# An exploration of local rules to map spawning processes to regular hardware architectures

Stewart Aitken

Master of Science (by research)

University of York

Computer Science

April 2022

# Abstract

This thesis presents an exploration of population growth via simulation in software to ascertain if a massively parallel hardware system can manage applications running within. Task execution happens dynamically and is controlled by the growth mechanism implementing efficient mapping in simulation.

Algorithms that provide population simulation models are often inspired by those evidenced in biology and in particular those of cellular automata and L-systems. These algorithms are of particular interest due to their complexity and self-replication and recent research has shown that it is the refinement of the biological methodology that has resulted in their complexity. Further to this, adaptation of the design has moved the algorithm on towards being able to organize and build itself from a single cell. A growth model is utilized in software systems to provide production of meaningful data. The development of bio-inspired software is constrained by using contemporary processor architectures.

# Contents

**abstract** ii

**1. Introduction** 1
1.1. Research Outcomes . . . 2
1.2. Thesis Structure . . . 3

**2. Field Survey and Review** 4
2.1. Computer hardware . . . 4
2.1.1. Basic architectures . . . 4
2.1.2. Multiprocessor Systems . . . 7
2.1.3. Topology, Mapping and the Spidergon Architecture . . . 11
2.1.4. FPGA Implementation . . . 16
2.2. Agent-based Modeling and Simulation . . . 20
2.2.1. Simulation Tools . . . 20
2.2.2. Mason Toolkit . . . 21
2.3. Algorithms under investigation . . . 22
2.3.1. Rewriting Systems . . . 23
2.3.2. L-systems . . . 24
2.3.3. Cellular Automata . . . 30
2.3.4. von Neumann's 29-state cellular automaton software simulations 51
2.4. Emergence . . . 56
2.4.1. Morphogenesis . . . 56
2.4.2. Embryonic Development . . . 58
2.4.3. Emergent engineering . . . 60

**3. Research Question** 62
3.1. Why . . . 62
3.2. Example and terminology . . . 62
3.3. How . . . 63
3.4. What . . . 63

**4. Simulation Methodology** 64
4.1. Agent-based Modeling and Simulation . . . 64
4.2. Simulation Grid Structure . . . 64
4.3. Neighbouring cell order disruption . . . 66
4.4. von Neumann neighbourhood - Tunnelling . . . 69
4.5. Constraints on grid height in built-in neighbourhood methods . . . 70
4.6. Random Number Generation . . . 71

**5. Simulation experiment using L-systems algorithm** **73**
5.1. Experiment 0 (a simulation using L-system algorithm.) . . . . . 73

**6. Simulation experiments that apply MASON Toolkit** **77**
6.1. Introduction . . . . . 77
6.2. Experiment 1 (Initial MASON investigation) . . . . . 79
6.2.1. Zig - Zag designated start position . . . . . 79
6.2.2. Zig - Zag random start position . . . . . 80
6.3. Experiment 2 (Single agent spawning). . . . . 81
6.3.1. Radial neighbourhood . . . . . 82
6.3.2. von Neumann neighbourhood . . . . . 84
6.3.3. Moore neighbourhood . . . . . 90
6.4. Experiment 3 (Multiple agent spawning). . . . . 94
6.4.1. Radial multi agent . . . . . 94
6.4.2. von Neumann multi agent . . . . . 96
6.4.3. Moore multi agent . . . . . 97
6.4.4. Selective Travel multi agent . . . . . 97
6.5. Experiment 4 (Multiple agent moving, growing and spawning). . . . . 100

**7. Conclusions and Future Work** **107**
7.1. Conclusions . . . . . 107
7.2. Future Work . . . . . 108

**A. JAVA Code.** **112**
A.1. MyObjectGrid2D class extension . . . . . 112
A.2. Code for Radial Neighbourhood . . . . . 113
A.3. Code for D0L Simple Model . . . . . 114

**B. Moore and von Neumann heat-maps.** **117**

**C. Other figures.** **120**
C.1. Radial Neighbourhood step by step . . . . . 120

**D. Google drive for media.** **122**
D.1. Google Drive link for Simulation films: . . . . . 122
D.2. Simulation film listing: . . . . . 122

**E. Cohens'*d* calculation** **124**
E.1. Statistical analysis of Radial local neighbourhood core occupancy . . . 124

**F. List of Acronyms** **127**
F.1. Acronyms . . . . . 127
F.1.1. Acronyms that cannot be linked . . . . . 128

**Bibliography** **129**

# List of Tables


2.1. Flynn's taxonomy, 1966. . . . . 5
2.2. One-dimensional Elementary Cellular Automata Transition Rules. . . . 33
2.3. Von Neumann's 29 states. . . . . 43
2.4. Cell State Transmission between Neighbours. . . . . 54
2.5. Encoding of the State Type. . . . . 55

4.1. Classical Moore neighbours location array for a hop of one, showing neighbours for an inside, corner and edge location. Bold numbers are the coordinates for the neighbour which has changed its selection order. 67
4.2. Moore neighbours array for a hop of one, origin at coordinates (2,2). The coordinates (3,3) in bold is the one that has changed position within the neighbour selection ordering. . . . . 68
4.3. von Neumann Neighbourhood Bounded 5x10 grid, origin at cell (2,5). . 72

5.1. D0L run output for seven generations. . . . . 76

6.1. Glossary of simulation terminology . . . . . 77
6.2. Radial neighbourhood, Bounded 5x5 Grid, Object Centre: (2,2):Radius 1. 84

7.1. Node to node router lookup table for FPGA 1 on Board A from figure 7.2, Out refers to routing beyond the local area network. . . . . 109
7.2. Node to node router lookup table for FPGA 1 on Board A from figure 7.3, Out refers to routing beyond the local area network. The number is the hop cost between two nodes, those that are directly connected also have a bullet point next to the number. . . . . 111

E.1. Random number generator seed, Odd and Even comparison . . . . . 126

# List of Figures


2.1. Spartan3 example taken from taught material IWC MSc, York. . . . . 9
2.2. A sample microprocessor systems-on-chips. . . . . 9
2.3. Interconnection network for the Ulysse1, internal and external custom processor[57, fig.3.3, p. 31]. . . . . 10
2.4. From [67], figure 3 on page 149: Comparison of spare core placement algorithm (a) and example CG, (b) 5x5 mesh NoC, (c) Proposed Mapping Technique, (d) FASA spare core placement. . . . . 12
2.5. Adapted from figure 7.2 on page 165 of [78]: Examples of regular network topologies. . . . . 13
2.6. Hybrid version network topology, Combination of 2D-Mesh and Spidergon. . . . . 14
2.7. Routing Sblock (R) to logic block (OR) shows (W) input matches (OR) output. [81, fig.1, p. 71]. . . . . 16
2.8. A simplified schematic of the system architecture. [40, fig.10, p. 5:12]. . 18
2.9. The Grid Lines Buffer's internal structure. [40, fig.16, p. 5:17]. . . . . 18
2.10. Generic architecture of an embryonic cell. [60, fig.1, p. 156]. . . . . 19
2.11. Relations between Chomsky classes of languages and language classes generated by L-systems. the symbols **OL** and **1L** denote language classes generated by context-free and context-sensitive L-systems, respectively. taken from, [65, fig. 1.2, p. 3]. . . . . 24
2.12. Example of a derivation in a D0L system. taken from, [65, fig. 1.3, p. 4]. 25
2.13. Development of a filament (*Anabaena catenula* ) simulated using a D0L system. taken from, [65, fig. 1.4, p. 5]. . . . . 26
2.14. (a) Turtle interpretation of string symbols. (b) Interpretation of a string. Angle increment $\delta$ is equal to 90 $\deg$. Initially the turtle faces up., taken from, [65, fig. 1.5, p. 7]. . . . . 27
2.15. Comparison of von Neumann model and 0L-systems model, taken from, [54, fig.1, p. 303]. . . . . 28
2.16. Some simple growing structures along with their CA interpretation. taken from, [75, fig.4, p. 200]. . . . . 29
2.17. Some simple branching structures along with their CA interpretation. taken from, [75, fig.5, p. 200]. . . . . 30
2.18. Productions used to obtain signal propagation, along with their CA interpretation. taken from, [75, fig.6, p. 201]. . . . . 30
2.19. Productions used to obtain signal divergence, along with their CA interpretation. taken from, [75, fig.7, p. 201]. . . . . 30
2.20. One-dimensional CA rule 62, Class 1 behaviour. . . . . 35
2.21. One-dimensional CA rule 56, Class 2 behaviour. . . . . 36
2.22. One-dimensional CA rule 30, Class 3 behaviour. . . . . 37

2.23. One-dimensional CA rule 110, Class 4 behaviour. . . . . 38
2.24. The Moore 9 cell neighbourhood at top left and The von Neumann 5 cell neighbourhood at bottom right for use with two dimensional CAs. . 38
2.25. Diagram of five triplets that do not fade on the first move, from an article by M.Gardner in the Scientific American, 223 (October 1970): 120-123. 39
2.26. Timeline from von Neumann to the present for self-reproduction in cellular automata. . . . . 40
2.27. von Neumann Universal Constructor taken from, [9] [fig. 27, p. 44]. . . 41
2.28. Confluent states progression modified from, [5, Fig. 6. p. 303]. . . . . 44
2.29. Ordinary transmission progression modified from, [5, Fig. 5. p. 303]. . 44
2.30. Sensitised tree progression taken from, [5, Fig. 7. p. 303]. . . . . 45
2.31. Destruction examples modified from, [5, Fig. 8. p. 303]. . . . . 45
2.32. Pulser (1101) modified from, [9, p. 147]. . . . . 46
2.33. Decoder (1101) modified from, [9, p. 148]. . . . . 46
2.34. Langton's loop showing stages of daughter construction taken from, [41, fig. 7, p. 141]. . . . . 46
2.35. Langton's loop showing subsequent steps after daughter completion taken from, [41, fig. 9, p. 143]. . . . . 47
2.36. Tempesti's loop taken from, [79, fig. 3, p. 2]. . . . . 49
2.37. Perrier's loop taken from, [62, fig. 4, p. 11]. . . . . 49
2.38. Universal construction loop using Tom Thumb algorithm taken from, [49, fig. 1, p. 180]. . . . . 50
2.39. Universal construction loop using Tom Thumb algorithm taken from, [49, fig. 10, p. 188]. . . . . 50
2.40. Signorini's data encoded binary values memory frame, [70, fig. 8, p. 180]. 51
2.41. Block diagram of a cell on the automaton, taken from, [70, fig. 11, p. 182]. 52
2.42. Alternative implementation of signal crossing in confluent element, adapted from, [63, fig. 11, p. 348]. . . . . 53
2.43. Block diagram of the "biodule", taken from, [5, fig. 11, p. 304]. . . . . 54
2.44. Ten bit encoding divided into three fields, taken from, [5, fig. 12, p. 304]. 54
2.45. FPGA content, taken from, [5, fig. 13, p. 305]. . . . . 55
2.46. Logic circuit, taken from, [5, fig. 14, p. 304]. . . . . 56
2.47. Example of morphogenesis in the game of life. . . . . 57
2.48. Design versus evolution spectrum [22, fig.8.1, p. 169]. . . . . 59
2.49. Gene regulatory interactions GRN [22, fig.8.3, p. 179]. . . . . 59
2.50. GRN subsequent positions [22, fig.8.3, p. 179]. . . . . 60
2.51. Simple Chaining [24, p. 13]. . . . . 60
2.52. Lattice formation by guided attachment. [24, p. 16]. . . . . 61

3.1. Simulation grids showing an empty grid and a grid with an active core. The cells with hatching are the local neighbourhood for a hop of one for both classic von Neumann and Moore. . . . . 63

4.1. Cells for all fixed start point simulations . . . . . 65

4.2. Numbered path of selected neighbouring cell occupied for each time step. The cell containing the red ring is the location that has stopped the simulation by having no empty cell within a hop of one. . . . . 66
4.3. von Neumann neighbourhood, full simulation . . . . . 70
4.4. von Neumann and Zig-Zag neighbourhoods, full simulation . . . . . 71

5.1. Development of a filament (*Anabaena catenula* ) simulated using a D0L system. taken from, [65, fig. 1.4, p. 5]. . . . . 74
5.2. UML Use case diagram for bacteria *Anabaena catenula* simulation. . . 75
5.3. UML activity diagram for two while loops, and an inner switch that forms the finite state automaton. . . . . 75

6.1. Zig-Zag neighbourhoods with designated start position. . . . . 80
6.2. Zig-Zag neighbourhoods with random start position. . . . . 81
6.3. Examples of certain cells falling within the region returned by the method getRadialLocations(...) with various settings of measurement rule (ALL, ANY, or CENTER) and closed-ness (open, closed). Figure from MASON Manual [45], page 123. . . . . 83
6.4. von Neumann neighbourhoods. . . . . 85
6.5. von Neumann neighbourhood, Occupied cells for each simulation grid version. . . . . 87
6.6. von Neumann neighbourhood, Occupied cells for each grid version. . . 88
6.7. von Neumann neighbourhood, Occupied cells for each grid version. . . 89
6.8. Moore neighbourhoods. . . . . 90
6.9. Moore neighbourhood, Occupied cells for each grid version. . . . . 91
6.10. Moore neighbourhood, Occupied cells for each grid version. . . . . 92
6.11. Moore neighbourhood, Occupied cells for each grid version. . . . . 93
6.12. Comparison of two Radial neighbourhood simulations, each with five agents and a hop of four. Figure 6.12a shows start positions for a random seed of three and figure 6.12c a random seed of four. The occupancy shown in figures 6.12b and 6.12d is quite different. . . . . 95
6.13. von Neumann neighbourhood simulation with three agents and the same randomised start position for all four different hop sizes. . . . . 96
6.14. von Neumann neighbourhood simulations with three agents and hops of one, two, four and eight when finished . . . . . 96
6.15. Moore neighbourhood simulation with five agents and the same randomised start position for all four different hop sizes. . . . . 98
6.16. Moore neighbourhood simulations with five agents and hops of one, two, four and eight when finished . . . . . 98
6.17. Selective travel neighbourhoods with ten agents and a hop of one. . . . 99
6.18. Moore neighbourhood simulations with twenty agents and a hop of four at different steps. . . . . 102
6.19. von Neumann neighbourhood simulations with twenty agents and a hop of four at different steps. . . . . 103

6.20. von Neumann local neighbourhood simulation of a many-core system. Depicting agents at various stages and core occupancy. Each local neighbourhood covers those cores within a hop of four. . . . . 106

7.1. Ethernet packet - physical layer, and Frame - data link layer. . . . . 108
7.2. SMBH custom board with 4 Spartan 3E FPGAs, von Neumann internal neighbourhood with Spidergon STNoC Architecture. . . . . 110
7.3. SMBH custom board with 4 Spartan 3E FPGAs, Moore internal neighbourhood with Spidergon STNoC Architecture. . . . . 110
7.4. 8 SMBH custom boards within local Intranet and external connection to Internet with Spidergon STNoC Architecture. . . . . 111

B.1. von Neumann neighbourhood, Occupied cells for each grid. . . . . 118
B.2. Moore neighbourhood, Occupied cells for each grid. . . . . 119

C.1. Simulation of a radial neighbourhood with a hop of four. Starting with five agents and increasing by a further five at each step. Random seed of 3 and limited cell occupation shown at step twenty. The cell at coordinates $(10, 0)$, coloured cyan with a black border is the one that is blocked from further movement and stops the whole simulation. . . . . 120

E.1. Boxplot of cell occupancy for odd random number seed versus even random number seed. Even random start positions lead to higher core occupancy than odd random start positions. . . . . 125

# Acknowledgement

Throughout the writing of this dissertation I have received a great deal of support and assistance.

I would like to thank both of my supervisors, Prof. Susan Stepney whose expertise was invaluable in the formulating of the research topic and guidance throughout each stage of the process, and Dr. Gianluca Tempesti for supporting the hardware thread and guiding the path we followed.

Thanks to the support staff of Computer Science, in particular Dean Welbourn for ensuring my departmental computer had the functionality that I required.

I would also like to thank Prof. Sean Luke, George Mason University for his assistance with the MASON Toolkit.

I would like to acknowledge Prof. Stephen Smith for preparing me for this project. I am thankful to the Electronics Engineering general office staff and in particular Emily Allcorn for looking after and ensuring my mobility scooter was always charged and ready for use. Travel on campus would have been impossible without this assistance.

Also, I would like to thank all of those on campus that I came in contact with for their support and encouragement throughout this journey. In particular those in the Special Cases Committee who supported my extended absence and permitting my return.

Last and not least, many thanks to my family for their encouragement and support, Cynthia my wife, my daughter Fiona, proofreader extraordinaire and her husband Mark. My grandson Stewart who is a constant source of joy.

## DEDICATION

This thesis is dedicated to my wife Cynthia who has supported me for a substantial period leading up to our golden wedding anniversary. It would have been impossible to finish this research without her care and considered support throughout

## DECLARATION

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

# 1. Introduction

Currently there is a problem of how computational tasks maps to multiple processors, for example many-core System on Chip (SoC) with Network on Chip (NoC) within a multiple Field Programmable Gate Array (FPGA) environment. The problem is the issue of task allocation and tasks spawning on multiple cores. NoC, SoC and FPGA‘s are examples of a general problem. We address a high level view of this problem in this thesis using these hardware environments as inspiring examples.

In search of the solution to the defined issue with task allocation, a distributed, bio-inspired approach is used, relying upon rule-based local neighbourhood algorithms within a two dimensional (2D) simulation. It is possible to develop algorithms for growth and movement that are both decentralised and dynamic for a single agent and multiple agents across an m * n grid. These algorithms will be applied to software processes which have the ability to grow, move and spawn further processes in a dynamic and discrete way. Each process will be applied to a single core with all processes operating in parallel. This will allow the question of how the size of the neighbourhood and complexity of neighbour selection affect the agent's growth pattern to be explored.

In the Field Survey and Review we focus upon several bio-inspired algorithms that may be applicable for direct implementation onto a FPGA. An examination of the developments involving application of such algorithms allows their adaptation in order to fulfil the cellular computation requirements of this work. This examination will also provide methods for implementation on embedded hardware and the Architecture Section is retained in the review so as to provide the selection basis for the simulations that have been carried out.

## 1.1. Research Outcomes

Local neighbourhood algorithms using the objective of task mapping to a many-core system are investigated by simulation and based upon the results in chapters 5 and 6 a hierarchy of of neighbourhoods is identified that are suitable, dependant upon circuit complexity for implementation in hardware. These neighbourhoods types are: von Neumann; classical, extended and user defined. Similarly Moore classical and extended and Radial extended. The complexity of each neighbourhood type using the concept that the simulation link between nodes is directly transposed to the circuit wires in a hardware implementation. Moore neighbourhood in figure 7.3 is obviously more complex than the von Neumann one in figure 7.2. We also show that fault tolerance can readily be implemented by placing a flag on any defective core preventing its use once identified.

## 1.2. Thesis Structure

Chapter 2 Field Survey and Review explores current research for task mapping and task spawning on many-core systems.

Chapter 3 Research Question is what is researched in this thesis.

Chapter 4 Simulation Methodology describes how the simulation tools are used to gather the research data.

Chapter 5 Simulation experiment using L-systems algorithm is preliminary research from a time when it was intended to implement this in hardware.

Chapter 6 Simulation experiments that apply MASON Toolkit is the main research area for this thesis

Chapter 7 Conclusions and Future Work discuses the results of this thesis and suggests areas worthy of future research.

Appendix A JAVA Code.

Appendix B Moore and von Neumann heat-maps.

Appendix C Other figures.

Appendix D Google shared drive for media and link to observe simulation films.

# 2. Field Survey and Review

In the Field Survey and Review we focus upon several Rewriting Systems algorithms, possibly from the bio-inspired area that may be applicable for direct implementation onto a FPGA. Examining what developments have been undertaken in application of these algorithms and identifying how they may assist us in developing them in to a form to suit our cellular computation requirements. Also we will examine methods of implementation on embedded hardware as justification for the final simulations that are undertaken.

## 2.1. Computer hardware

**Architectures**

For this review of processor architectures we take a high level view exploring the attributes of the system visible to the programmer such as the instruction set. This allows the identification of the architectures that are more suitable for parallel processing rather than sequential operation. We will draw upon some low level designs by Corporaal [14], Perrier [62] and Mudry [57] to assist in creating co-processors for multi-parallel operation.

### 2.1.1. Basic architectures

The bio-inspired algorithms when modelling growth creates very large data sets and needs a high amount of computational resources, they can also take a long time to process. However when parallel processing is used this can both reduce the time taken and improve the final outcomes. There are a number of architectures that could be consid-

ered for parallel processing and these are explored in the following section. Flynn's taxonomy [82] as shown in table 2.1 classifies computer architectures and are based upon the number of concurrent instruction and data streams available in the architecture.

Table 2.1.: Flynn's taxonomy, 1966.

| | Single instruction | Multiple instruction |
|---|---|---|
| Single data | SISD | MISD |
| Multiple data | SIMD | MIMD |

Single Instruction Single Data (SISD) stream [29] is the oldest computer type and no longer available.

Single Instruction Multiple Data (SIMD) stream which operates on a lockstep basis when given an instruction sequence which is applied to multiple items of distinct data at the same time then the calculations are completed in parallel . These are typical in operations carried out by a graphics processing unit. Lockstep causes all of the processors to synchronously operate on a global clock tick. When there is no data suitable for parallelisation then the processor operates sequentially and array processors are in this group [34].

Single program multiple data (SPMD) is a later addition to those of Flynn and was proposed by Darema [16] at IBM in 1984. Programs emphasize parallelism at the sub program level rather than at the instruction level. SPMD operates asynchronously by running the same program on different data using a MIMD machine.

Multiple Instruction Multiple Data (MIMD) stream has a set of processors which simultaneously execute different instruction sequences on different data sets at the same time. The MIMD may have a shared memory multi-processor or a distributed memory computer and involves multiple control units as well as multiple execution units [29, 34]. MIMD computers with shared memory are known as multiprocessors or tightly coupled machines, those with an interconnection network are known as multicomputers or loosely coupled machines[15, 17].

Multiple Instruction Single Data (MISD) is a type of parallel computing architecture where fault tolerant computers which execute the same instruction in parallel to detect any differences in the result. This model is not in general use.
Pipelining is a means of introducing parallelism into the essentially sequential nature of a machine instruction program. Each instruction is broken down into a fetch - execute cycle and steps through the pipeline, subsequent instructions are processed in turn but are only one step behind each other rather than a full set of steps. This allows the processor to maximise the instruction throughput. Distributed memory computer such as in a MIMD type is known as an interconnection network or distributed system, where each processor has its own private local memory with no shared memory. Data must be able to communicate between processors. Granularity has different definitions for parallel computing and reconfigurable computing. In the first instance, it means a qualitative measure of the ratio of computation to communication. *Coarse-grained*: large amounts of computation are done between data communication events. *Fine-grained*: individual tasks are relatively small in terms of code size and execution time. Data is transferred between processors in one or very few memory words. In the latter instance granularity refers to data path width. *Fine-grained*: in an FPGA the configurable logic block one bit wide processing elements and *Coarse-grained*: 32bit wide data paths such as used by a microprocessor CPU.

