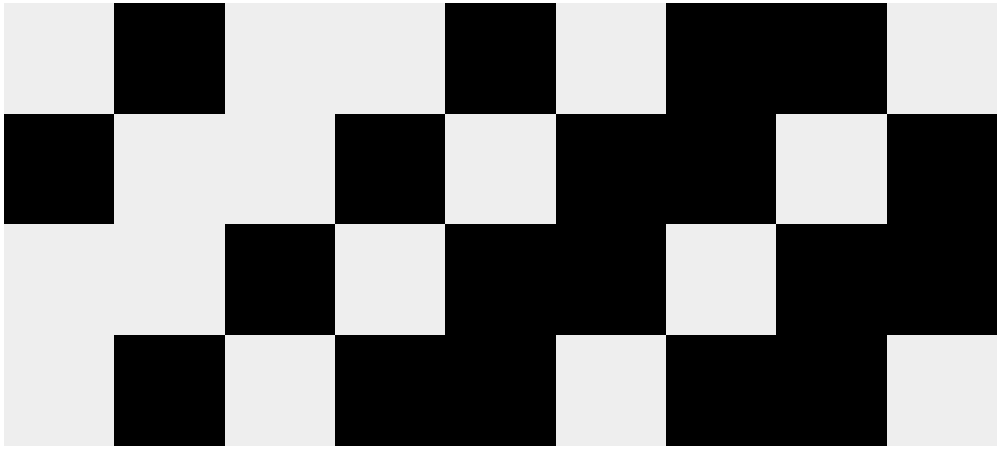


Figure 14: After the third step

Some results

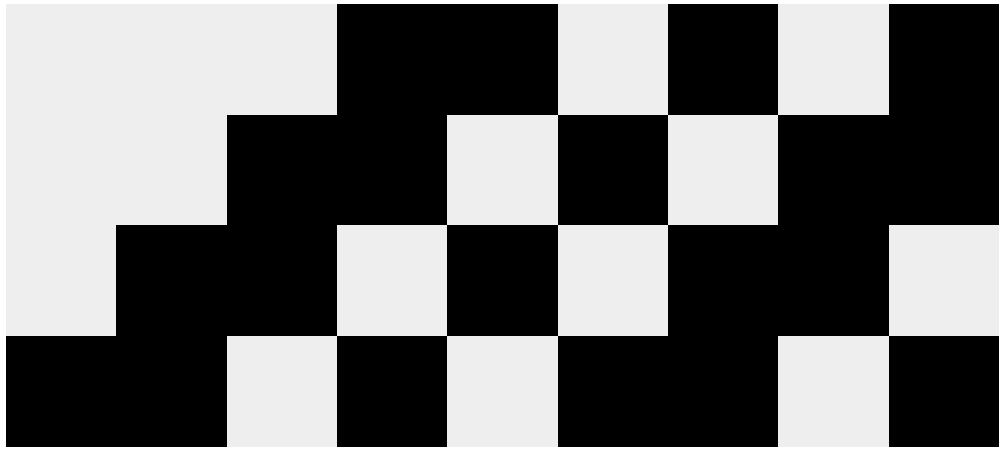


Figure 15: After the fourth step

Some results

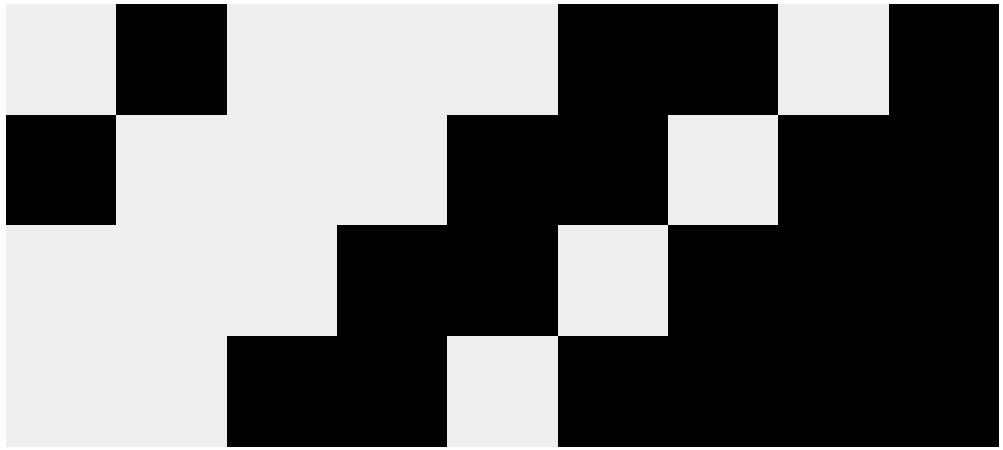


Figure 16: After the fifth step

Putting It Together

Small Scale CAMUS

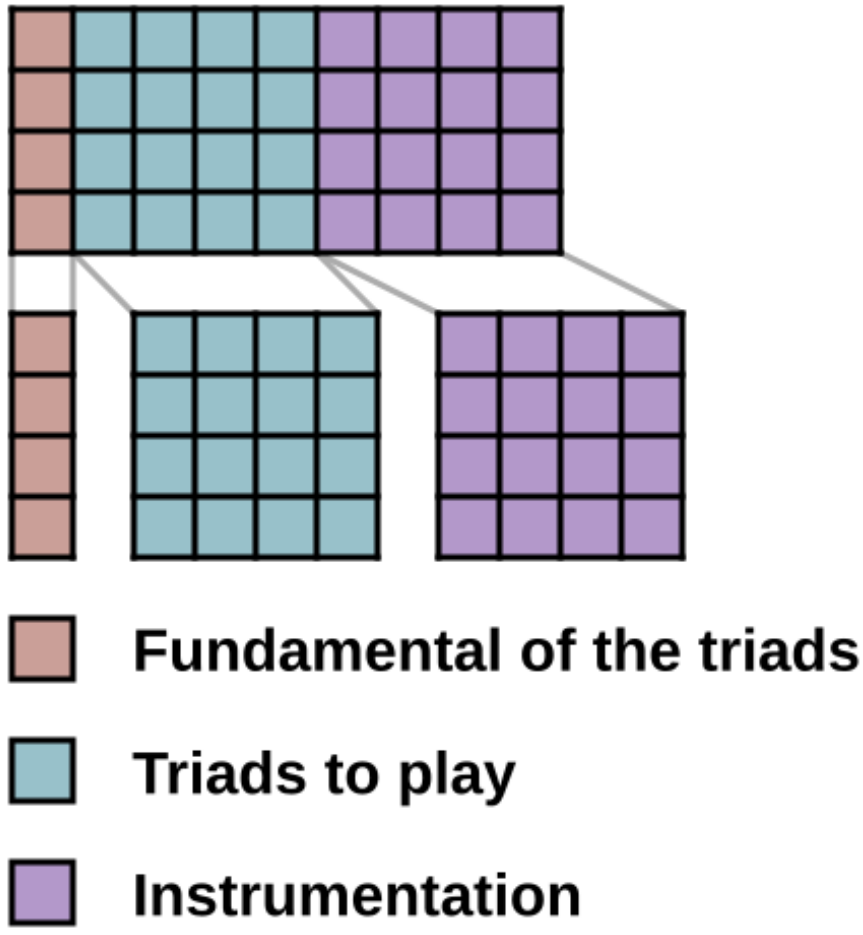


Figure 17: How we split our input grid for CAMUS

The Main Loop

Our main process loop is as follows

1. Prepare the state
2. Apply the update circuit
3. Measure the state
4. Feed the result into CAMUS
5. Feed the result back into step 1.

IBM Jakarta

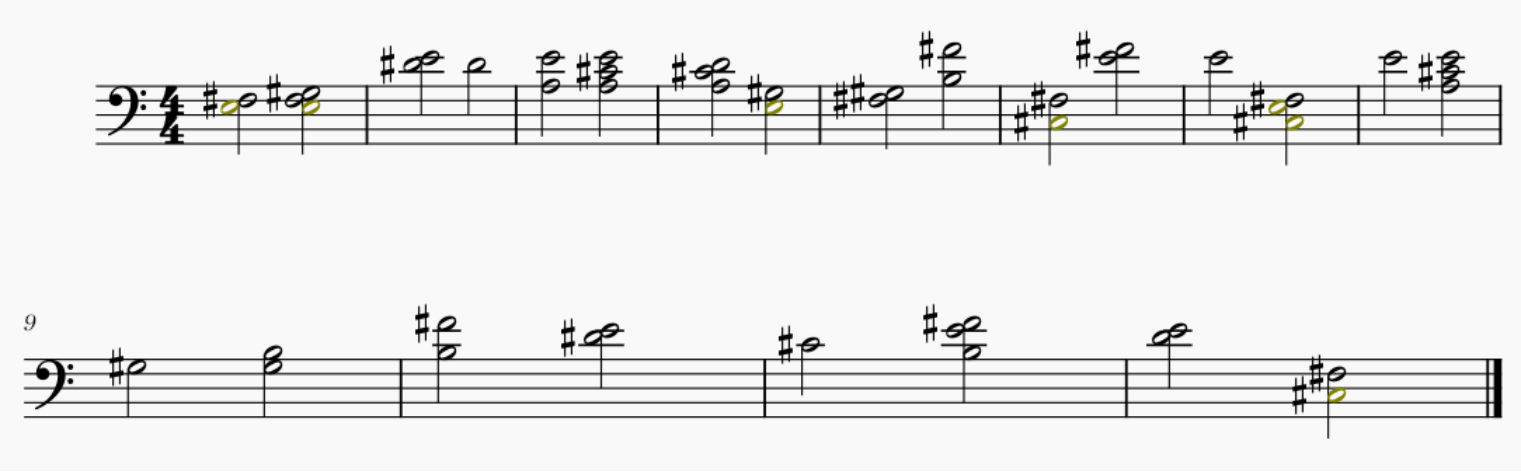


Figure 18: (Simplified) example of Quantum CAMUS run on IBM Jakarta

Conclusion

Our Goal

To facilitate musicians using quantum computers to aid composition

In order to synthesise music with this system, a musician:

- Specifies a collection of small quantum circuits and tessellations
- Makes some stylistic choices about how to implement CAMUS

Artefacts

Software

`pqca` is a Python package that handles Partitioned Quantum Cellular Automata

Tutorials

`https://iccmr-quantum.github.io/` hosts Jupyter notebooks to guide musicians through the process

Thank You

Acknowledgements

This work was funded by the QuTune project.

Tutorials are available on the ICCMR website.