From this we can see that there are several main techniques to implementing parallel processing which can be implemented independently or combined:

- Multiprocessor system.
- Pipelining.

These architectures can also be based upon Multi-Processor Systems (MPSoC) which are discussed in the next section.

**Multi-core processor** A multi-core processor is a computer processor on a single integrated circuit with two or more separate processing units, called cores, each of which

reads and executes program instructions. The instructions are ordinarily CPU instructions, such as add, move data, and branch, but the single processor can run instructions on separate cores at the same time, increasing overall speed for programs that support multi-threading or other parallel computing techniques.

### 2.1.2. Multiprocessor Systems

System-on Chip (SoC), is an electronic circuit that implements most or all of the functions of complete electronic system. An MPSoC is a system-on-chip that contains multiple instruction-set processors Central processing Unit (CPU). Software design is an overall part of the chip design.

SoC constrains [35] that don't apply to general computing are;

- Must perform real-time computations.
- Must be area efficient.
- Must be energy efficient.
- Must provide the proper I/O connections.

SoC's can be compiled onto FPGA's as they have all of these constraints built-in. Implementation of SoC's onto FPGA's is a wide field of research but they are viable currently for custom logic.

#### Network on a chip

A network on a chip (NoC) is a network based communications subsystem on an integrated circuit, most typically between modules in a System on Chip (SoC).

Networks-on-Chips (NoC) use packet networks to interconnect the processors in the SoC. MPSoC's can be connected in large numbers over a network to create distributed systems and consideration should be given to security in the design process [35, 34, 29].

Given that the proposed hardware implementation is based upon four Xilinx Spartan 3 s4000 fg676 FPGA devices on an internal designed custom board then there are two options that can be implemented: MicroBlaze™processor or a custom processor.

Using a “black box” system of modeling we will progress through several stages. Firstly a single processor node on a single FPGA and then multiple processor nodes on a single FPGA, this can be regarded as being tightly coupled. Secondly the same steps but with a custom board containing four FPGA’s and lastly extending to other FPGAs in groups of four on separate custom boards connected by Ethernet. All of these latterly options are examples of being loosely coupled.

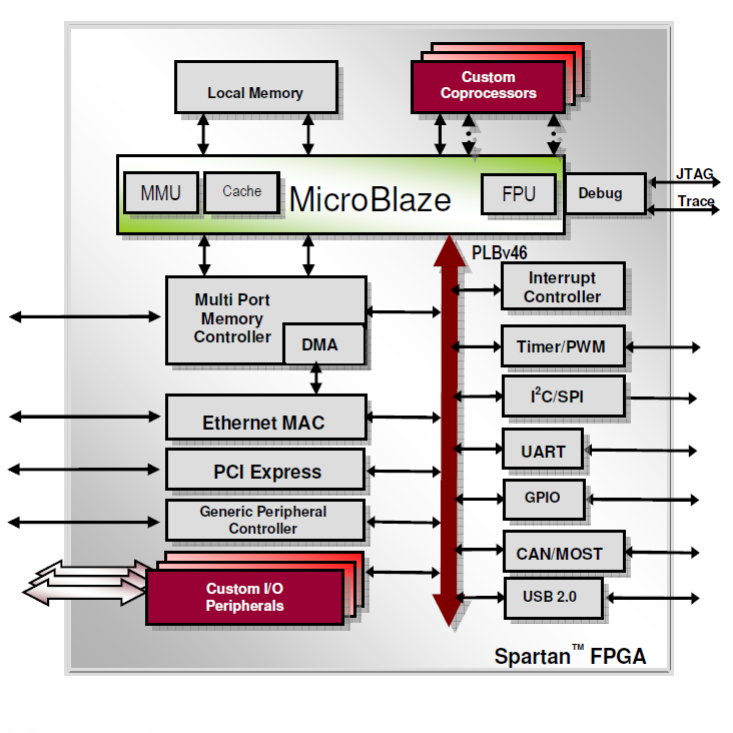
### Xilinx MicroBlaze™

The 32 bit MicroBlaze™ embedded processor soft core is a Reduced Instruction Set Computer (RISC) optimized for implementation on Xilinx® Field Programmable Gate Arrays (FPGAs) [31]. The Xilinx FPGA architecture can embed several MicroBlaze processors each of which takes up only about 2% of the total die area leaving plenty of room for implementing buses, hardware Intellectual Property (IP) cores and interfaces. Xilinx also provides a large IP library and useful development tools. Figure 2.1 contains an overview of MicroBlaze™ features and information of a particular Spartan3 system.

Smaller functional units used in large parallel operations may not require full processors, such as small state logic machines with registers or directly addressed memory or Digital Signal Processing (DSP) units. these can easily be accommodated on the FPGA. Figure 2.2 is a simple example of how an MPSoc can be built on an FPGA with the MicroBlaze™ processors acting as CPU’s using memory and large amounts of small parallel processing units.

### Custom Processor

In this section we consider two differing approaches to implementation of custom processors on relatively small FPGAs. Mazonka and Kolodin in their paper [55] consider




**MicroBlaze Processor System Architecture**

Figure 2.1.: Spartan3 example taken from taught material IWC MSc, York.

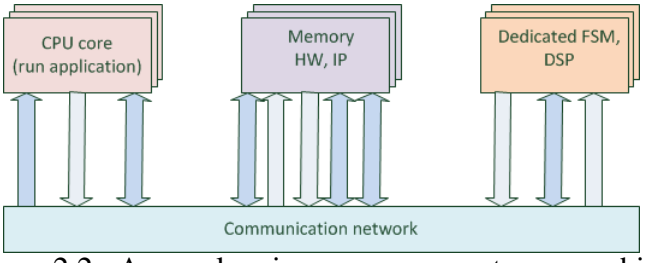



Figure 2.2.: A sample microprocessor systems-on-chips.

three styles of design, Transport Triggered Architecture (TTA) machines, Bit manipulating Machines (BMM) and Arithmetic Based Turing-Complete Machines with their choice for implementation being the last one due to the limitations of the Field Programmable Gate Array (FPGA). They chose to make a multi-processor which consists of twenty eight 32 bit processors.

Mudry [57], implements a processor targeted on the *cellular computing paradigm* built on a Xilinx Spartan3-XCS200 FPGA that follows on from Corporaal's microprocessor book [14] and develops architectures from a base of Very Long Instruction Word (VLIW) to TTA and the *MOVE32INT* processor. Figure 2.3 shows the intercon-

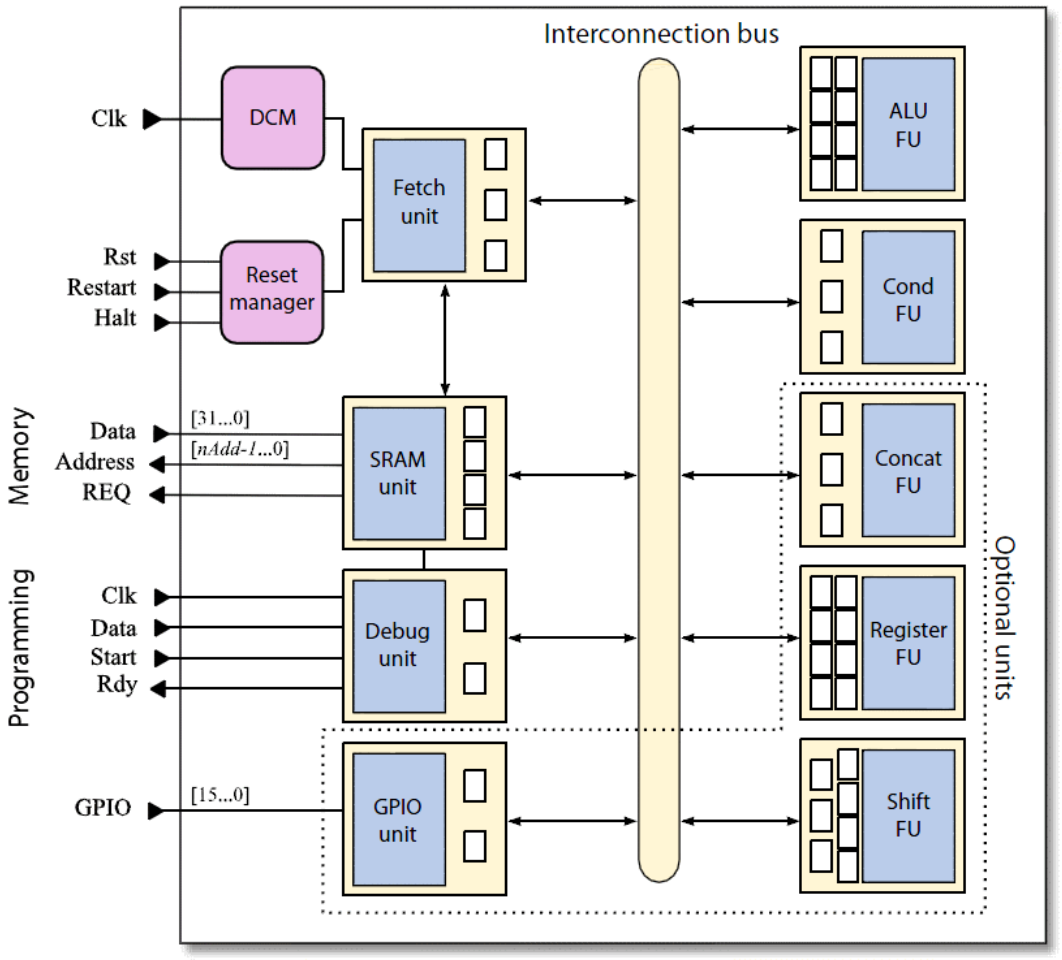



Figure 2.3.: Interconnection network for the Ulysse1, internal and external custom processor[57, fig.3.3, p. 31].

nection network (IN) for Mudry's processor which uses a shared bus topology on which Functional Units (FU) are plugged in. These functional units have a single functionality and common access via addressable locations with or without a register. The IN has three different buses for the source address, the destination address and the data being transported [57].

Both of these custom processors have areas of interest but are not suitable for our needs as they both require custom compilers to meet their programming needs. However the areas of interest are in the first case the connectivity of multiple processors on a single FPGA. Secondly the FUs, as we are basing our processor on cellular comput-

ing with a very large number of cells requiring parallel operation. The concept of for instance in an L-system model using FUs as finite state machines for each cell will require relatively small silicone space and thus allow a very large number of them to be implemented.

Tatas et.al. show a project utilising several MicroBlaze processors to create multi-core processing using FPGA's on pages 263 and 264 of [78] in chapter 11.6. A development board such as Spartan based is recommended for a complete lab experience of this project. The MicroBlaze processor will be used and can be connected to many different cores with further MicroBlaze processors connected together.

### 2.1.3. Topology, Mapping and the Spidergon Architecture

#### Task Mapping

Task mapping is the process of mapping software processes to processing cores in a many-core system. Tatas et al., [78] and Bonney [8]consider task mapping in relation to energy efficiency and fault tolerance applications. It is suggested that task mapping allows for the reservation of a small number of cores for fault tolerance and a larger number can assist with heat dissipation. In this research a small number of cores are used initially with an increase in usage of cores as the number of processes grows.

The development of many-core processors and NoCs realises increasing numbers of processors on a chip. Thus, consideration must be given to undertaking parallel computing and process mapping, routing algorithms are required to connect processors to each other.

**Task mapping** is the action of mapping software processes to processing cores contained within local neighbourhoods which are part of a many-core system. The many-core system will, upon start-up, initialise all cores into an empty ready state. Subsequently, mapping occurs in real time responding to the hardware status of the many-core system.

There has been a lot of research into mapping of multiple applications for fault tolerance on many-cores. Khalili and Zerandi [38] allocate a single spare core to each application during the mapping process. They then use a function to minimize the weighted Manhattan Distance between vertices which are then placed upon cores. The Manhattan Distance is the distance between two transmitting cores and the traffic volume between tasks located at the cores.

Reddy et.al. [67] also suggest a mapping algorithm using spare core allocation to different tile or vertices allocated to available cores. This is to address energy-efficient fault tolerance using spare core for mainly faulty cores. Their proposed technique where failure probability is calculated by Fault Aware Spare Allocation (FASA). Comparison of their proposed algorithm and spare core placement algorithm (FASA) is shown in figure 2.4. "Mapping simple Application Core graph (see figure 2.4a) onto a 5x5 grid (see figure 2.4b). Proposed fault aware core mapping technique (see figure 2.4c), and (see figure 2.4d) shows previous spare core placement." They observed that throughput increases for the proposed mapping in comparison with FASA.

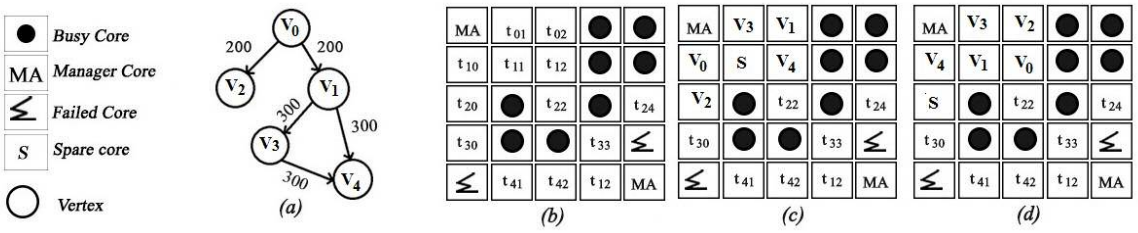



Figure 2.4.: From [67], figure 3 on page 149: Comparison of spare core placement algorithm (a) and example CG, (b) 5x5 mesh NoC, (c) Proposed Mapping Technique, (d) FASA spare core placement.

Tatas et.al. [78] suggest five key elements to a fault tolerant design: avoidance, detection, containment, isolation and recovery.

Bonney [8] considers reserving a small reserve of cores used for replacement from a faulty core, whilst at the same time using large amounts of unused cores as heat dissipaters from the active cores. This area of unused cores is described as "*dark silicon*"

and is about 20% of the chip area at any time. Bonney suggests that requirements for *dark silicon* and the need to provide fault tolerance by having a reserve of cores unused at any one time coincides.

### Topology

The topology is the first fundamental aspect of NoC design, and it has a profound effect on the overall network cost and performance. The topology determines the physical layout and connections between nodes and channels.

**Network topology**

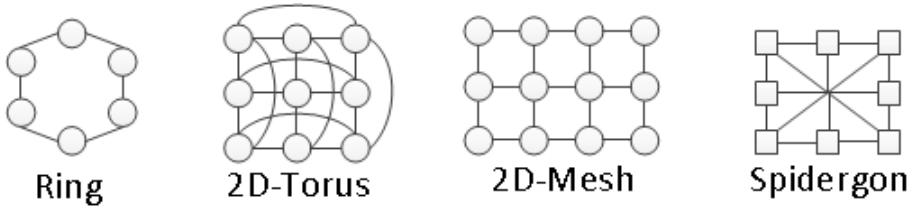



Figure 2.5.: Adapted from figure 7.2 on page 165 of [78]: Examples of regular network topologies.

Network topology is the topological structure of a network and may be depicted physically or logically. Network topology is the structure of the elements (links, nodes, etcetera.) of a network. It is an application of graph theory, wherein communication devices are modelled as nodes and the connections between the devices are modelled as links and are lines between the nodes. Physical topology is placement of the various components of a network (e.g., device location and cable installation), while logical topology illustrates how data flows within a network. Distances between nodes, physical interconnections, transmission rates or signal types may differ between two different networks, yet there logical topologies may be identical and networks' physical topology is a particular concern of the physical layer of the Open Systems Interconnection (OSI) model. Examples of networked topology are found in Local Area Network (LAN), a common computer network installation. Any given node in the LAN has one or more physical links to other devices in the network; graphically mapping these links results in

a geometric shape that can be used to describe the physical topology of the network. A wide variety of physical topologies have been used in LANs, including ring, bus, mesh and star. Conversely, mapping the data flow between the components determines the logical topology of the network in comparison, controller area networks, common in vehicles are primarily distributed control system networks of one or more controllers interconnected with sensors and actuators over, invariably a physical bus topology.

The 2D mesh topology in figure 2.5, provides a simple physical design where the links are assumed to be the same length. In addition, the grid shape simplifies the area required and as the number of nodes are increased the growth is almost linear. However, the 2D mesh topologies can exhibit congestion within the centre of the architecture and not at the edges because of particular routing algorithms used [78].

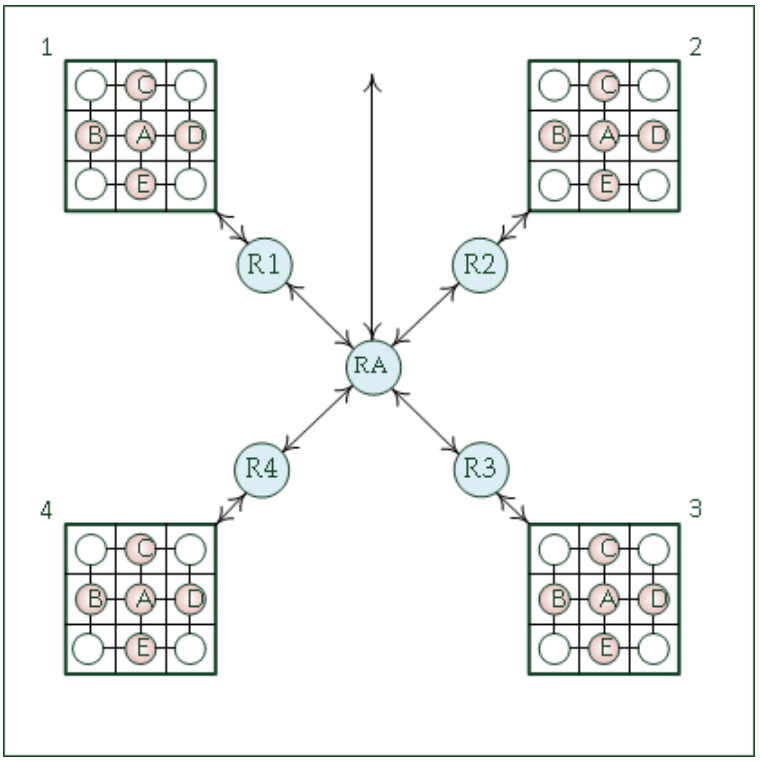


Figure 2.6.: Hybrid version network topology, Combination of 2D-Mesh and Spidergon.

**Mapping**

Requires real time task mapping [8].

**Node (networking)** The definition of a node depends on the network and protocol

layer referred to. A physical network node is an electronic device that is attached to a network, and is capable of creating, receiving, or transmitting information over a communication channel. A passive distribution point such as a distribution frame or patch panel is consequently not a node.

**Computer networks,** If the network in question is a local area network (LAN) or a Wide Area Network (WAN), every LAN or WAN node that participates on the data link layer must have a network address, typically one for each network interface controller it possesses. Equipment such as an Ethernet hub or modem with serial interface, that operate only below the data link layer does not require a network address. If the network in question is the Internet or an Intranet, many physical network nodes are host computers, also known as Internet nodes, identified by an IP address and all hosts are physical network notes. However, some data link layer devices such as switches, bridges and wireless access points do not have an IP host address, and are not considered to be Internet nodes or hosts, but are considered as physical network nodes and LAN nodes.

Improvements in the processors mean that they are operating at an increased frequency and are able to implement an architecture that exploits instruction level parallelism. These improvements are now reaching their limitations in their own complexity and the next refinement is to add thread level parallelism. This will be achieved through the use of a multi-stage process using the following developmental stages: [i] Simple software simulation, [ii] dynamic software simulation using various types of neighbourhoods.

In the future it is hoped to create a dedicated processor which will apply both instruction level and thread level parallelism via a black box model utilizing a Xilinx© Field Programmable Gate Array (FPGA). With this in mind, development of bio-inspired algorithms for growth and movement of single and or multiple agents which are seeded across an m * n grid of FPGA logic cells, both dynamically and scalable distributed, thus, there will not be a single primary controller.

### 2.1.4. FPGA Implementation

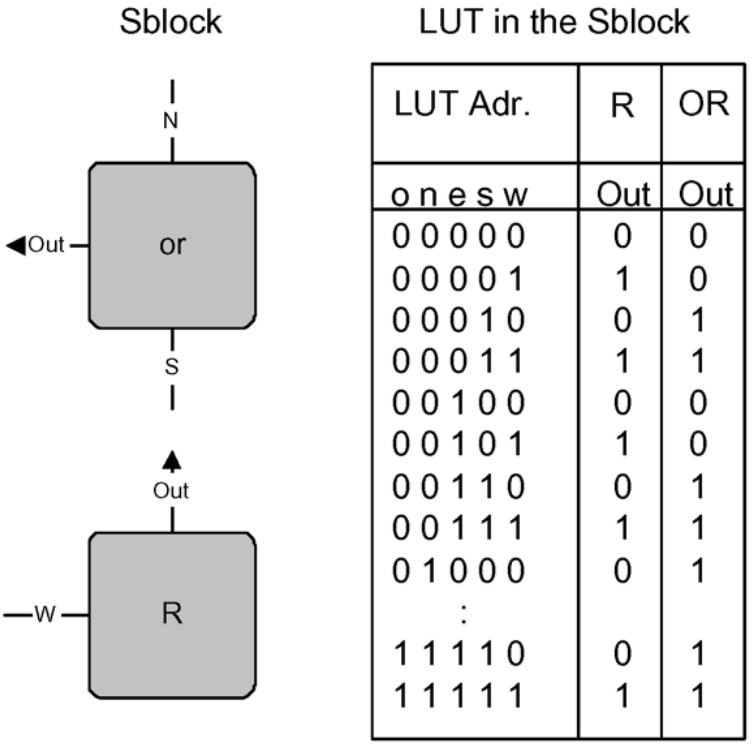



Figure 2.7.: Routing Sblock (R) to logic block (OR) shows (W) input matches (OR) output. [81, fig.1, p. 71].

Tufte and Haddow in [81] describe a Virtual Evolvable Hardware (VEHW) FPGA by mapping from their virtual technology to a physical FPGA, this can be considered to be a one-to-one mapping for the genotype-phenotype transition. The biological development is based upon L-systems which are a rule based artificial development system (see section 2.3). They have created an Sblock which can be considered as the equivalent of our FU which resides in a neighbourhood of four with the Sblock sides being the neighbourhood connection. These Sblock's have no external routing and they may be configured as a logic or memory component with direct connections to its four neighbours or as a routing element to non-local nodes thus allowing connections to further away nodes. Figure 2.7 shows Sblock (R) configured as routing and the logic block (OR) to the north of (R) outputs the (w) input, which can be confirmed by examination of the LUT. They go on to say that they require further research into a development

algorithm that can control pattern formation within the cells based upon the cell types.

In [39, 40], Kyparissas and Dollas investigate a new architecture for Cellular Automata (CA) simulation on FPGA technology, along with a framework to generate the architecture.

• The entire architecture is available as open-source under the Creative Commons Licence, and it can be found in:

`https://github.com/nkyparissas/Cellular_Automata_FPGA`.

This paper gives a review of these prior FPGA based accelerators, with a description of prior results and architectures.

- Toffoli and Margolus, Cellular Automata Machines (1984 - 2000)
- CEPRA: Cellular Processing Architecture (1994 - 2000)
- SPACE: Scalable Parallel Architecture for Concurrency Experiments (1996)
- Kobori, Maruyama and Hoshino: FPGA CA system (2001)
- Other significant work
  - Bouazza et al. ArMen Machine (1991)
  - Cappuccino and Cocorullo, CAREM (2001)
  - Murtaza, Hoekstra and Sloot: FPGA CA system (2007 - 2010)
  - Lima and Ferreira, CA Architecture (2013)

It goes on to describe a proposed architecture for large neighbourhood CA simulation on a medium sized FPGA. This system supports either 4-bit or 8-bit cells with neighbourhood sizes of up to 29 x 29 cells. Users of this system are only required to create the rule computations, possibly, by modifying an existing template. 'Every line that is loaded into the graphics controller buffer is also loaded into the CA Engine buffer'. The key variables are:

- $n$, $n$ x $n$ neighbourhood size, $n \in [3..29]$.
- $c$, cell size in bits, $c \in \{4, 8\}$ .
- $b$, memory burst size in bits, $b \in \mathbf{N}$.
- $x$, $y$, the grid dimension in cells, $x$, $y \in \mathbf{N}$.

'A CA is a discrete world with a discrete time, operating in distinct time steps'. All state changes must have occurred within a single time step and double buffering is used, with one copy of the current state and one copy of the next state.

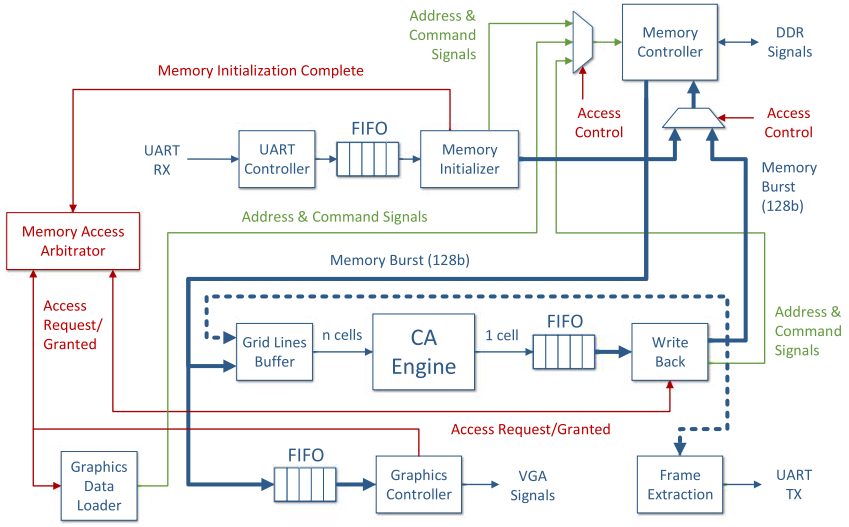



Figure 2.8.: A simplified schematic of the system architecture. [40, fig.10, p. 5:12].

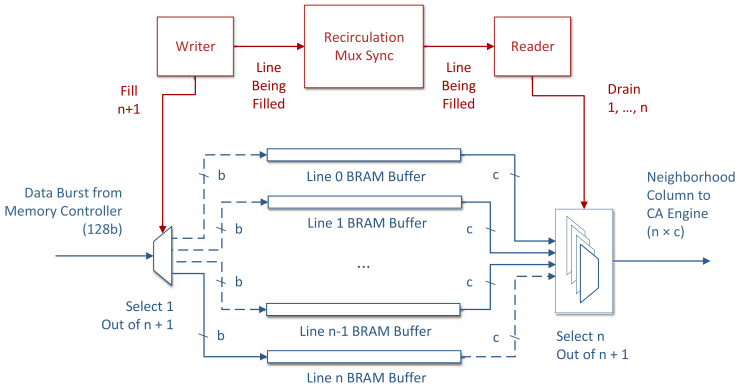



Figure 2.9.: The Grid Lines Buffer's internal structure. [40, fig.16, p. 5:17].

The grid lines buffer, figure 2.9 is designed on the ideas of Margolus pipeline buffer [51]. Which allows the CA Engine to produce a new cell per clock cycle if the neighbourhood cells arrive at clock rate.

" 'The grid lines buffer consists of $n$ Block RAM (BRAM) modules that store the grid lines needed to supply the CA Engine with the $n$ x $n$ neighbourhood of all the cells located in a grid line. Plus one BRAM module used as a write buffer. Thus the number of BRAM sources is equal to $(n + 1)$ x $x$ x $c$ bits = $(n + 1)$ x $b_l$ x $b$ bits. In the case described in the paper the most values for this architecture (corner case) are x = 1920 n = 29, c = 8.' "

Ortega and Tyrrell in [60] present a new version of the MUXTREE embryonic cell for implementation in a Virtex® from Xilinx™. The embryonic project takes biological processes and transports those universal mechanisms to the arena of electronic, programmable arrays. Figure 2.10 suggests the widespread structure of an embryonic array.

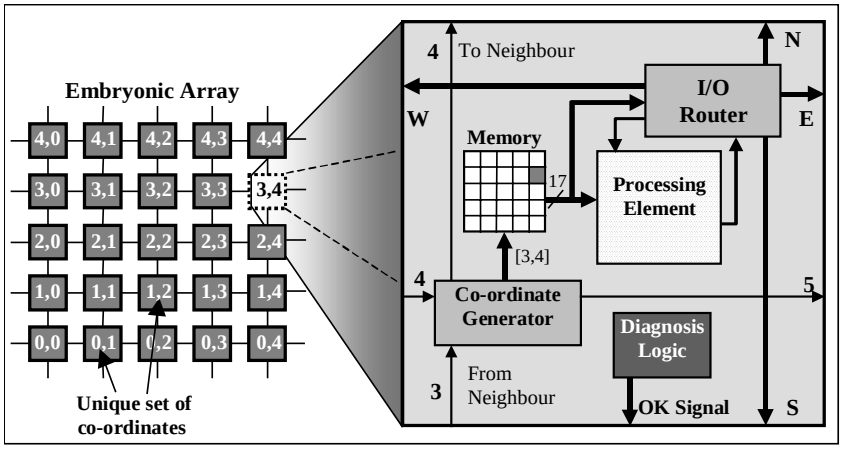



Figure 2.10.: Generic architecture of an embryonic cell. [60, fig.1, p. 156].

This version allows the implementation of complete arrays. The memory required in this version is much reduced over the generic embryonic cell by using a chromosomic way, storing only the configuration registers for all of the relevant cells. The result of this is that every cell is only required to store $(n + 1)$ registers for an $n$ x $n$ grid. It is also suggested that implementing the memory block as a look-up table results in an area

efficient circuit. Where there are reductions in the uses of resources this lends itself to releasing these for further functions.

## 2.2. Agent-based Modeling and Simulation

### 2.2.1. Simulation Tools

In Railsback et al., [66], five different platforms are reviewed including Agent Based Model Simulations (ABMS). MASON, Repast and Swarm are "framework and library" platforms this imparts a conceptual framework for setting up and designing ABMS and corresponding software program libraries. NetLogo is the highest-degree platform, presenting an easy but effective programming language, integrated graphical interfaces, and complete documentation. There are two versions of Swarm, Java Swarm and Objective-C Swarm.

Their conclusions for each of the ABMS are, that NetLogo, is most suitable for novice programmers but is more limited as the model complexity increases. Both Swarm versions have similar benefits and limitations such that they are not as good as MASON and Repast. For these last two there are pros and cons for each and the user should identify which is most suitable for their particular requirements.

A complete literature survey of the state of the art in software programs for agent based computing and its embodiment inside the modelling and simulation domain is provided by Abar et al., [1]. The importance of this survey is two-fold: (1) Highlighting the salient features, merits, and shortcomings of such multi-faceted utility software program; this text covers eighty five agent based toolkits that can help the device designers and builders with common tasks. (2) Provide a usable reference that aids engineers, researchers and academicians in effectively deciding on the precise agent based modelling and simulation toolkit for designing and growing their device prototypes, conscious of both their know-how in their utility domain. In essence, an extensive synthesis of Agent Based Modelling and Simulation (ABMS) assets has been executed so this evaluation

stimulates similar research into this topic.

One of the most fundamental attributes of the ABMS tool is its range; specifically the area where it is capable of performing modeling and simulation scenarios. ABMS is recognised in many scientific disciplines to simulate large-scale dynamic complex systems and observe emerging behaviours. These systems can be thought of simply as a collection of interacting actors or entities. The integrated development environment (IDE) is a standalone application programming environment with a typical code editor, compiler, tester/debugger, and an interactive graphical user interface generator (GUI). Typically, agent-based modeling (ABM) simulations involve processing a large number of agents (millions) that cannot be modelled in a single computer node due to memory problems. This implies that any ABM simulations would need to be run on a dedicated workstation or a high-performance parallel programming platform. Therefore, it is often necessary to run distributed simulations using a dedicated computing cluster or grid to reduce simulation time. In order to exploit the full potential of the ABMS model, researchers are carefully working on next-generation agent-based simulation test panels that can be extended into exascale computational structures. The authors [1] recommend that readers should investigate the ABMS tools further for greater in-depth analysis in line with their specific demands.

Based upon the ABMS requirements given in the introduction, and the ABMS scope or application domain in this paper [1], suggests that there are thirty eight different toolkits that meet the application domain criteria. However, when the other criteria are considered from this list there are only two possibilities MASON and Repast Simphony.

### 2.2.2. Mason Toolkit

MASON has been chosen as it meets all of the requirements of this project and it is in use by others within the research groups, which means that there is likely to provide a support pool available locally. MASON is described in [45], p.9 as a Multi-Agent Simulation of ‘Neighbourhoods and Networks’. The architectural layout is divided into

two sections, where the first section is the model and the second section is the visualisation. The architecture and visualisation can be generated in either 2D or 3D, however for our purposes we have only considered the 2D presentation. As these are both separate it is possible to have a simulation without the visualisation. In addition the model can be checkpointed, allowing the simulation to be paused and reused by saving the checkpoint to storage. The manual [45] has full documentation of the system including a 14-part tutorial. The ECJ owner's manual [44] is the documentation for an 'evolutionary computation framework written in Java'. This work is only referenced here because MASON uses the random number generator created by the ECJ referenced on p. 41.

Luke et. al., give a description of the history and current position for MASON, [43]. There are also many suggestions from the workshop in 2013 on how MASON could be revised and upgraded to benefit cutting edge research in the future. They are improving MASON by making it more robust and creating a distributed version that includes Geographical Information Science (GIS). [37].

## 2.3. Algorithms under investigation

The primary algorithms under consideration are those that fall within the Rewriting Systems field. Some but not all Cellular Automata (CA) are included in this area [33].

Rewriting systems are derived from formal grammars and consist of the formation of rules for strings in a formal language. The word "formal" refers to the fact that all the rules for the language are explicitly stated in terms of what strings of symbols can occur. A Lindenmayer system or L-system is a parallel rewriting system where each rule is applied to every symbol in parallel with repeated application. They can be used to describe plant development, fractal, and complex geometric design.

### 2.3.1. Rewriting Systems

Rewriting systems define complex objects by successively replacing parts of a simple object using a set of rewriting rules or productions. Among those most studied and understood are operations on character strings such as Chomsky's work on formal grammars mentioned below and Lindenmayer's L-systems discussed in 2.3.2. Rewriting systems are shown to cover a range of other types [18], such as strings, terms, graphs, constrained and parallel. Other classic examples of rewriting are the Koch snowflake curve and the array-rewriting system of Conway's game of life [65, 85, 28]. Also the rewriting research field covers two diverging directions, programming languages and automated deduction [18], which covers these topics, symbolic and algebraic computation, unification and matching and parallel deduction. A formal grammar is a system that produces a language which comprises a set of strings. The symbols used in the language are called the alphabet. For example $\Sigma$ = { a b c ... x y z ′ - } are the whole set of letters in the alphabet of the English language plus the apostrophe and hyphen. Languages can be defined to have strings or words of certain lengths. For example language $L_1$ can go from zero characters through to say length four characters and the empty word will be represented by $\Lambda$.

Language $L_1$ has words of length 0, 1, 2, 3, 4.

The empty word has zero characters equals $\Lambda$.

Thus if a word has four characters then length will be 4.

Language L will contain all of the words of any length possible for that language including the empty word and $L^+$ for all of the possible words not including the empty word.

A string is a sequence of characters, such as a word or phrase or some numbers and a string within a string is called a sub-string. The length of a string is the number of characters in it, such as **bab** which has length 3, the empty string has length 0 and

a single character is a one-character string. Concatenation consists of appending one string to another, for example **bab** and **aba** give **bababa**.

### 2.3.2. L-systems

Lindenmayer systems or L-systems for short are named after their creator Aristid Lindenmayer who was a Hungarian biologist, [65]. The string rewriting system grew out of "an attempt to describe the development of multicellular organisms in a manner which takes genetic, cytological and physiological observations into account in addition to purely morphological ones." [65, 75] The distinguishing characteristics of L-systems is to create complex objects by successively replacing parts of simple objects by applying rewriting rules or productions. Due to the biological motivations of L-systems productions are applied in parallel and replace all characters in a string simultaneously whereas in Chomsky grammars productions are applied sequentially [65, 59, 3].

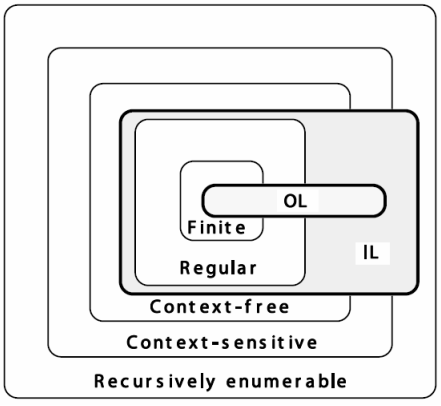



Figure 2.11.: Relations between Chomsky classes of languages and language classes generated by L-systems. the symbols **OL** and **1L** denote language classes generated by context-free and context-sensitive L-systems, respectively. taken from, [65, fig. 1.2, p. 3].

**D0L-systems** are the simplest class of L-systems the D0L comes from D for deterministic, 0 for zero and L-systems. Consider the simple example shown in figure 2.12 with the following explanation.

The alphabet $\Sigma$ = { a, b } is used by two productions or rules;

a $\rightarrow$ ab means that the letter a is replaced with the string ab,

and the rule b $\rightarrow$ a replaces the letter b with a letter a.

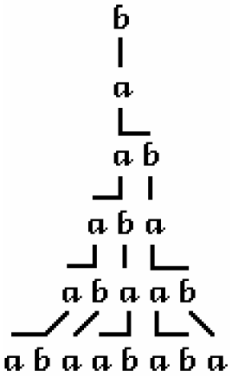


Figure 2.12.: Example of a derivation in a D0L system. taken from, [65, fig. 1.3, p. 4].

Starting at the top of figure 2.12 and working down, b is the starting string also known as the axiom and is rewritten by using production b $\rightarrow$ a . In the next cycle the production a $\rightarrow$ ab is applied. The third cycle is applied in parallel with both productions being used simultaneously replacing the a with ab and the b with a giving the resulting string of aba, this process can continue as long as wished. Two further cycles are shown in figure 2.12 .

In figure 2.12 the productions are context-free and apply regardless of context. In other cases where the production is dependant upon the predecessor's context then this is known as context-sensitive [74, 75, 76].

The relationship between Chomsky grammars and L-systems grammars can be seen in figure 2.11. It can also be seen in figure 2.11 that 0L-systems can generate some of the context-free and some of the context-sensitive languages whilst Chomsky context-free and or context-sensitive languages are mutually exclusive.

Figure 2.13 shows a further model of D0L-systems for the development of a "fragment of a multicellular filament such as that found in the blue-green bacteria *Anabaena catenula*" [65]. The symbols *a* and *b* represent cytological states of the cells. The subscripts *l* and *r* indicate cell polarity, specifying the positions in which daughter cells of type *a* and *b* will be produced. The development is described by the following L-system:

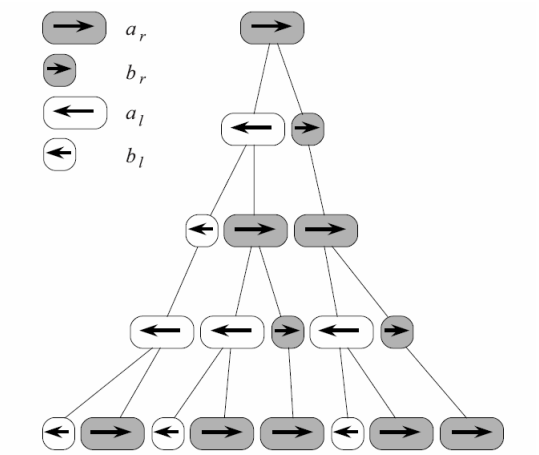


Figure 2.13.: Development of a filament (*Anabaena catenula* ) simulated using a D0L system. taken from, [65, fig. 1.4, p. 5].

$$\omega : \mathrm{a_r}$$

$$\mathcal{P}_1 : \mathrm{a_r} \rightarrow \mathrm{a_l b_r}$$

$$\mathcal{P}_2 : \mathrm{a_l} \rightarrow \mathrm{b_l a_r}$$

$$\mathcal{P}_3 : \mathrm{b_r} \rightarrow \mathrm{a_r}$$

$$\mathcal{P}_4 : \mathrm{b_l} \rightarrow \mathrm{a_l}$$

"Under a microscope, the filaments appear as a sequence of cylinders of various lengths, with *a*-type cells longer than *b* type cells. Figure 2.13 is the corresponding schematic image of filament development. Note due to the discrete nature of L-systems, the continuous growth of cells between subdivisions is not captured by this model," [65].

**Turtle interpretation of strings** Figure 2.13 is a very simple graphical representation that is not complex enough to model higher order plants, [65, p. 6], Prusinkiewicz developed just such a system based upon a LOGO-style turtle. The turtle state is a triplet $(x, y, \alpha)$ where $(x, y)$ are Cartesian coordinates representing the turtle's position, and angle $\alpha$, called the heading, is interpreted as the direction in which the turtle is

facing. Given step size $d$ and an angle increment $\delta$, the turtle can respond to commands represented by the following symbols:

F Move forward a step of length $d$. The state of the turtle changes to $(x', y', \alpha)$, where $x' = x + d\cos\alpha$ and $y' = y + d\sin\alpha$. A line segment between points $(x, y)$ and $(x', y')$ is drawn.

f Move forward a step of length $d$ without drawing a line.

\+ Turn left by angle $\delta$. The next state of the turtle is $(x, y, \alpha + \delta)$. The positive orientation of angles is counter-clockwise.

– Turn right by angle $\delta$. The next state of the turtle is $(x, y, \alpha - \delta)$.

Figure 2.14 is a graphical representation of the turtle rules given above [65, p. 7].

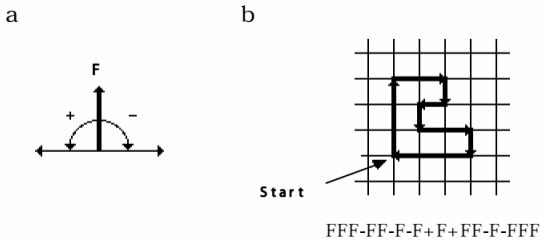



Figure 2.14.: (a) Turtle interpretation of string symbols. (b) Interpretation of a string. Angle increment $\delta$ is equal to 90 deg. Initially the turtle faces up., taken from, [65, fig. 1.5, p. 7].

**Context-sensitive L-systems** Several types of context-sensitive L-systems exist but we examine one with the production form $U < A > X \rightarrow DA$. Where the letter $A$ can produce the word $DA$ if and only if $A$ is preceded by the letter $U$ and followed by $X$ [75, p. 196]. Hence, letters $U$ and $X$ form the context of $A$ in this production. The strict predecessor can also have a one sided context such as to the left as shown in the example below, or to the right. This results in the formation of a growth function and can include rules for termination of a string.

The following 1L-system uses context to simulate signal propagation throughout a

string of symbols [65, p. 31]:

$$\omega : \text{baaaaa}$$

$$\mathcal{P}_1 : \text{b} < \text{a} \rightarrow \text{b}$$

$$\mathcal{P}_2 : \quad \text{b} \rightarrow \text{a}$$

The first few words generated by this L-systems are:

baaaaa

abaaaa

aabaaa

aaabaa

aaaaba

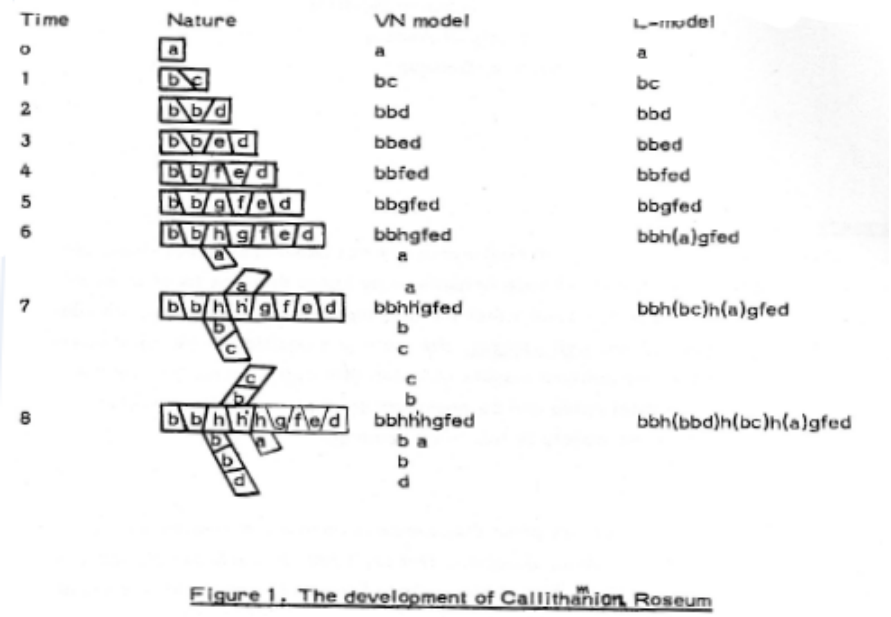



Figure 1. The development of Callithamnion Roseum

Figure 2.15.: Comparison of von Neumann model and 0L-systems model, taken from, [54, fig.1, p. 303].

The standard bracketed system uses square brackets to indicate when a branch is introduced, examples can be seen in ABOP [65, Figure 1.24, p. 25]. Both of these papers by Mayoh [54] and Stauffer [75] take a non-standard approach to brackets that represent branching. It is interesting to note the model comparison from Mayoh [54] shown in figure 2.15, where comparison is made between von Neumann and an 0L-system for the same red algae *Callithanion Roseum*. The model assumes that it occupies a 2-dimensional space and each cell has eight neighbours. Mayoh shows there are three disadvantages of the von Neumann model: growth and cell division can only occur when the cells are in a particular state; unwanted interaction because of the severe restrictions on growth direction; and inflexibility caused by having a fixed global limit on the number of neighbours a cell can have. The von Neumann model fails this time but if the Stauffer method shown below had been applied the the outcome would have been more favourable.

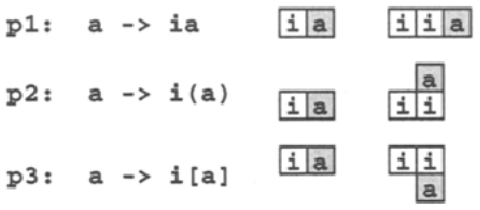



Figure 2.16.: Some simple growing structures along with their CA interpretation. taken from, [75, fig.4, p. 200].

Stauffer, now describes how L-systems can be used to describe cellular development. The L-system grows a one-dimensional string of characters and also can be translated into a one-, two-, three-dimensional image [75, p. 199]. "This developmental model comprises of four main processes: [1] simple growth, (fig. 2.16), [2] branching growth, (fig. 2.17), [3] signal propagation, (fig. 2.18), and [4] signal divergence (fig. 2.19)."

In figures 32-35 various parts of a tree can be represented and in a biological context has branch segments which are represented by the () and [] symbols, giving a left and right branch. The a is an apex and the i an internode, so plants can be fully modelled with left and right branches and left and right sub branches [75]. Figure 34-35 additionally

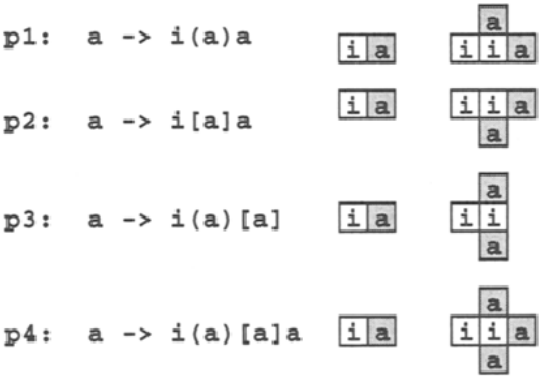



Figure 2.17.: Some simple branching structures along with their CA interpretation. taken from, [75, fig.5, p. 200].

show signal propagation and signal diversion where signal s changes an internode i on the right to an s. In the divergent phase it is reminiscent of Langton, Stauffer and Sipper, of a change of direction at the loop corner. [47, 75]


Figure 2.18.: Productions used to obtain signal propagation, along with their CA interpretation. taken from, [75, fig.6, p. 201].

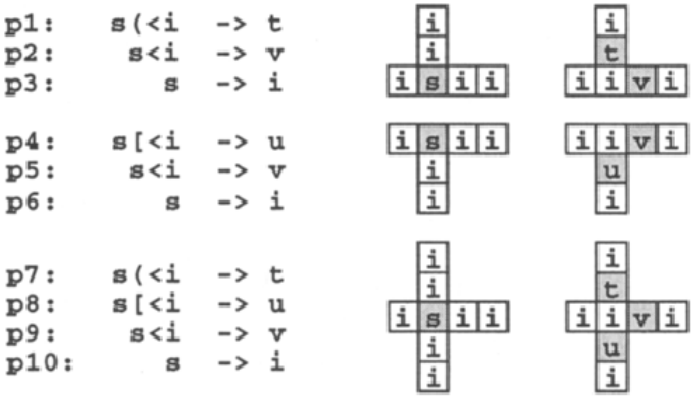



Figure 2.19.: Productions used to obtain signal divergence, along with their CA interpretation. taken from, [75, fig.7, p. 201].

### 2.3.3. Cellular Automata

Cellular Automata (CA) were known as tessellation structures and iterative circuit computers [9, p. xi] before Ulam suggested to von Neumann that the area that he, von Neu-

mann called automaton was best described as CA in [7, p. 86], and [6, p. 232], also [9]. Since then CA's have been used for simulations in biology, computer science and physics with the purpose of complex computations, parallelism of multiple processors and complex natural processes. Cellular automata can also be considered as rewriting systems on rectangular arrays and matrices.

A CA has a finite or infinite d-dimensional grid of cells. One, two and three-dimensional cases are the most commonly used [36] but in this review the focus is on one- and two dimensional CAs [61, 85, 86, 88]. Each cell can have a value from a finite set of states. A neighbourhood relation is defined over the grid, which shows which neighbours, affect each individual cell. The value of a cell at time t is based upon the values of the cell and relevant neighbours at t-1 and subsequent application of the transition rules [69, 86]. All of the cells are updated synchronously according to the relevant transition rules. The following formal definition comes from [72, p. 50-52]. Hence a cellular automaton $\boldsymbol{A}$ is a quadruple

$$\boldsymbol{A} = (\boldsymbol{S}, \boldsymbol{G}, \boldsymbol{d}, \boldsymbol{f}),$$

where $\boldsymbol{S}$ is a finite set of states, $G$ is the cellular neighbourhood, $\boldsymbol{d} \in \mathrm{Z}^{+}$ is the dimension of $\boldsymbol{A}$, and $\boldsymbol{f}$ is the local transition function.

Given the position of a cell $\mathbf{i}$, $\mathbf{i} \in \mathrm{Z}^{\mathrm{d}}$, in a regular $d$ - dimensional uniform grid, its neighbourhood $\mathbf{G}$ is defined by:

$$\boldsymbol{G} = \{\mathbf{i}, \mathbf{i} + \mathbf{r}_1, \mathbf{i} + \mathbf{r}_2, ..., \mathbf{i} + \mathbf{r}_n\},$$

where $n$ is a fixed parameter that determines the size of the neighbourhood and $\mathbf{r}_j$ is a fixed vector in the $d$ - dimensional space.

The local transition rule $f$

$$f : S^{\mathrm{n}} \rightarrow S$$

maps the state $s_i \in S$ of a given cell **i** into another state from the set $S$, as a function of the states of the cells in neighbourhood $\boldsymbol{G_i}$. In uniform CAs $f$ is identical for all cells, whereas in non-uniform ones $f$ may differ between different cells, i.e., $f$ depends on **i**, $f_1$.

For a finite size CA of size $N$ a configuration of the grid at time $t$ is defined as

$$C(t) = (s_{\mathbf{0}}(t), s_{\mathbf{1}}(t), ..., s_{\mathbf{N-1}}(t)),$$

where $s_i(t) \in S$ is the state of cell **i** at time $t$ . The progression of the CA in time is then given by the iteration of the global mapping $F$

$$F : C(t) \rightarrow C(t+1), \qquad t = 0, 1, ...$$

through the simultaneous application in each cell of the local transition rule $f$.

### One-dimensional CA

The simplest class of one-dimensional CAs is a special case named elementary CA [84, p. 8] with two possible states per cell, $S = \{0, 1\}$. The cells form a single horizontal row in the panel and after each generation the new row is formed on the top of the last row. A history of generations can be seen showing however many generations have occurred. Then $f$ is a function and the neighbourhood size $n$ is usually taken to be $n = 2r + 1$ such that:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}, \qquad s_i(t+1) = f(s_{i-r}(t), ..., s_i(t), ..., s_{i+r}(t)),$$

where $r \in Z^+$ is a parameter known as the radius, representing the cellular neighbourhood and $r = 1$ which gives a neighbourhood size of $n = 3$, [84, p. 8], [72, p. 52]

$$f : \{0, 1\}^3 \rightarrow \{0, 1\}, \qquad s_i(t+1) = f(s_{i-1}(t), s_i(t), s_{i+1}(t)).$$

The domain of $f$ is the set of all $2^3$ tuples which are indicated in the three state truth table 2.2, on page 33, which shows what the next states are for each of the three cell neighbourhoods of transition rules 30 and 110 [84].

Table 2.2.: One-dimensional Elementary Cellular Automata Transition Rules.

| 3 Cell Neighbourhood | | | Rule 30 | Rule 110 | |
|---|---|---|---|---|---|
| Start state | | | Next state | Next state | Decimal |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 2 |
| 0 | 1 | 0 | 1 | 1 | 4 |
| 0 | 1 | 1 | 1 | 1 | 8 |
| 1 | 0 | 0 | 1 | 0 | 16 |
| 1 | 0 | 1 | 0 | 1 | 32 |
| 1 | 1 | 0 | 0 | 1 | 64 |
| 1 | 1 | 1 | 0 | 0 | 128 |
| Right | Centre | Left | 30 | 110 | |

As can be seen from table 2.2 each elementary rule is specified by an eight bit sequence, $2^8 = 256$ different elementary CA's. The grid boundaries are typically calculated in one of two ways, either the leftmost neighbour of the left hand end cell on a row is the rightmost end cell of that row, giving a wrap around effect based upon Modulo N. Rule 30 in binary is $(00011110)_b$, hence if the row of cells comprises of:

11001010110

then the left end cell will be changed based upon neighbours $(0, 1, 1)$ giving new state 1 and the right hand end will use neighbours $(1, 0, 1)$ which gives new state 0. Thus at t+1 the new states will be:

10111010100

alternatively an additional cell is added to the beginning and end of the row which always holds a zero value then the first and last cells counted within the grid only use two cell values that are changeable rather than three for the remainder of the row.

Empirical investigations of CA's as dynamical systems were carried out by Wolfram

[85, 88, p. 413-423] and after many experiments four distinct classes of CA behaviour has been identified. It should be noted in the first papers covering these four classes Wolfram [84, 85] identified 32 of the 256 rules as being legal rules. These rules are identified by the Wolfram code which is the 8 bit binary number of the rule. The legal rules have amongst the other rules copies and mirror images of themselves which is why these other rules have been discounted. However further investigations have been carried out and some of the non-legal rules have been found to have interesting properties which has resulted in there being 88 possible unique elementary cellular automata, one of which, rule 110, is now included in Class IV behaviour [12]. Additionally rule 110 is known as the simplest Turing machine and supports universal computation [88, p. 575-577][13]. In each space-time diagram the horizontal rows are consecutive configurations with the top row being the initial configuration.

Class I Cellular Automata evolve to a homogeneous state after a finite number of time steps. The transaction rule and graphical output as a space-time diagram is shown in figure 2.20.

Class II Cellular Automata evolve to short period structures. The transaction rule and graphical output as a space-time diagram is shown in figure 2.21.

Class III Cellular Automata evolve to a chaotic aperiodic pattern, The transaction rule and graphical output as a space-time diagram is shown in figure 2.22.

Class IV Cellular Automata evolve to complex localised structures, sometimes long lived. The transaction rule and graphical output as a space-time diagram is shown in figure 2.23.

Wolfram states in [87], "The simple structures generated by class 2 CA's are stable or periodic with typically small periods and can act as filters." They also correspond to a set of words in a regular grammar. Whilst in class 3 they are aperiodic chaotic patterns but there may be small groups of sites that can be considered *defects* and execute random walks. One such example is rule 30 that provides the random number generator in

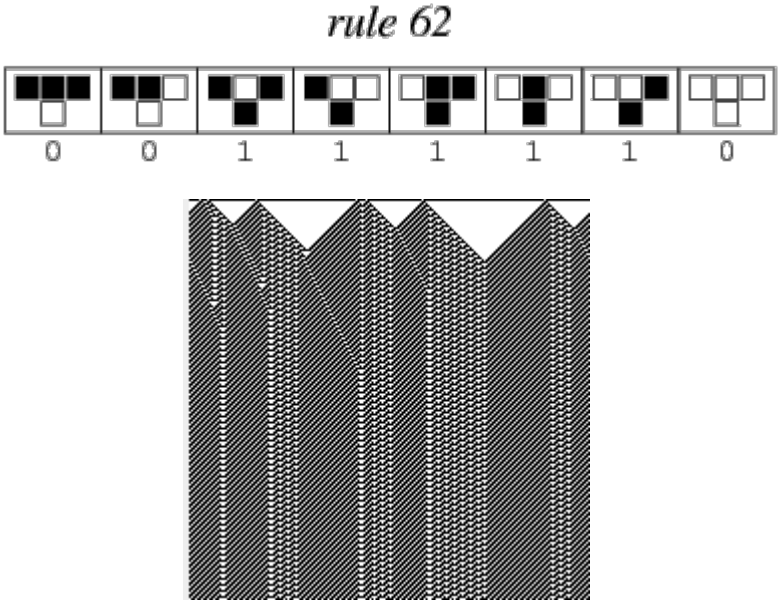


Figure 2.20.: One-dimensional CA rule 62, Class 1 behaviour.

Mathematica. In the case of class 1 CA's all initial states yield the same final state. Gosper's, unpublished work cited by Wolfram [87] speculates that class 4 CA's have the capability for universal computation and that this applies to the two dimensional CA known as 'Game of Life'. The final problem with Wolfram's classification systems is that the class cannot be predicted before [88] implementation of the rule algorithm, but rather, the rule must be run and then the diagrammatic output be assessed for property matching to a particular class.[9, 36, 50, 72, 84, 88]

### Two-dimensional CA

In a CA of two dimensions there are primarily three shapes for the cells within the grid that can be used, square, triangular and hexagonal. Figure 2.24 shows a representation of both the 5 cell and 9 cell neighbourhood used in a two dimensional square shaped cell grid CA [61]. Each cell can have k states, typically k = 2 and a neighbourhood of r = 1 which gives the general neighbourhood case as the 9 cell neighbourhood. On some occasions the neighbourhood can comprise of more than the immediate neighbours with r = 2 or more. The 5 cell version is a special case and it can be seen that by overlapping the 5 cell onto the 9 cell neighbourhood the 5 cell fits within the 9 cell boundaries on

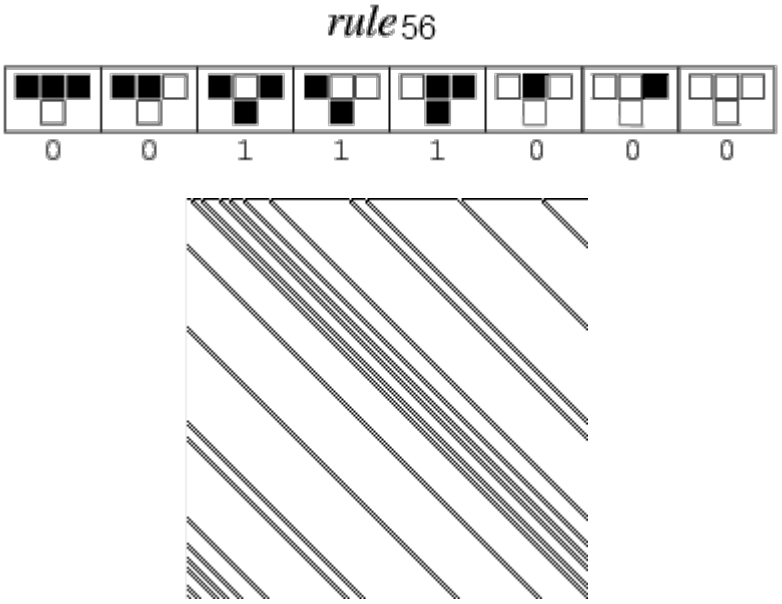


Figure 2.21.: One-dimensional CA rule 56, Class 2 behaviour.

figure 2.24. In figure 2.24 the dark central square with the white C in each pattern is the one that is updated and the shaded squares make-up the surrounding neighbours [50, 61].

The Game of Life is Conway's simplification of von Neumann's two dimensional cellular automaton based upon the square grid and was initially a board game similar to 'Go'. It is also of interest to a wide range of research fields such as; computer scientists; biologists; mathematicians; physicists; and others looking at complex systems emerging from simple rules [28, 2, 52, 50, 4]. The criteria to be met for Conway's rules are: [26]

1. There should be no explosive growth.
2. There should exist small initial patterns with chaotic, unpredictable outcomes.
3. There should be potential for von Neumann universal constructors.
4. The rules should be as simple as possible.

The standard rules for Game of Life are that a cell can be alive or dead. $K_0$ = white and dead and $K_1$ = black and alive. Each cell has the same rule for cell updating. As the cellular automaton evolves, emergent computation begins to appear [77, 28]. The

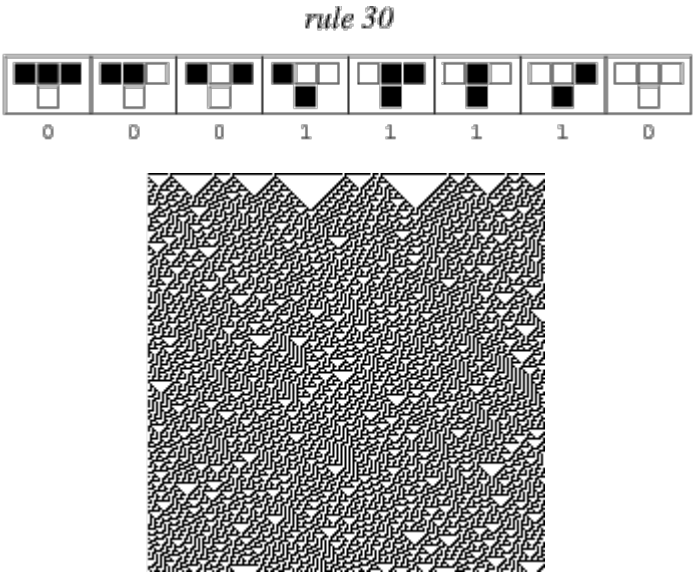


Figure 2.22.: One-dimensional CA rule 30, Class 3 behaviour.

rule is that every cell with two or three living neighbouring cells survives to the next generation. For deaths there are two conditions, four or more living neighbours then the cell dies from overpopulation and one or less living neighbours the cell also dies from isolation. Cells become alive with exactly three neighbours.

The patterns in figure 2.25 show two of the three main types of pattern occurring; still life's such as block **d**; oscillators like the blinker **e** with period 2; and spaceships that travel across the board,[2] not shown. Also shown are the three cell arrangements that do not die in the first generation **a, b, c** but do die on the second generation.

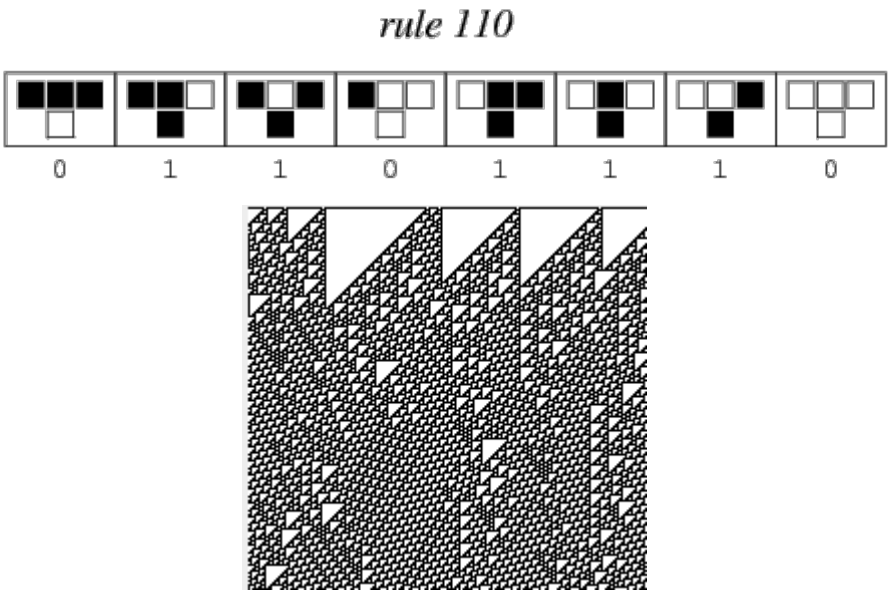



Figure 2.23.: One-dimensional CA rule 110, Class 4 behaviour.

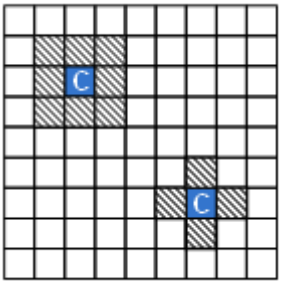



Figure 2.24.: The Moore 9 cell neighbourhood at top left and The von Neumann 5 cell neighbourhood at bottom right for use with two dimensional CAs.

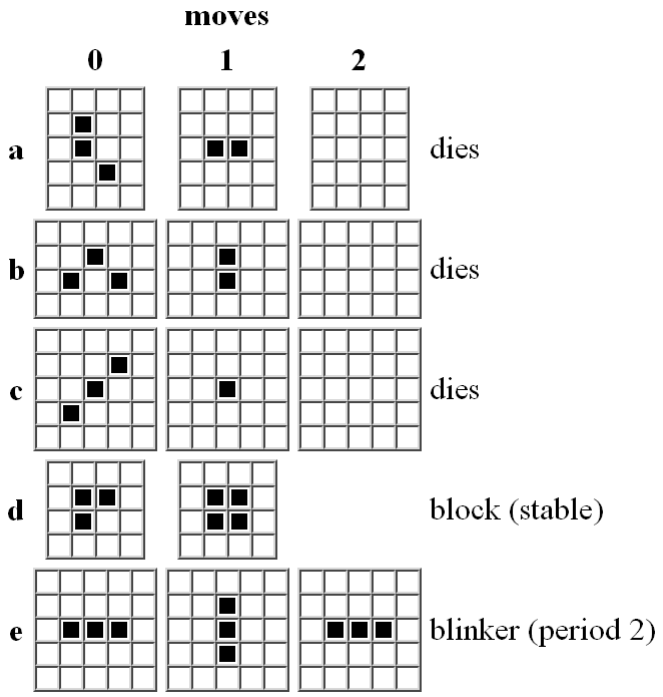



Figure 2.25.: Diagram of five triplets that do not fade on the first move, from an article by M.Gardner in the Scientific American, 223 (October 1970): 120-123.

### Self-Replication in Cellular Automata

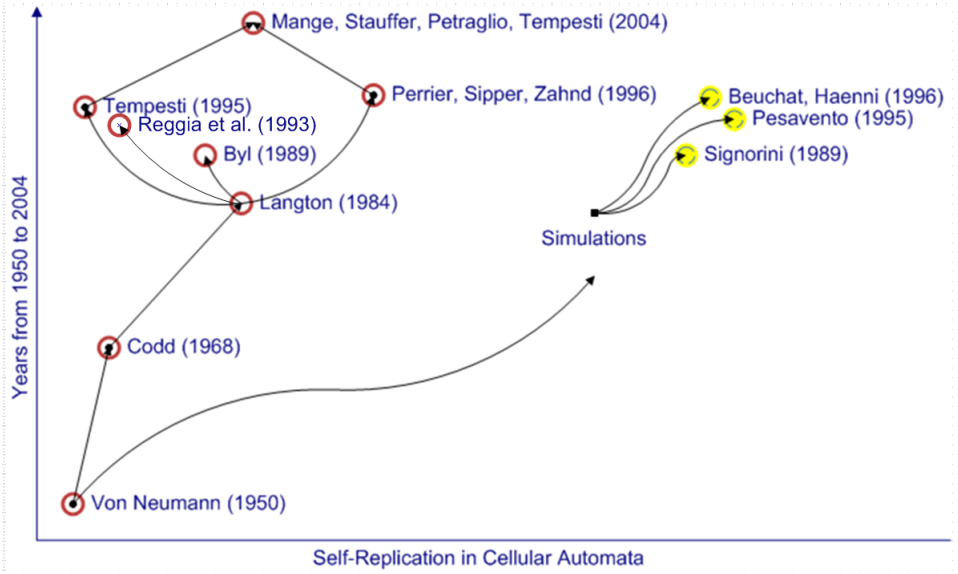



Figure 2.26.: Timeline from von Neumann to the present for self-reproduction in cellular automata.

This part of the review concentrates upon self-replication in cellular automata which stem directly from the initial work done by von Neumann on his universal constructor. The chronological connections can be seen in figure 2.26. Sipper states 'the motivation of this work is to understand the information processing principles and algorithms in self-replication' [71]. This work can be broadly broken down into several categories, the first of which are the universal constructor-computers which are capable of complicated tasks beyond just self-reproduction, [9, 11] produced by von Neumann and Codd.

Secondly are the machines derived from Langton [41], which are based upon Codd's [11] periodic emitter with eight states per cell but does not require a facility for universal construction. This machine is very simple and is completely achievable but can do nothing other than self-replication. Byl follows on from Langton using Langton's criteria for self-reproduction but minimises the time steps by changing the transition steps such that only six states are required rather than the eight previously, also the inner wall of the sheath is removed. Reggia et al. [68] discovered that sheaths on one or both sides of the data path were not essential and thus was able to construct smaller self-replicated

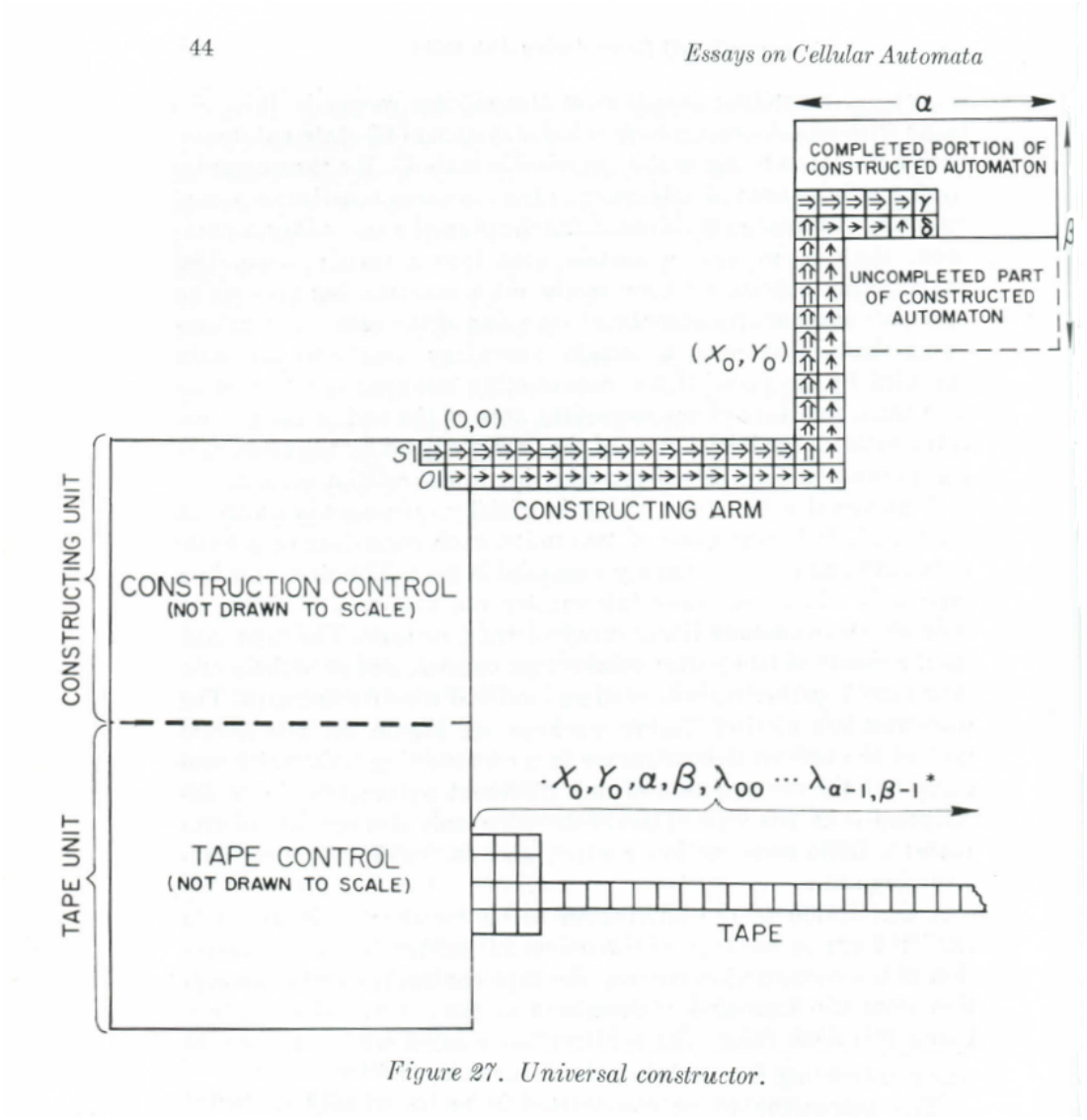



Figure 2.27.: von Neumann Universal Constructor taken from, [9] [fig. 27, p. 44].

loops.

The next category are those simple machines with a capacity to operate an inbuilt program, first of which by Tempesti [79], adapts Langton's loop to perform independent constructional and computational tasks where the program is stored entwined with the replication code. Perrier et al. [62] goes beyond Tempesti by their system having three parts: loop, program and data produced in that order.

Petraglio et al.[49] suggests a new algorithm, the *Tom Thumb algorithm* will create a self-replicating loop with both universal construction and computation properties. This may be a further category in the complexity of self-replication in cellular automata.

**von Neumann** More detailed examination is made of some of the self-replication systems described above starting with figure 2.27 is the final version of von Neumann's universal constructor. It should be borne in mind that all of the design as carried out by von Neumann is purely theoretical and was new and unimplementable at that time, Burks [9] and Thatcher [9, p. 132-186] continued the theoretical development after von Neumann's death. Additionally even today it is unlikely that there are the resources to implement a hardware version of the universal constructor.

The universal constructor is intended to have two powers:

1. Simulate any Turing machine.

2. Construct an automaton in a designated "Empty" region of the cellular space.

Von Neumann's self reproducing machine requires around 200,000 cells to meet the above powers and it intends to create electrical circuits and components ultimately reproducing itself. The Game of Life is also a 2d CA but is of much simpler construction as seen in section 2.3.3 which can produce abstract structures such as gliders.

The first design of the 29-state automaton had both a single path in the construction arm and also the tape input unit [9, p. xii] with a later design changing to a dual path constructor arm as shown in figure 2.27. The reason for the change was to accommodate a separate path for each of the ordinary and special transaction states. Burk states [9, p. xiii] that he thinks that von Neumann had realised that the dual path could also be implemented on the tape unit but that this design change was not implemented due to von Neumann's death.

Each cell in the automaton can have one of twenty nine different states as can be seen in table 2.3, on page 43. There are five categories of these states four of which take one time step to operate and the fifth where confluents take two time steps.

There are two types of transmission states which represent wires and an **OR** gate in a circuit, ordinary and special, each type can be in an excited state which is indicated by a

Table 2.3.: Von Neumann's 29 states.

| State | Symbol | |
|---|---|---|
| Sensitised states | $S_\theta, S_{10}, S_1, S_{00}, S_{01}, S_{10}, S_{11}$ and $S_{000}$ | |
| | Passive | Active |
| Quiescent state | U | |
| Ordinary Transmission | $\uparrow$<br>$\downarrow$<br>$\leftarrow$<br>$\rightarrow$ | $\cdot\uparrow$<br>$\cdot\downarrow$<br>$\cdot\leftarrow$<br>$\cdot\rightarrow$ |
| Special Transmission | $\Uparrow$<br>$\Downarrow$<br>$\Leftarrow$<br>$\Rightarrow$ | $\cdot\Uparrow$<br>$\cdot\Downarrow$<br>$\cdot\Leftarrow$<br>$\cdot\Rightarrow$ |
| Confluents | $C_{00}$ | $C_{01}$<br>$C_{10}$<br>$C_{11}$ |

dot next to the directional arrow or a quiescent state. When in a quiescent state the cell has no influence on its neighbourhood. The ordinary transmission state can receive input from all non-output directions and transmits the excited state in the output direction. The special transmission state acts similarly to the ordinary transmission states except when they output to a confluent state where it is converted to a quiescent state U. An example of an ordinary transmission passing the excitation is shown in figure 2.29. The ordinary and special transmission states are mutually antagonistic.

The confluents, $C_{uv}$, where $u$ and $v$ can be 0 or 1, $u$ is the current state and $v$ is the next output state, shows the double delay in this component. Confluents receive excitation from ordinary transmission states and transmit to both ordinary and special transmission states. A confluent acts as an **AND** logical gate to the excited inputs. Also it can act as a fan out by splitting the transmission excitation to all of the neighbouring cells that do not point to it. Figure 2.28 shows the progression of confluent states in this situation.

A cell in the quiescent state U is an unexcited or blank state. The construction process changes U into the sequence of sensitised states and subsequently to passive forms of

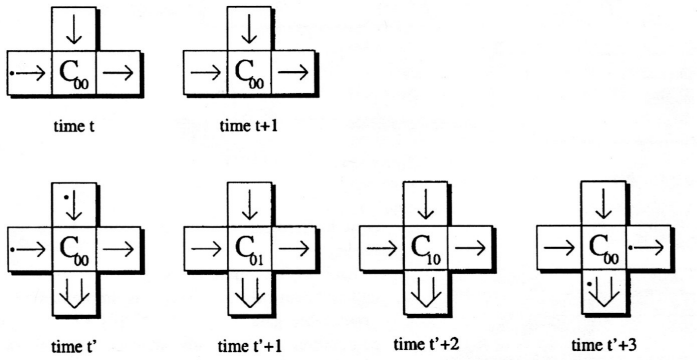



Figure 2.28.: Confluent states progression modified from, [5, Fig. 6. p. 303].

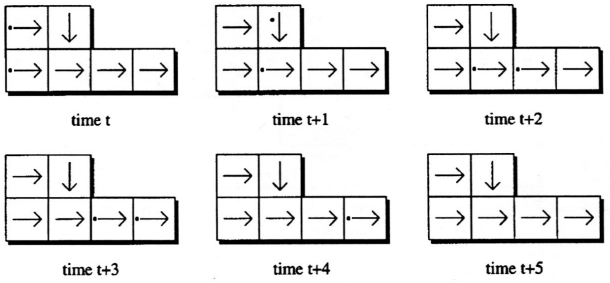



Figure 2.29.: Ordinary transmission progression modified from, [5, Fig. 5. p. 303].

confluent or transmission states. In the destruction process any confluent or transmission state changes into U in one time step [9, 70, 63, 5].

The sensitised states do not propagate signals but are intermediate states, that convert a quiescent state to one of the nine unexcited states, of which there are four ordinary transmission states, four special transmission states and a confluent state. Examination of figure 2.30 details the sensitised tree progression. the quiescent state that is excited enters the sensitised tree path and then dependant upon the subsequent signals received moves down the tree to reach the required final constructed state. A cell in the sensitised state has no effect upon any neighbours.

The transition from a transmission or confluent state can be reversed and the cell is returned to a quiescent state. An unexcited special transmission state which receives an excitation from an ordinary transmission state is destroyed . Similarly, the opposite is also true with the addition that if the receiving cell is in the confluent state this will be destroyed and examples of destruction are shown in figure 2.31. There are several limitations to von Neumann's two dimensional system design. The first issue is the

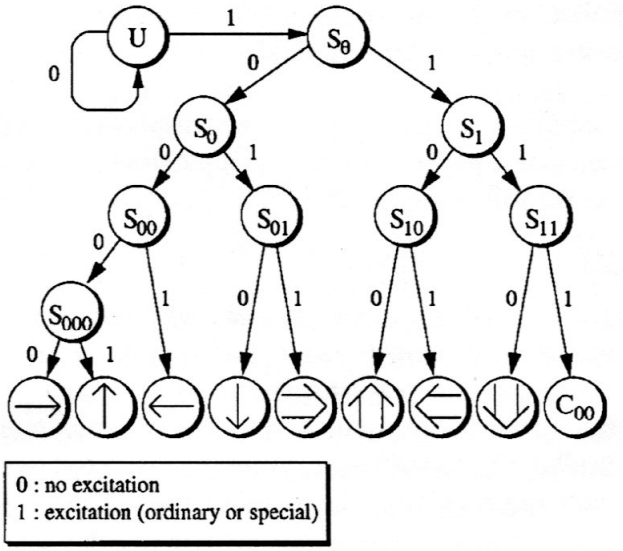



Figure 2.30.: Sensitised tree progression taken from, [5, Fig. 7. p. 303].

signal crossing problem, where circuits meet head on has no way for the signal to pass and secondly there is no **NOT** gate which by its absence can make circuit construction harder.

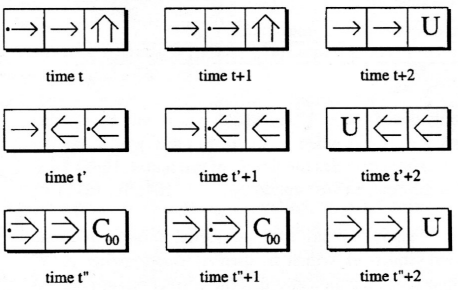



Figure 2.31.: Destruction examples modified from, [5, Fig. 8. p. 303].

Von Neumann designed some basic organs in the system and nearly completed the design of an indefinitely expandable tape and its control. The organs, *Pulser* and *Decoder* shown in figure 2.32 and 2.33 are derived from Thatcher [9, p. 132-186]. The pulser will be used for coding of commands to the main channel and also for the tape and constructing operations. It has one input and one output and is operated by the input of a single pulse. The output is formed by multiplexing within the pulser as can be see in figure 2.32 which gives the output sequence P(1101). The decoder can also be built to recognise various sequences of input and restrict unwanted sequences. From these two

organs other more complex organs can be created. A decoder that recognises D(1101) is shown in figure 2.33. Each of those that give a method for the creation of the basic organs do so in slightly different ways, however I prefer Thatcher's algorithmic methods as being fairly straight forward in its construction.

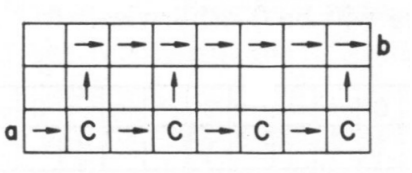



Figure 2.32.: Pulser (1101) modified from, [9, p. 147].

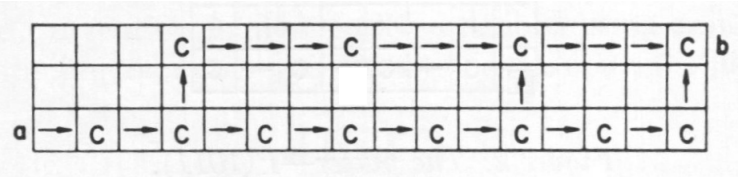



Figure 2.33.: Decoder (1101) modified from, [9, p. 148].

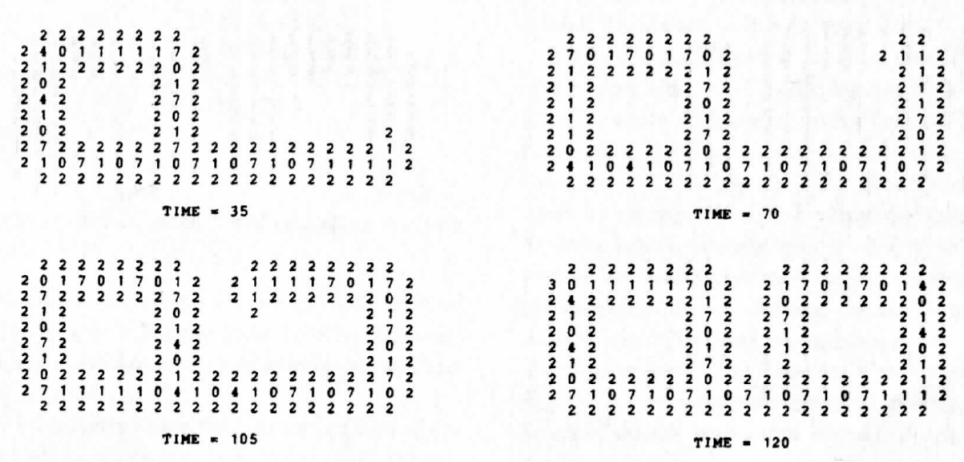
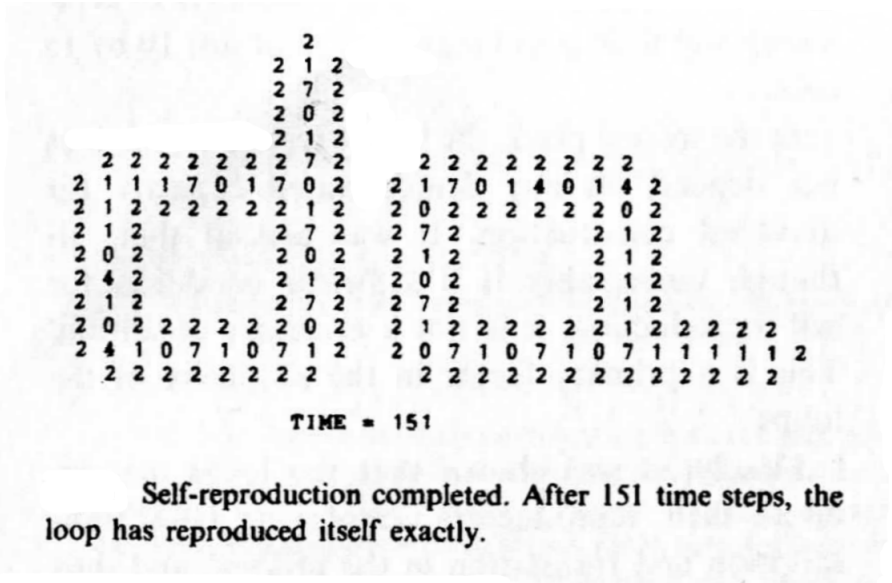



Figure 2.34.: Langton's loop showing stages of daughter construction taken from, [41, fig. 7, p. 141].

**Langton** In terms of complexity where von Neumann [9] and Codd [11] are the most complex, Langton's loop [41] is at the opposite end of the scale and can be considered to be the most simple. This self reproducing loop is an adaption from Codd's universal constructor of which the timing unit is called the periodic emitter. This has the form of a loop and a constructing arm at the bottom right hand corner formed of an outer




Self-reproduction completed. After 151 time steps, the loop has reproduced itself exactly.

Figure 2.35.: Langton's loop showing subsequent steps after daughter completion taken from, [41, fig. 9, p. 143].

sheath then a data path and finally an inner sheath [11]. As the data travels around the path when it reaches the T-junction the signal is duplicated with one copy going up the right hand side of the loop and the second out along the exiting path. Langton identified that the periodic emitter can be used as a storage unit and thus with modification of the transaction rules created his self-replication loop. As can be seen in figure 2.34 the configuration of the loop is a square path equal to the number of instructions to build one side and a left hand corner, which are:

$$70 - 70 - 70 - 70 - 70 - 70 - 40 - 40$$

where the '70' cells extend the construction arm by six cells and then the two '40' signals build a left hand corner at the end of the arm. On the fourth of these cycles the end of the arm fuses back on to itself using the transaction rules, which separate the two loops and then implement the signals for new constructor arms to be implemented for each loop as shown in figure 2.35. The original loop can make four copies of itself assuming there is space to do so, then the offspring can only create new copies in the unused grid area until the edge of the grid is reached. When each loop has made all permitted copies the data instructions are erased and the loop becomes dead.

**Tempesti** This self-replicating loop extends beyond Langton's loop by adding computational and constructional capabilities. The main differences are the use of the 9-cell neighbourhood, a single sheath on the inside of the loop [79], rather than on the outside of the loop as described by Byl [10]. In addition there are four 'gate cells' in the same state as the sheath. These cells are initially in the open position and allow the constructing arms to extend beyond the loop through them, then once the loop is replicated and the program data has been transferred the cells close[79]. The gate cells are indicated by the four number 1's outside the program loop in the time equals zero part of figure 2.36. The constructing arms extend automatically and only require signalling as to when to turn by 'messengers' who move at twice the speed of the constructing arm. The loop is constructed in two parts, first the sheath is constructed in its entirety and then then secondly the program data to complete the loop. Finally the constructing arm withdraws and the cell gate closes, however the loop does not die as the program contained within remains able to execute. The constructing arm is border sensitive and upon reaching an edge detects the edge and retracts without crashing in the way Langton's does [41, 79]. Figure 2.36 is an example of different stages in the loop described above. The program operation is carried out in space that forms the loop centre starting from the top left hand side. [5, 48, 63, 70, 71, 80]

**Perrier, Sipper and Zahnd** This self-reproducing loop uses Langton's version as a starting point with the capability of universal computation. It comprises of three parts; loop, program and data, as indicated in figure 2.37, where the dots indicate the sheath, the arm to the right is the constructing arm. The program is run by a Turing machine model, the W-machine, introduced by [Wang, 1957, p. 11] as cited in [62]. The reproduction of the loop is carried out in the same manner as Langton does with the exception of the special signals required for the program and data operations. Once the loop has been created, the constructing arm partially withdraws leaving the bottom sheath, which becomes the data path for both the program and data to be copied to the reproduction.

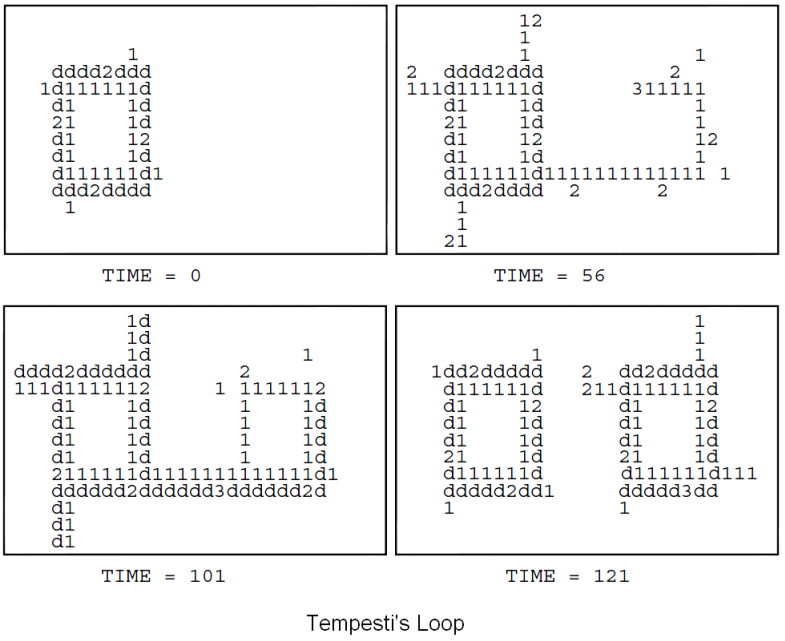



Figure 2.36.: Tempesti's loop taken from, [79, fig. 3, p. 2].

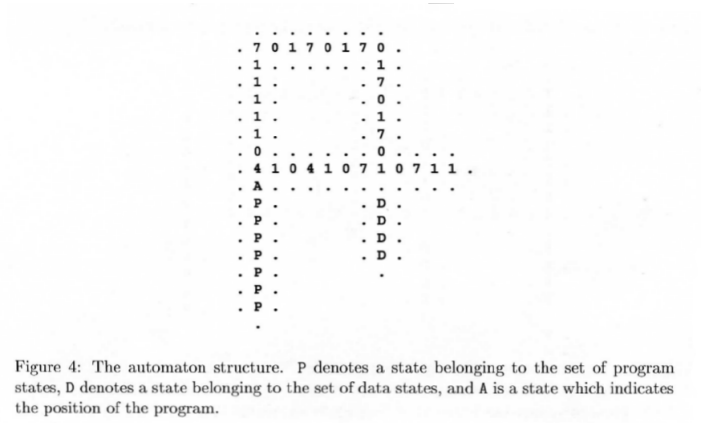
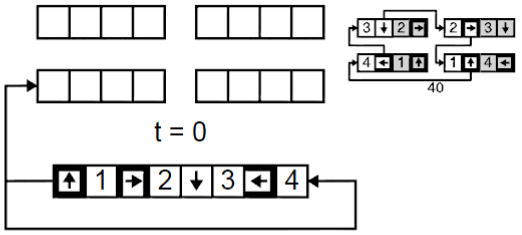


Figure 4: The automaton structure. P denotes a state belonging to the set of program states, D denotes a state belonging to the set of data states, and A is a state which indicates the position of the program.

Figure 2.37.: Perrier's loop taken from, [62, fig. 4, p. 11].

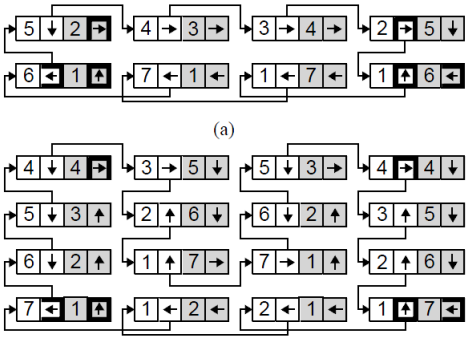
Upon completion of these steps, the constructing arm fully withdraws and the program is executed in the first machine whilst the offspring machine starts its own reproduction. There are possible issues with the data tape where the copy is identical to the original and in theory is of unlimited length, but in practice is limited by the capacity of the machine system, and the number of CA states in use is 63, [62, p. 24].

**Stauffer et al.** A new kind of cellular automaton is created for this hardware implementation of the 'Tom Thumb algorithm', as it has both processing and control units




The minimal cell ($2 \times 2$ molecules) with its genome at the start ($t = 0$).

Figure 2.38.: Universal construction loop using Tom Thumb algorithm taken from, [49, fig. 1, p. 180].




Two examples of non-minimal self-replicating loops: (a) a $4 \times 2 = 8$ molecules loop ($\Delta t_n = 20$, $\Delta t_e = 28$, $\Delta t_c = 32$); (b) a $4 \times 4 = 16$ molecules loop ($\Delta t_n = 40$, $\Delta t_e = 60$, $\Delta t_c = 64$).

Figure 2.39.: Universal construction loop using Tom Thumb algorithm taken from, [49, fig. 10, p. 188].

[62]. These authors change the definition for cell in the CA's to molecule as they do not wish to conflict with the biological definition. Thus, this self-replicating loop is capable of universal construction and can be implemented within hardware effortlessly. In addition, they wish to introduce ' the data and signals of cellular automaton which perfectly suits the specifications of their basic molecule. Thus allowing a straightforward and systematic methodology for synthesising cellular automata' [62, p. 179]. The minimal cell compatible with the Tom Thumb algorithm is made up of four molecules in a two by two matrix (fig. 2.38). Each molecule can hold, in its four memory positions, four hexadecimal characters of the artificial genome, thus the whole cell has a maximum of sixteen such characters. In figure 2.38 the cell to the right shows the completed cell

after forty time steps, based upon the initial input shown. The character spaces can have one of three types, empty, molcode data ( for molecule code data, from 1 to 7) or flag data (from 8 to E). Molcode data is used for the final artificial organism configuration and flag data is used to construct the skeleton of the cell. Moreover, each character has a status of fixed data or mobile data. The characters in the final cell (fig. 2.38) are fixed where the background of the character is shaded and mobile where the background is white. In this algorithm, the odd characters of the genome are always a flag F, whilst the even characters are always a molcode M. The mother cell should be able to construct two daughter cells, northward and eastward, so that the artificial organism can grow in both horizontal and vertical directions. Figure 2.38 shows two examples of non-minimal loops. The molcode can be used as a configuration string for controlling a field-programmable gate array(FPGA). This has been used with two thousand FPGAs for the LSL Biowall to demonstrate self-replication in hardware [62, p. 188-189].

### 2.3.4. von Neumann's 29-state cellular automaton software simulations

There are three main approaches to simulation of von Neumann's 29-state cellular automaton, Signorini 1989, Pesavento 1995 and Beuchat & Haenni 2000.

The first, **Signorini**, concentrated on an implementation of the transition rule on a Single Instruction, Multiple Data (SIMD) class of parallel computers [70]. A Gapp NCR45CG72 processor has 72 cells available for parallel processing and a collection of these is used to give the required cellular structure. From the 29 states shown in table 2.3 there are four sets of data to be encoded and will need a total of thirteen bits in a memory frame as seen in figure 2.40.

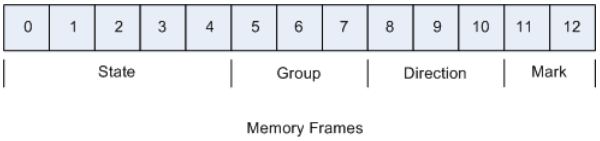



Figure 2.40.: Signorini's data encoded binary values memory frame, [70, fig. 8, p. 180].

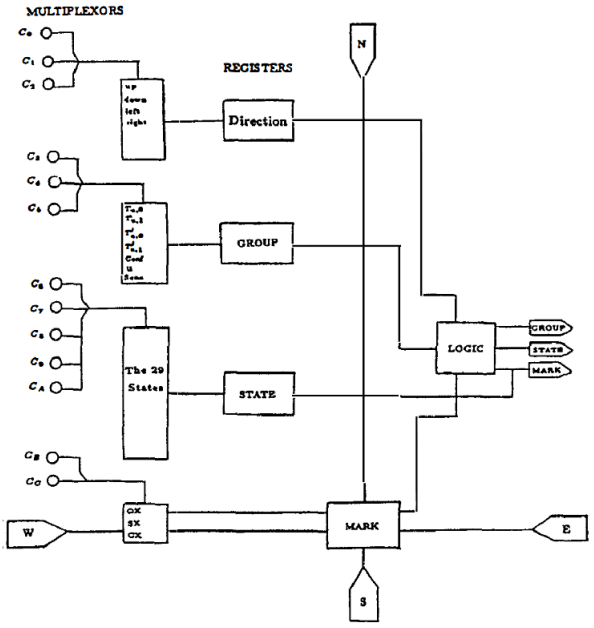



**Figure 11: Block Diagram of a Cell on the Automaton.**

Figure 2.41.: Block diagram of a cell on the automaton, taken from, [70, fig. 11, p. 182].

The algorithm to undertake this will be implemented in two phases, cell marking and then cell updating. Cell marking is where any excited cells mark their neighbour cells which they are pointing to. A marked cell, depending on what group it is in, is then at the next time step killed or excited. For the second phase seven sets of micro-instructions are executed in succession [70, p. 181] and finally, the block diagram (fig. 2.41) is the transition algorithm circuit. The mark register takes two cycles to operate with the first cycle for writing and the second for reading to then compare with the group register contents and subsequent updating of other registers dependent upon the comparison outcome. Signorini states 'The transition rule is computed each time step, and implementing this CA goes through the development of new tools for the construction of complex configurations from organs, and programming them'[70].

In the second software simulation **Pesavento** makes some alterations to the basic self-reproducing machine structure. The main one being the changes to the confluent element, which means that it is no longer a 29-state machine but rather a 32-state machine. The primary goal here is for self-replication and not universal computation.

This design follows quite closely to that of von Neumann's and in Thatcher's work[9, p. 132-186]. A lot of the components follow von Neumann's design whilst others have been replaced by smaller pieces of equipment. The confluent has the same four uses as those mentioned in 2.3.3 and these extra uses: 'as a memory unit (when none of its four neighbours is an outgoing transmission element); as a one step delay (when one of its neighbours is an incoming transmission element and another is an outgoing transmission element); and as a crossing unit (when its neighbourhood consists of two pairs of transmission elements, with each pair containing an incoming and an outgoing transmission element)' [63, p. 347].

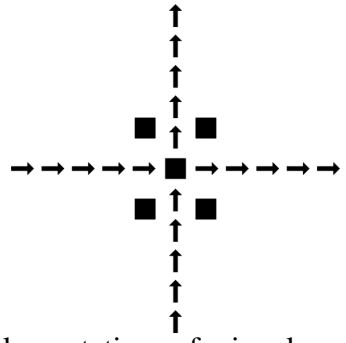


Figure 2.42.: Alternative implementation of signal crossing in confluent element, adapted from, [63, fig. 11, p. 348].

Figure 2.42 demonstrates what is meant by the description described in the last confluent use. Pesavento concludes his work by stating that using cellular automata as models for parallel processing requires more complex transition rules. Moreover, using von Neumann's machine to simulate a sequential Turing machine is not efficient and other strategies should be considered [63, p. 350].

The final simulation described here comes from **Beuchat and Haenni** and is a hardware solution to the logic to implement the behaviour of a single cell and an algorithm for a simple approach to the construction of organs which use the same methodology as von Neumann which is described in section 2.3.3. [5, p. 300].

The hardware logic module for a single cell is called a "biodule" and has connections to allow other modules to be interconnected to build a small cellular array. As seen in figure 2.43 the logic module is made in two parts. The first is the computation unit and

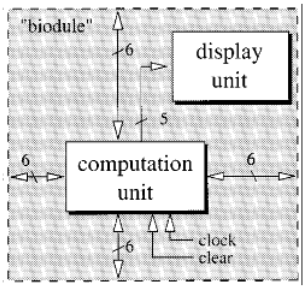



Fig. 11. Block diagram of the "biodule".

Figure 2.43.: Block diagram of the "biodule", taken from, [5, fig. 11, p. 304].

is built on an FPGA and calculates the cell's future state, stores the current state and outputs it to the second unit, the display unit. The display unit is a dot-matrix display which shows the current state of the cell at all times. In this implementation each cell only needs to know five of the twenty nine states and so these neighbour notifications are coded on three bits as shown in table 2.4.

Table 2.4.: Cell State Transmission between Neighbours.

| Code | Meaning |
|---|---|
| 0ΦΦ | don't care |
| 100 | unexcited ordinary transmission state |
| 101 | special excitation |
| 110 | confluent excitation |
| 111 | ordinary excitation |

Hence each cell has three bits of input and output and doing this reduces complication of the computation for this part. State encoding is based on three fields with a total size of ten bits as shown in figure 2.44 and we can see the state type content in table 2.5.

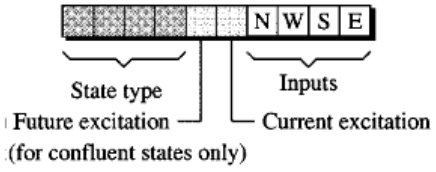



Fig. 12. Encoding on ten bits divided into three fields.

Figure 2.44.: Ten bit encoding divided into three fields, taken from, [5, fig. 12, p. 304].

Table 2.5.: Encoding of the State Type.

| Code | Meaning |
|---|---|
| 0000 | quiescent state |
| 0001 | special transmission state |
| 0010 | confluent state |
| 0011 | ordinary transmission state |
| 1xxx | sensitised state number xxx |

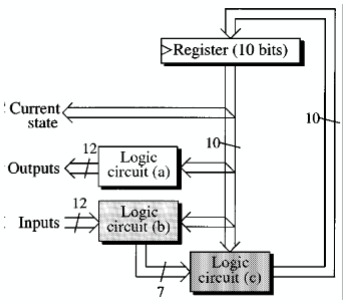



Fig. 13. FPGA content.

Figure 2.45.: FPGA content, taken from, [5, fig. 13, p. 305].

The transition rules are implemented on an FPGA, the content of which is seen in figure 2.45 with the logic circuit (c) shown in figure 2.46. The three logic circuits serve the following operations; logic circuit (a) makes the outputs of the current cell according to the current state, logic circuit (b) generates the seven signals shown on the left hand side of figure 2.46 and logic circuit (c) ascertains the future state of the cell.

In figure 2.46 the ROM is a look-up table based upon the arbitrary rule of the last step in figure 2.30. The constructing arm as described by von Neumann is not used in this implementation due to the limited number of “biodules” so they are physically configured as the required organ. The single path construction procedure is used to extend to a new area within the grid without using the external arm to achieve this. Hence the algorithm computes all of the required signals with each cell having a state and a mark. The mark is used to identify the cell for the construction order. The cell mark number is the length of the path across the cells from the connector to a particular cell with all cells initially being marked zero. So the cells are built from the last to the first, one cell at a time and the path through the organ being built is the constructing arm.

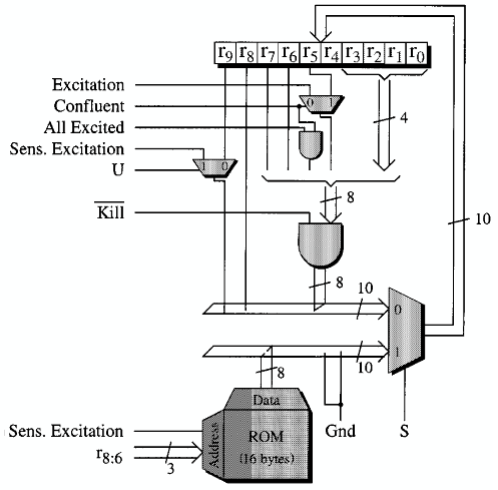



Fig. 14. Logic circuit (c).

Figure 2.46.: Logic circuit, taken from, [5, fig. 14, p. 304].

Also where there are alternative paths of the same length the choice is made randomly. Only small organs can be created due to resource restrictions but are a successful first hardware implementation [5].

## 2.4. Emergence

**Artificial Life – Biologically Inspired Engineering** In this section we begin to examine several strands of research by Doursat and associates into artificial development and programmed networks. This [22] introduces a multi-agent simulation and exploitation of morphogenetic processes by integrating self-assembly, pattern formation and genetic regulation to form an evo-dev inspired approach. In emergent engineering [23, 24], self-assembling networks composed of dynamic nodes with the ability to have wireless connectivity are considered for Internet and e-networking using simple chaining and lattice formations.

### 2.4.1. Morphogenesis

How can we link the biological development idea of morphogenesis to computer science and engineering? Morphogenesis is sometimes stated as being “more than the sum of

its parts", leading to the idea that *emergence* [21] is one of the key concepts in the study of morphogenesis.

From previous examination of Conway's game of life in section 2.3.3, we found that it had emergent behaviour from the three simple transaction rules. Hence we can see that this is similar to a biological cell where no one cell is any more important than another and the growth is purely based upon the transaction rules in force at any particular time step. In the biological sense these rules may also change whereas in Life the rules remain the same for the full run time.

An example of morphogenesis in the game of life is shown in Figure 2.47. The patterns are created within the same three by four cell grid and occupy the same x,y coordinates within the overall grid. The change occurs at each time step with the individual cells changing according to the transition rules.

1. Shape A and B are two possible patterns which will create pattern D at the next time step.

2. Shape C or shape D will then create pattern E at the subsequent time step.

3. Finally the last change will turn shape E into shape F.

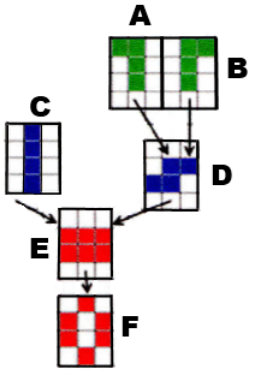



Figure 2.47.: Example of morphogenesis in the game of life.

The final shape is a still life known as a Beehive and is considered to be an attractor because several other shapes converge to it. The pattern will remain in the beehive shape

from one generation to the next, providing there is no further interaction from external neighbours in subsequent time steps. Still life's are considered to be an oscillator with unit period 1 [83].

These three steps demonstrate the idea of 'trap doors' where once the pattern has gone through the trap door then there can be no going back. This is due to the pattern that was used not being retained and therefore cannot be replicated correctly. This leads to another concept that *morphogenetic processes* cannot be deduced from their final form.

### 2.4.2. Embryonic Development

**Self-assembly and adaptive self-organisation.** In these sections we consider research undertaken by Doursat in the area of *artificial life* and in particular *biologically inspired engineering*.

There is huge growth in computer systems and traditional system design is having difficulty in keeping up with future trends. Currently, open source software is leaderless and programmers operate in groups where they modularise the work and do not rely on a single controlling mind. This method is approaching the one proposed by Doursat. "Biological organisms, which might give the illusion of deliberate design are in fact the product of *undesigned evolution* through random variation and non-random natural selection, excluding the need to invoke any form of intelligent design for them [22, p. 168]." Figure 2.48 leads us on from this statement where *undesigned evolution*(UE) is beyond the brick wall and is not achievable by current design methodologies. However it is suggested that *intelligent design*(ID) is the standard engineering design point which moves towards *intelligent meta-design*(IMD) and a final goal of *evolutionary meta-design*(EMD) at some future point. This means that the fundamental laws required to model the self-organisation and self-building of structures from a single cell with the ability to differentiate the differing stages of growth have to be created for the IMD stage and then the possibility of evolution taking over at the EMD stage.

Self-organised and structural systems in this model are initially based upon a cellular

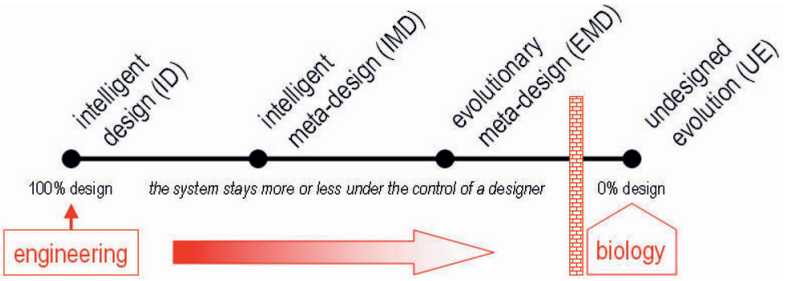



Fig 8.1. Four stops and one brick wall on the design-evolution line.

Figure 2.48.: Design versus evolution spectrum [22, fig.8.1, p. 169].

automata lattice as discussed previously but much closer attention is paid to emulate the genetics and biological development or 'evo-devo'. Each cell contains a genetic regulatory network (GRN) [19, 20, 22] which is modelled as a feed-forward hierarchy of switches [19]. In [19], Doursat talks about reverse engineering possible solutions but does not categorically state that there is not more than one path in the GRN switch process and thus the issue of 'trap doors', may arise without us knowing what route to take and thus reverse engineering may not be possible.

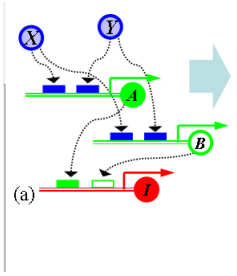



Figure 2.49.: Gene regulatory interactions GRN [22, fig.8.3, p. 179].

Complex morphogenesis has three fundamental ingredients; self assembly(SA); pattern formation(PF) and genetic regulation(GR), where the model can be thought of as either moving cellular automata or heterogeneous collective motion [20] and modularity is an essential condition of evolvability.

Figures 2.49 and 2.50 show the complete pathway from start to finish where figure 2.49 is the switches for a single cell GRN which can settle in various on/off states. Figure 2.50 then shows (b) continuing from (a) and on to (c) a lattice of cells and (d) final positioning by the local morphogen gradients (X, Y). The creation grows by cell pro-

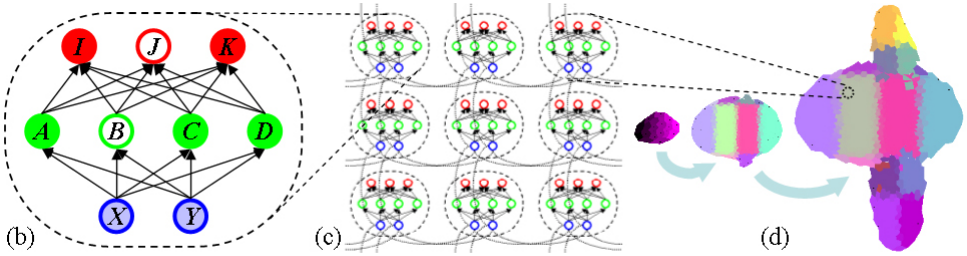



Figure 2.50.: GRN subsequent positions [22, fig.8.3, p. 179].

liferation, creating new local gradients within single identity domains in (d) [22]. This cellular process intends to create a novel engineering model capable of self-assembly by an abstraction of biological development rather than a central control.

### 2.4.3. Emergent engineering

There are broadly two types of abstract model for a self made e-network, [23, 24], simple chaining and lattice formation. The nodes have limited positional awareness and during self assembly can exchange messages and make links. As the network expands and node positions change, nodes adapt by switching different rule subsets on or off, thus triggering growth of chains, lattices and more complex composite topologies.

Simple Chaining, figure 2.51 is the basic node for this type of self assembling structure, it is realised with two ports *X*, *X'* and two internal gradient values *x*, *x'* in each node. Initial conditions for nodes are both ports open and gradients set to 0.

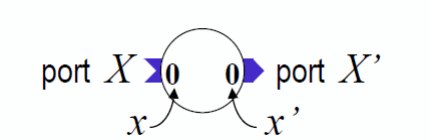



Figure 2.51.: Simple Chaining [24, p. 13].

The ports can be free or occupied and also open or closed. New ports are set to open with a gradient of zero and the value of *x* is sent out of *X'* with a value of *x*+1. Each new node can connect to any available port and the ports are closed as soon as *x*+*x'* = n – 1. The gradient counters keep track of the position of the node in the chain. Then the gradient number can be used for decisions such as length of chain or branching

to allow more complicated structures. Its not clear in [23] in the case where a port is occupied and then closed whether the connection is retained. Finally node operations are carried out in this order **G** ⇒ **P** ⇒ **L** and then back to **G**. Gradient update(**G**); port management(**P**); both of which are executed by nodes in the network and link creation (**L**) generic logic to select an open port.

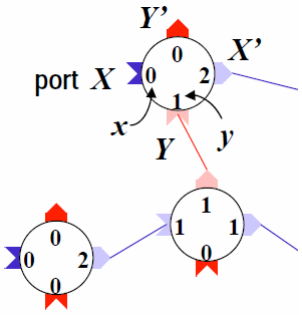



Figure 2.52.: Lattice formation by guided attachment. [24, p. 16].

Lattice formation by guided attachment has the same routines as above **G** ⇒ **P** ⇒ **L** and then back to **G**. It also now has two pairs of ports with *X*, *X'* and *Y*, *Y'* and also two sets of internal gradients *x*, *x'* and *y*, *y'* as shown in figure 2.52. The lattice begins with one node above and one node to the side as the basic start and subsequently expands by filling the in the internal corner until $x+x' = n - 1$and or $y+y' = n - 1$. The nodes can continue branching even though one or more chains has reached its limit as long as the branch line has not. Single nodes can be replaced with a cluster of nodes to give an element of mutability by the addition of an extra port (**C**). Possible other features are modular structures. Modules can do one of several things with one of these being that they can have different gradient ports identified by tags *a, b, c*. Hence a change of rule is required for **L** so links can only be created between ports with the same tag. Also this can give the option of alternative paths instead of branching. Finally it can now be seen that figures 2.49 and 2.50, have the same modular structures by local gradients as does e-networks created by the modular structures by local gradients methodology mentioned above.

# 3. Research Question

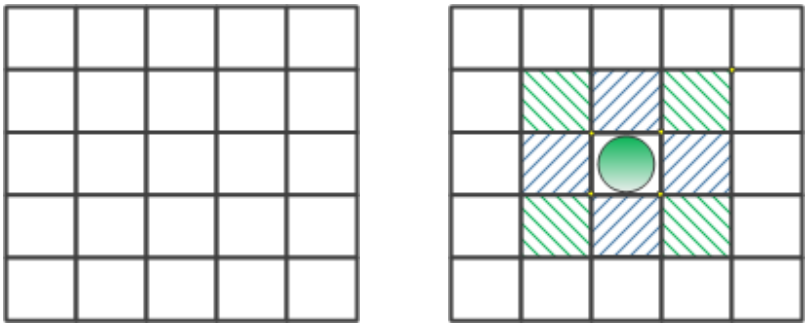
## 3.1. Why

This is an exploration of many-core parallel systems based upon a NoC where each software process can either move or spawn further software processes within a local neighbourhood of processor cores across a large grid of cores by simulation. There is a need to have local rules to allocate dynamically changing processes to multiple processors.

## 3.2. Example and terminology

We use an Agent-Based simulation approach. This section defines some terminology. Figure 3.1a is a 2D simulation grid which represents a *many-core* system. Each *cell* is a single *processor* or *core*. *Agents* are shown as a coloured object contained within a single cell and are a *software process*, Each process can operate within a single core until they either terminate, move by relocating within the local neighbourhood and or spawn further software processes in the local neighbourhood.

Figure 3.1b has a single active cell at coordinates (2, 2) with the cells within one hop in the classic von Neumann method shown with blue hatching. The cells shown in green and blue hatching combined are the classic Moore neighbours for a hop of one. Hop's are the distance in cells from the current active cell that constitutes the local neighbourhood.



(a) Inactive Simulation grid. (b) Active simulation grid.

Figure 3.1.: Simulation grids showing an empty grid and a grid with an active core. The cells with hatching are the local neighbourhood for a hop of one for both classic von Neumann and Moore.

## 3.3. How

Given a simulation where the cells of the grid have state, which either contain a software process or are awaiting to receive one, the processes are distributed within the grid by the local neighbourhood algorithm. The algorithms under investigation provide:

- A local neighbour selection ordering.
- A transition function dictating operation conditions.
- Neighbourhood size dependant on local distance of neighbourhood.

## 3.4. What

The detailed research question investigated is: how is the agent growth pattern affected by the size of the neighbourhood and complexity of neighbour selection?

# 4. Simulation Methodology

## 4.1. Agent-based Modeling and Simulation

The criteria required by us for Agent Based Modeling and Simulation (ABMS) toolkits are as follows:

- Source Code: Java.
- Type of Agents based upon its interaction behaviour: Agents / objects as Java classes.
- Integrated Development Environment (IDE): Eclipse.
- Simulation models' scalability level: High/Large scale.
- Application domain: General purpose 2D simulations, cellular automata, complex adaptive systems.
- Utilise the MVC Design Pattern: Model, View, Controller.
- Free for Academic use.

## 4.2. Simulation Grid Structure

ObjectGrid2D [1] is our choice for most of the experiment grid structures due to only allowing 0 or 1 objects per location. In most of the simulations each cell can only

[1] Whilst using the built-in neighbourhood methods in Grids of Objects we identified three different software bugs which turned out to be undocumented default behaviours. After consultation with Professor Sean Luke, lead developer of MASON,
`https://cs.gmu.edu/~eclab/tools.html`
these areas of undocumented behaviour were resolved.

contain a single agent thus ObjectGrid2D is the most appropriate for these simulations. Section 5.3.2.1 of the MASON manual [45] gives a comparison of the differing grid choices to help make the best choice. All simulation grids are bounded regardless of the size of the grid, hard edged and cannot be crossed. Whilst we are limiting this research to simulation the ultimate goal is to implement the model directly in hardware.

The following terminology has been chosen:

**"Hop"** Is the distance from the central location of the local neighbourhood to the farthest neighbours.

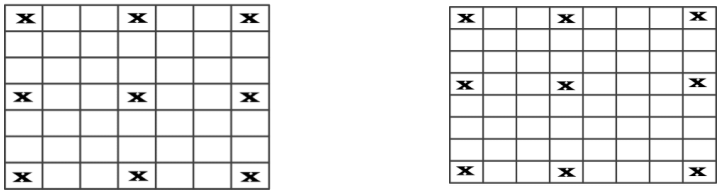
**"Step"** Is a process of calculating the simulation's next state.

**"Time step"** Is the time interval for which the simulation will progress during the next "Step".

Thus in the simulation experiments shown in chapter 6, an example of Hop and Step are; change of agent state for each step dependant upon the hop size and time step.

In some of the neighbourhoods we have chosen to have a designated starting cell position in nine different positions. These are: the four corners, four edge mid points and a central cell. This shows if any of these positions are better starting points when compared to their total occupancy. Some of the grids have odd sides and others have even sides. In the case of an odd sided grid the starting cells are shown in figure 4.1a and similarly the starting cells for even sided grids is shown in figure 4.1b.

These simulations in terms of mapping processes are using local neighbourhood algorithms to move or spawn processes within the grid structure. As each cell represents a core this is a direct link of the processes being brought into action on the cores.



(a) Odd grid starting cell choices (b) Even grid starting cell choices.

Figure 4.1.: Cells for all fixed start point simulations

## 4.3. Neighbouring cell order disruption

Each of the built-in neighbourhood methods has the option of including or excluding the central cell within the local neighbourhood array. These methods have a lot of overhead and the manual [45], advises that omitting the central cell reduces the total load. When omitting the central neighbourhood cell, the latter half of the coordinate arrays are disrupted. In order to rectify the disruption, a JAVA class extension is added to revise ObjectGrid2D to MyObjectGrid2D. The table 4.2 is the coordinate neighbour array of a Moore neighbourhood for a hop of one, with both the ObjectGrid2D class and the MyObjectGrid2D class neighbourhoods with and without the origin for comparison. The origin cell in this table for all versions is at location (2,2). The ObjectGrid2D version without the origin has one location with the coordinates shown in bold text. The reordered neighbour is the cause of early ending of this simulation.

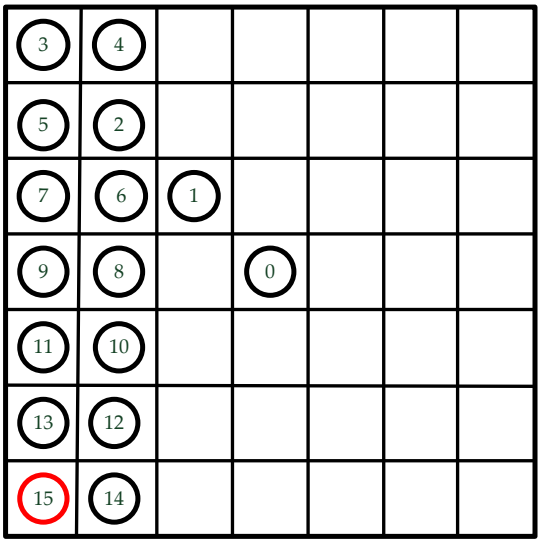



Figure 4.2.: Numbered path of selected neighbouring cell occupied for each time step. The cell containing the red ring is the location that has stopped the simulation by having no empty cell within a hop of one.

The table 4.1 shows the neighbouring cell coordinates for the Moore neighbourhood at an internal origin position, a corner origin position and an edge origin cell location. Each of these has one cell reordered, such that rather than filling cells from the left to the

right. The changed cell position when it becomes the origin leaves an empty cell to the left. That empty cell becomes the next to be occupied whereupon all other empty cells are out of reach. Such as the one shown at coordinates (6,0) in figure 4.2 is occupied. This is the bottom left hand cell of the grid and is shown with a red ring containing the number 15 is positioned, such that all further empty cells are beyond a hop of one.

| Neighbour Cell Coordinates for ObjectGrid2D version | | | | |
|---|---|---|---|---|
| | with inside origin cell (2,2) | | without origin cell | |
| Location | X | Y | X | Y |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 1 | 2 |
| 2 | 1 | 3 | 1 | 3 |
| 3 | 2 | 1 | 2 | 1 |
| 4 | *2* | *2* | **3** | **3** |
| 5 | 2 | 3 | 2 | 3 |
| 6 | 3 | 1 | 3 | 1 |
| 7 | 3 | 2 | 3 | 2 |
| 8 | 3 | 3 | | |
| | with corner origin cell (0,0) | | without origin cell | |
| 0 | 0 | 0 | **1** | **1** |
| 1 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 1 | 1 | | |
| | with edge origin cell (0,2) | | without origin cell | |
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 2 | **1** | **3** |
| 2 | 0 | 3 | 0 | 3 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 1 | 2 | 1 | 2 |
| 5 | 1 | 3 | | |

Table 4.1.: Classical Moore neighbours location array for a hop of one, showing neighbours for an inside, corner and edge location. Bold numbers are the coordinates for the neighbour which has changed its selection order.

The JAVA class extension code for the revised MyObjectGrid2D is in appendix A.1.

| Neighbour Cell Coordinates | | | | |
|---|---|---|---|---|
| | ObjectGrid2D version with origin | | MyObjectGrid2D version with origin | |
| Locations | X | Y | X | Y |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 1 | 2 |
| 2 | 1 | 3 | 1 | 3 |
| 3 | 2 | 1 | 2 | 1 |
| 4 | 2 | 2 | 2 | 2 |
| 5 | 2 | 3 | 2 | 3 |
| 6 | 3 | 1 | 3 | 1 |
| 7 | 3 | 2 | 3 | 2 |
| 8 | 3 | 3 | 3 | 3 |

| | ObjectGrid2D version without origin | | MyObjectGrid2D version without origin | |
|---|---|---|---|---|
| Locations | X | Y | X | Y |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 1 | 2 |
| 2 | 1 | 3 | 1 | 3 |
| 3 | 2 | 1 | 2 | 1 |
| 4 | **3** | **3** | 2 | 3 |
| 5 | 2 | 3 | 3 | 1 |
| 6 | 3 | 1 | 3 | 2 |
| 7 | 3 | 2 | 3 | 3 |

Table 4.2.: Moore neighbours array for a hop of one, origin at coordinates (2,2). The coordinates (3,3) in bold is the one that has changed position within the neighbour selection ordering.

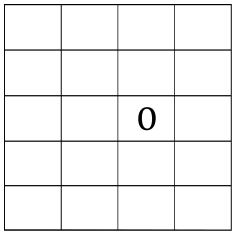
This extension resolves the cell order disruption.

## 4.4. von Neumann neighbourhood - Tunnelling

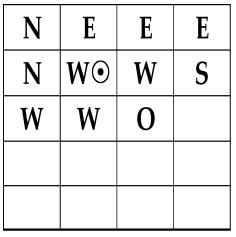
At each time step of the simulation a local neighbourhood array of cores is generated for each agent. The number of cores is dependent upon the hop size. What happens during each time step is down to the particular algorithm being applied. The array of cores is checked for an unoccupied core from left to right. If an unoccupied core is found then the it is used as the next location for a process to be applied. If no empty core is found then the last (X,Y) coordinates are held in the next core method and tunnelling can happen.

Tunnelling is when the next location finder method holds the last pair of coordinates within the array and moves to the next step and those (X,Y) coordinates. This is repeated until an empty core is found, whereupon a process is applied to that core. Assuming there are now one or more empty cores the normal operation continues. Figures 4.3a, 4.3b and 4.3c will help explain what is occurring during a simulation. This example uses the von Neumann neighbourhood and cores that can be selected if empty are in the following order, West, North, East and South from the origin. Figure 4.3a is step one of the simulation with the first cell occupied and indicated with ‘O‘. The next figure 4.3b, is ten steps further on. Each letter in the cells indicates which direction was used to get to that particular core. This core ‘W⊙‘ has all four neighbours occupied. This is where tunnelling begins on the next step. The simulation at each step looks for an empty core to occupy and if none are found the next core coordinates for the last direction in the array is used but no object is added. The last figure 4.3c shows two cores where the tunnelling has occurred and are indicated by the downward arrow in those two cells. The one with the ‘S‘ being the empty cell reached and the remainder of cells are occupied in the normal manner. Tunnelling is not detrimental to the simulation once one understands what is happening. An alternative which prevents this is to have a finish condition when a cell whose neighbours are all occupied is entered. In large grids preventing tunnelling caused a substantial amount of cells to be unavailable. Tun-

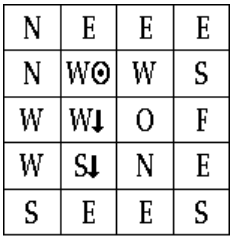
nelling was identified when considering multi-process experiments. These required the simulation speed to be reduced and the and tunnelling was observed. Subsequently the single agent simulations were re-run and in some cases the occupancy factors are much reduced. It was decided that the finish condition would be applied to all simulations as if tunnelling is allowed then all most all cores are occupied regardless of algorithm or neighbourhood size and shape.



(a) Step 1 of simulation.




(b) Blocked cell at step 11.




(c) Tunnelling shown.

Figure 4.3.: von Neumann neighbourhood, full simulation

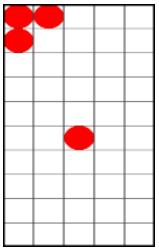
## 4.5. Constraints on grid height in built-in neighbourhood methods

In order to identify if there is an optimal grid shape or size both in simulation and subsequently in hardware, simulations were carried out on differing shape rectangles. Long horizontally, square or long vertically. Both square and horizontal shapes completed successful simulations. However, the built-in neighbourhood methods did not work fully with the vertical grids, failing to fill cells with an Y coordinate greater than the maximum X coordinate. This is due to the the developers of MASON designing these built-in neighbourhood methods with an expectation that simulations would have a square shape. In a typical grid with width of five cells and a depth of ten cells we carried out von Neumann simulations as shown in 4.4a, 4.4b and Zig-Zag simulation in 4.4c.

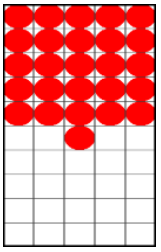
Figure 4.4a shows the starting cell with coordinates (2,5) and subsequent three steps in the von Neumann neighbourhood simulation. One must note that JAVA coordinates

start at zero rather than one and the Y axis is in the opposite direction to what is considered normal. Table 4.3 at the top of the table with start position (2,5), the next five items are the neighbourhood for this cell position. The next step's neighbourhood should be (1,5) from the first set rather than what is shown (1,0). Thus the Y coordinate is out of bounds as Y > X. The next two steps show normal neighbour selection . Figure 4.4b then shows the final step and if the simulation is allowed to run no further cells are occupied below Y = 4. On the other hand figure 4.4c is the Zig-Zag neighbourhood which starts in cell (0,0) and travels to the last cell as shown. It occupies each cell in turn and and leaves the cells a differing shaded grey depending on how long since they were visited.

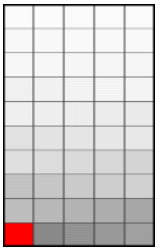
Both of these methods use similar code to generate the simulations with the exception of the neighbourhood method. Experiments have shown that the in-built methods are constrained to square and long horizontal grids.



(a) Simulation start and three steps, von Neumann.



(b) Simulation end, von Neumann.



(c) Simulation end, Zig-Zag.

Figure 4.4.: von Neumann and Zig-Zag neighbourhoods, full simulation

## 4.6. Random Number Generation

Random number generation is required in those simulations where the starting cell location is randomly chosen, particularly for those with multiple-agents. The random number generator used within MASON is derived from ECJ 27 a Java based evolutionary computation research system [44], and is the MersenneTwisterFast version. This is not a subclass of java.util.Random and cannot be used for multiple threads. Thus a

| Bounded = 5x10 Start position: (2, 5) |
|---|
| 0: ( 1, 5)<br>1: ( 2, 4)<br>2: ( 2, 5)<br>3: ( 2, 6)<br>4: ( 3, 5) |
| Start position: (1, 0) |
| 0: ( 0, 0)<br>1: ( 1, 0)<br>2: ( 1, 1)<br>3: ( 2, 0) |
| Start position: (0, 0) |
| 0: ( 0, 0)<br>1: ( 0, 1)<br>2: ( 1, 0) |

Table 4.3.: von Neumann Neighbourhood Bounded 5x10 grid, origin at cell (2,5).

copy of the MersenneTwisterFast and MersenneTwister classes have been included in the MASON Toolkit. MersenneTwister class is threadsafe and a direct replacement for java.util.Random. The MASON manual page 57, [45] recommends both of these rather than the default java.util.Random. Whilst java.util.Random is threadsafe, concurrent use over the same java.util.Random instance, suffers from contention if used with many threads, and consequent poor performance. Michaelis in [56] investigated Randomness of Java Runtime Libraries and found that the Java Development Kit (JDK) version ran slower and only passed 30 out of 114 black-box tests.

The MersenneTwisterFast and MersenneTwister classes are Java versions of the C-program for MT19937 Integer version created by Makoto Matsumoto and Takuji Nishimura [53]. These versions of the MersenneTwister are seeded with a 32 bit integer and if you set the seed to be the same for different runs with the same parameters then the agent start position created is the same each time. Therefore if high occupancy is required then preselecting the random seed chosen from previous simulations which have had agent starting positions that achieved high occupancy can be beneficial.

# 5. Simulation experiment using L-systems algorithm

Initial experiments used an L-system approach to examine process "growth". Experiment 0 describes this preliminary work, and why change to an agent-based approach was made.

## 5.1. Experiment 0 (a simulation using L-system algorithm.)

If we consider the simulation to represent a hardware layer and a virtual machine layer with a link between each, so, as the cell multiplication takes place the virtual list is a double linked list with knowledge of the cells to the left and right, also each cell will know the id of the state machine processor. This allows the insertion of additional cells between a cell and its creator whilst simultaneously adding new cells to the hardware state machines at the end of the processor list which in later versions will be hardware based upon different FPGA's. Here actual cells are being simulated, where each cell occupies a single core and thus when the simulation grows many cores are being occupied.

### D0L simple model

Figure 5.1 shows a model of D0L-systems for the development of a "fragment of a multicellular filament such as that found in the blue-green bacteria *Anabaena catenula*" [65], which is now to be used for the first attempt at a simulation model in software

to help identify the issues that may be relevant to the creation of a suitable parallel processing architecture.

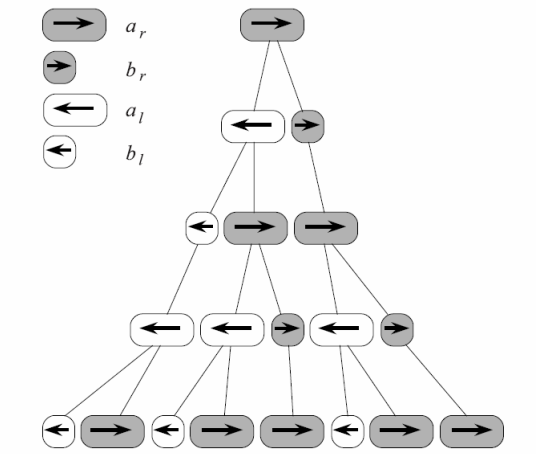



Figure 5.1.: Development of a filament (*Anabaena catenula* ) simulated using a D0L system. taken from, [65, fig. 1.4, p. 5].

The simulation is based upon a finite state automaton which uses the transactions shown to create the next generation of cells.

Transactions:

$$\omega : \mathrm{a_r}$$

$$\mathcal{P}_1 : \mathrm{a_r} \rightarrow \mathrm{a_l b_r}$$

$$\mathcal{P}_2 : \mathrm{a_l} \rightarrow \mathrm{b_l a_r}$$

$$\mathcal{P}_3 : \mathrm{b_r} \rightarrow \mathrm{a_r}$$

$$\mathcal{P}_4 : \mathrm{b_l} \rightarrow \mathrm{a_l}$$

Figures 5.2 and 5.3 helped in the program design. The activity diagram shows an outer loop for the generation cycles and an inner loop to select the cell to be processed by the inner state machine.

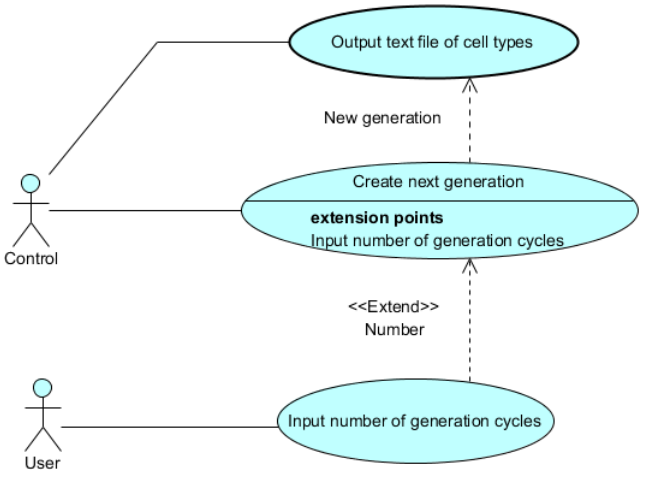



Figure 5.2.: UML Use case diagram for bacteria *Anabaena catenula* simulation.

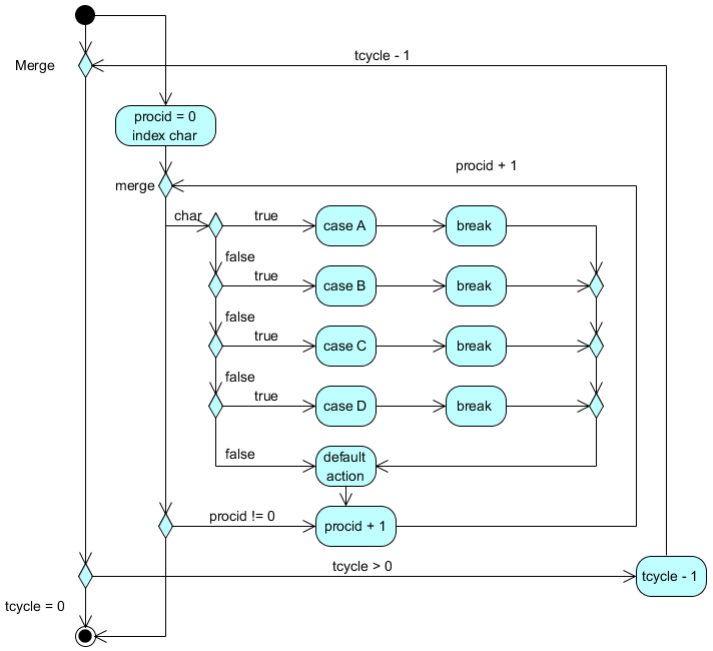



Figure 5.3.: UML activity diagram for two while loops, and an inner switch that forms the finite state automaton.

This simulation model is based upon the Java collection ArrayList which is a linked list inside an array. This allows the list to dynamically resize as the cells are generated at each turn and the total number of cells grows. Procid is the index for the content of ArrayList and matches the cell type to its processor position. In this simulation the cell types are generated but the process is sequential rather than parallel. This type of bacteria generally grows for ten generations and rarely up to a maximum of twenty generations therefore each process will only occupy a core for a relatively short time. Table 5.1 shows the output for a simulation run of seven generations starting from cell type "A". The simulation follows the Fibonacci series and produces correct cell growth for a maximum of thirty eight generations with 39088169 cells output. The program fails on inputs beyond thirty nine due to running out of memory despite running on a computer with eight Gigabyte of memory. hence the decision was made to move to an Agent-based simulation approach.

Table 5.1.: D0L run output for seven generations.

| procid: | cellType |
|---|---|
| 1 | A |
| 2 | C, B |
| 3 | D, A, A |
| 5 | C, C, B, C, B |
| 8 | D, A, D, A, A, D, A, A |
| 13 | C, C, B, C, C, B, C, B, C, C, B, C, B |
| 21 | D, A, D, A, A, D, A, D, A, A, D, A, A, D, A, D, A, A, D, A, A |
| 34 | C, C, B, C, C, B, C, B, C, C, B, C, C, B, C, B, C, C, B, C, B, C, C, B, C, B, C, C, B, C, B |

# 6. Simulation experiments that apply MASON Toolkit

## 6.1. Introduction

Table 6.1.: Glossary of simulation terminology

| |
|---|
| Simulation grid is the many-core system. |
| Cell is a single core of the many-core system. |
| Agent is a task or a software process or an application implemented on a single core. |
| Local neighbourhood is those neighbouring cells available for task mapping. |
| Some researchers refer to the local neighbourhood as a tile. |

In this chapter we use the MASON simulation toolkit as the simulation core for all experiments. In chapter 4 we identified several MASON default behaviours, one of which we referred to as "Tunnelling". This is where the local neighbourhood has no empty cores available within the hop distance and uses the last pair of x,y coordinates within the neighbourhood search as the next location to centre upon for the next search. Hence the central location is changed at each step even though no new task is mapped to a core. Until an empty core is found and a new task is applied to that core. If this is allowed then each simulation occupies all cores eventually. This defeats our investigation of the different local neighbourhood comparison. Therefore for all simulations a *finish* condition has been applied for when a local neighbourhood has no empty cells.

All of the grid cells represent a single core processor and each agent represents a single process their relationship within simulations is shown in table 6.1. Five different local neighbourhood algorithms are examined; three built-in and two we created. These are von Neumann, Moore, Radial, Zig-Zag and Selective Travel. Each of the simulations follows the Model-View-Controller (MVC) design pattern, which is used to separate the

application's concerns.

- **Model** - Model represents an object carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualisation of the data that the model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

The agent's can have three different behaviours move, grow and spawn. Not all simulations include all of these behaviours. Move is used to analyse behaviour. Whilst growth has several uses: firstly a time for the agent to take an action such as move, spawn new agents and or finally terminate. Spawning creates a number of new agents dependant on the available empty cores within the local neighbourhood.

- **Move** - Move an agent by relocating from current position to a new position within the local neighbourhood.
- **Grow** - Growth is shown by the agent's starting colour transitioning to a different colour over a number of time steps.
- **Spawn** - Spawn where an agent creates one or more new agents within available local neighbourhood empty cores.

Experiment one is the preliminary investigation of MASON to identify how it can be used in creating simulations. Zig-Zag, this experiment helped to identify and guide the simulation outcomes. Experiments, two starting with a single agent and three starting with multiple agents, allowed verification of how the size and shape of the simulation grid combined with the local neighbourhood affects the total cell occupancy. Finally experiment four combines all three areas of experiments one, two and three. It is a pre-cursor of possible future simulations for future research.

**Experimental set-ups**, there are two different approaches:

- Designated start position which only requires one run because it's deterministic and it is true all of the time.
- Random start position where 100 runs are done to gather statistics as each run is different.

It should be noted that in both JAVA and MASON the Y-axis is different to the expected norm in that the cell with coordinates **X = 0, Y = 0** is located at the top left hand corner. Y-axis increments positively in a downward direction.

## 6.2. Experiment 1 (Initial MASON investigation)

In this experiment two views have been provided for each simulation. The first contains the single agent presented as an oval which traverses over each cell as the simulation proceeds. The second is a rectangle and also has a trail from the start position to the final position. Experiment 1, examines a rule based algorithm for a single agent from a designated starting point through movement across the grid until unable to move any further. At each step the agent moves from the current location to a new designated location from a single cell neighbourhood for a hop of one. The Zig - Zag simulation is being used as a initial development tool and proof of concept. These two simulations have either a defined start point or a random start point. The designated start Zig-Zag is an initial calibration exercise and the random start is the first true experiment. In addition random placement of agents are more commonly used in the later experiments.

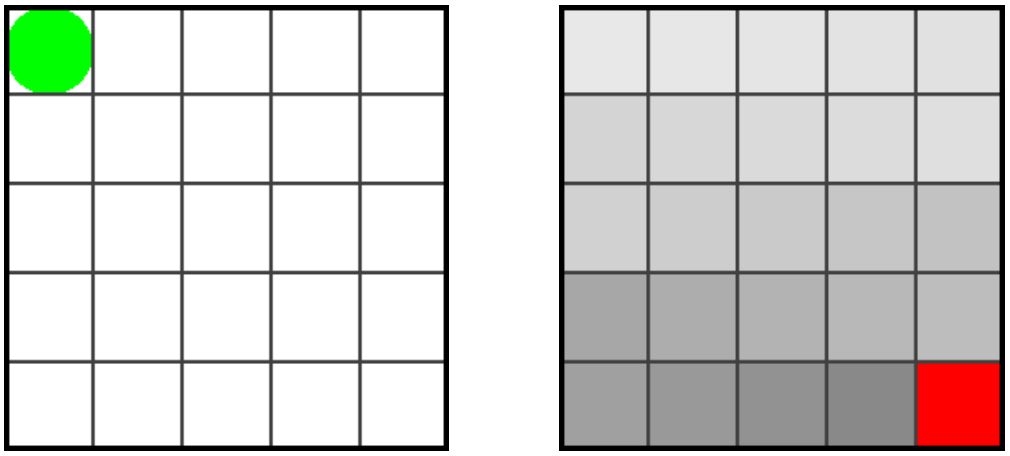
### 6.2.1. Zig - Zag designated start position

MASON is used here as a calibration exercise and identifies the features required for MVC. This simulations start coordinates are [0, 0] as shown in figure 6.1a. The neigh-

bourhood consists of a single cell defined by the following rules. Obviously starting in this position ensures that all of the cores are occupied one core at a time.

- If **Y** is even then **X = X + 1**
- else if **Y** is odd then **X = X - 1**
- If **X = MinX or MaxX** then **Y = Y + 1**
- When last core is reached, stop.

Figure 6.1b shows the path that the agent has followed with the shade of grey becoming lighter the further from the end position and the red rectangle represents an agent in the final position for this calibration exercise. Thus a verification of the environment is shown.



(a) Zig-Zag Neighbourhood starting position. (b) Zig-Zag full simulation path.

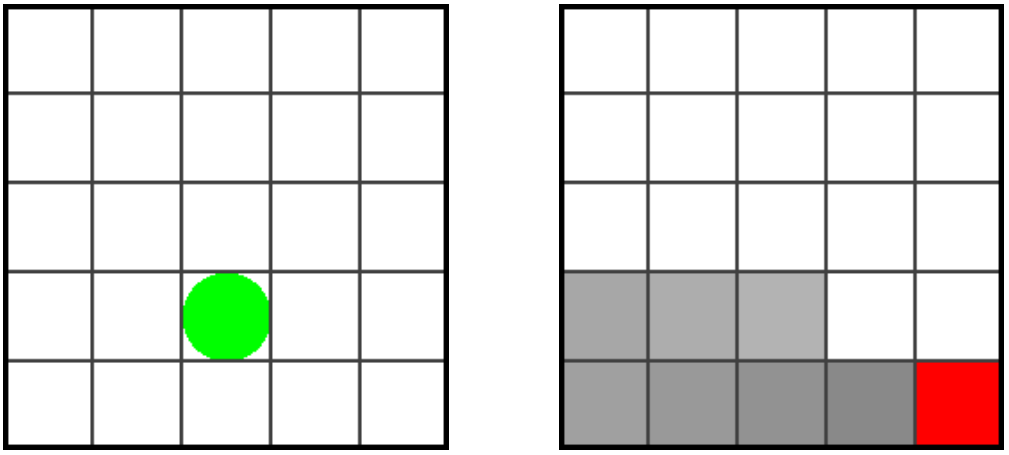
Figure 6.1.: Zig-Zag neighbourhoods with designated start position.

### 6.2.2. Zig - Zag random start position

This simulation uses the same rules as Zig - Zag designated start except for placement of the first agent which is a randomly selected core using the MersenneTwister random number generator described earlier [45]. The aim is to visit as many cores as possible and figures 6.2a and 6.2b show that this algorithm is flawed as there are more cells

unoccupied than occupied. This simulation gives a different start position each time it is run and does not accidentally fill the whole grid as the programmed travel direction is downwards from the start position. Generally there are more cells unoccupied than occupied. For the random start simulation over a large number of runs very few had a nearly full cell occupancy The random start position had an occupancy average of 72 cores for 100 runs. The main benefit of the random algorithm is the implementation of the random start position code which is used in the later experiments.

Both of these Zig-Zag simulations have provided the framework for the creation of simulations for the experiments that follow and is a proof of concept.



(a) Zig-Zag Neighbourhood starting position. (b) Zig-Zag full simulation path.

Figure 6.2.: Zig-Zag neighbourhoods with random start position.

## 6.3. Experiment 2 (Single agent spawning).

Experiment 2 gives consideration to the in-built neighbourhoods; von Neumann, Moore and Radial. The simulations all start with a single task randomly placed within the simulation grid. Subsequently at each time step a single new task is spawned into one of the empty cores in the local neighbourhood. That new task reorientates the local neighbourhood by becoming the central occupied core. Each local neighbourhood is different dependant upon the type of neighbourhood and the hop distance. Regardless of

the type of local neighbourhood there is an assumption that the cores are closer together and this is better than them being widely distributed within the simulation grid.

Firstly, to the Radial neighbourhood as there is a requirement of using whole cells and some of the radial boundary options use partial cells.

Initial inspection of the Radial algorithm is restricted to just identifying the number of neighbours that make a local neighbourhood for a hop of one for each of the different options. Subsequently simulations of a hop of one, two, four and eight have been carried out on von Neumann and Moore neighbourhoods in turn. These start with a single agent an at each time step replicate that agent within an empty core in the local neighbourhood. This continues until all of the cores are occupied or there are no empty cores within the neighbourhood that can be reached by the active agent which would result in the simulation being terminated.

The same algorithms are applied to a number of differing size grids.

Figures 6.4a, 6.4b, 6.8a and 6.8b show the neighbouring cores in von Neumann and Moore local neighbourhood for a hop of one and a hop of two. These cores are those available for either placement of a new agent or existing agent to be relocated into an unoccupied core. Grids are five by five and the starting core is located at coordinates [2, 2]. The numbers in the von Neumann figures 6.4a, 6.4b and Moore figures 6.8a, 6.8b indicate the order in which the search for an empty core occurs. All grids occupancy are shown as heatmaps. von Neumann, hop of one and Moore, hops of one, two, four and eight are shown in *appendix* B. Whilst the von Neumann heatmaps for hops of two, four and eight are contained in the von Neumann section 6.3.2 below.

### 6.3.1. Radial neighbourhood

The getRadialLocations is different to the von Neumann and Moore neighbourhoods as the neighbouring cells are from a circular area based upon the real-valued radius. There are three measurement rules to define how the cells fall within the boundary of the circle. Figure 6.3 is extracted from the MASON manual [45] showing all of the

boundary options described below. .

1. Grid2D.ANY, some part of the cell must fall within the circle boundary.

2. Grid2D.CENTER, the centre of the cell must fall within the circle boundary.

3. Grid2D.ALL, the entire cell must fall within the boundary of the circle.

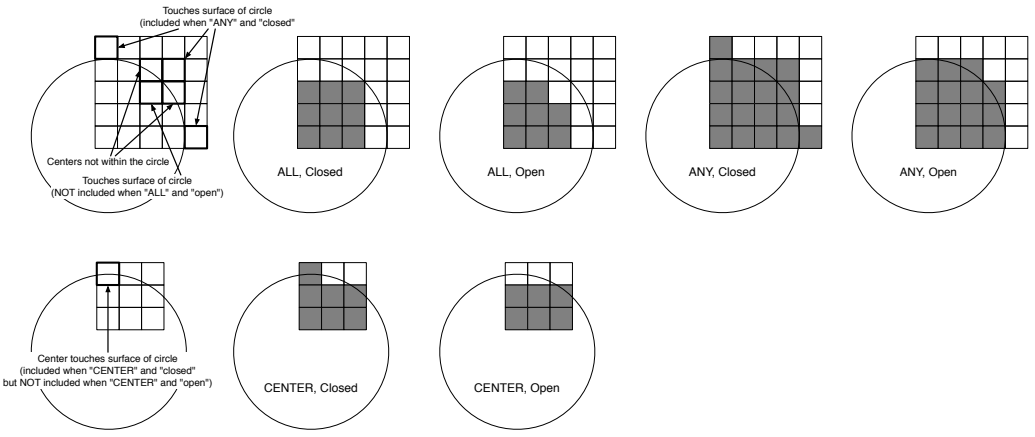



Figure 6.3.: Examples of certain cells falling within the region returned by the method getRadialLocations(...) with various settings of measurement rule (ALL, ANY, or CENTER) and closed-ness (open, closed). Figure from MASON Manual [45], page 123.

Mason provides an additional consideration, 'Closedness', defining what the circle boundary does in relation to a point lying exactly on the circle boundary. If true then the circle is closed and any point on the outer edge of the circle is within the boundary, when if false the circle is open and any similar point is outside the boundary.

The radial simulation for a hop of one investigates which of any of the boundary options are suitable. Each of the six options have been tried with a single agent within a 5x5 grid and designated location of x=2 and y=2. *Appendix* A.2 shows the code for all three of the Grid.2D rules with true Closedness. Table 6.2 is the local neighbourhood for all Grid.2D radial rules with both true and false closedness for comparison and the cells coordinates. This table is also the results for this part of the investigation.

| | True | False | True | False | True | False |
|---|---|---|---|---|---|---|
| | Any | Any | Center | Center | All | All |
| 0 | (3, 3) | (3, 3) | (3, 2) | (2, 2) | (3, 3) | (3, 3) |
| 1 | (3, 2) | (3, 2) | (2, 3) | | (3, 2) | (3, 2) |
| 2 | (3, 1) | (3, 1) | (2, 2) | | (3, 1) | (3, 1) |
| 3 | (2, 3) | (2, 3) | (2, 1) | | (2, 3) | (2, 3) |
| 4 | (1, 3) | (1, 3) | (1, 2) | | (2, 2) | (2, 2) |
| 5 | (1, 1) | (1, 1) | | | (1, 3) | (1, 3) |
| 6 | | | | | (1, 1) | (1, 1) |

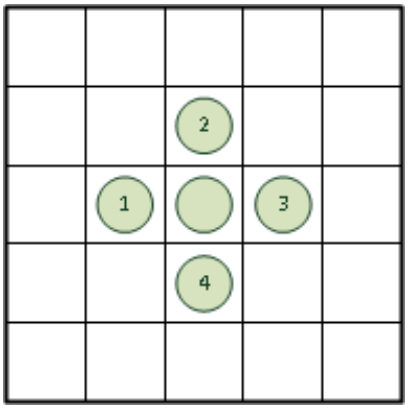
Table 6.2.: Radial neighbourhood, Bounded 5x5 Grid, Object Centre: (2,2):Radius 1.

For these examples of the three rules, CENTER has the lowest number of cells with the false choice for closedness having a single cell within the neighbourhood. Rule ALL has the greatest number of cells for both true and false whilst rule ANY has one cell less than rule ALL in their neighbourhood.

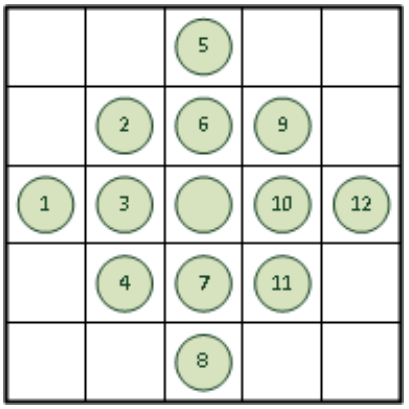
### 6.3.2. von Neumann neighbourhood

The von Neumann neighbourhood for a hop of one, has four cells in the shape of a cross as shown in figure 6.4a. Whilst the neighbourhood for a hop of two has twelve cells as in figure 6.4b. Hops of four have forty one cells as in figure 6.4b and eight have one hundred and forty five cells as in figure 6.4b respectively. Each of these neighbourhoods have the selection ordering within each node.

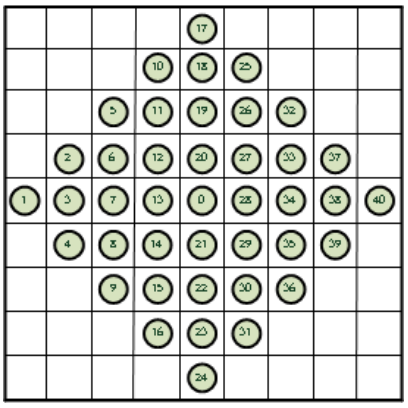
von Neumann simulations start with a single agent and each simulation uses different hop sizes. There are nine simulations with a designated start position. Each simulation is only run once as the final outcome cannot change. Also a random selected start position is used for one hundred runs and the arithmetic mean of these runs is reported in the heatmap. The current agent spawns a single new agent at each time step. The newly spawned agent's coordinates takes on the role of local neighbourhood central location for the next step. Further time steps continue until no empty cores can be found in the local neighbourhood. Simulation's end when either there are no empty cores within the



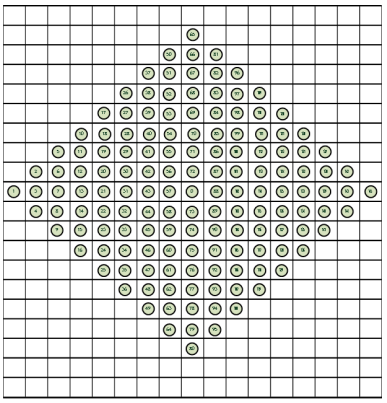

(a) Neighbourhood for a hop of 1.




(b) Neighbourhood for a hop of 2.



(c) Neighbourhood for a hop of 4.



(d) Neighbourhood for a hop of 8.

Figure 6.4.: von Neumann neighbourhoods.

local neighbourhood or all of the simulation grid cores are occupied.

Simulation runs for hops of one, two and four can be inspected in film *vN*1_2_4e.*wmv* with a link to its location in Google Drive *appendix D*. It can be seen that in this film each simulation successfully occupies all cells.

In the simulations the larger hops locate empty cores more readily and where an agent cannot find an empty core the simulation is terminated.

A heatmap is a graphical representation of the grid core occupancy where pale yellow are low numbers of cores transitioning to dark blue which are high numbers of cores. The actual numbers for each box are also included. There are four heatmaps comprising of sixteen different grid shapes with nine designated starting points and an average of one hundred random starting points for both the default ObjectGrid2D and the extended MyObjectGrid2D as described within section 4.3. Comparison of the heat maps for a

hop of one for both versions of the grid structure can be seen in *appendix* B.1. Heat maps for a hop of two, four and eight are shown in figures 6.5, 6.6 and 6.7.

As new agents are added the local neighbourhood is redefined, centring on the new agent that was added. All simulation runs shown in the heatmaps mentioned above start at the either the same designated start location or a random location depending upon which combination of start location and grid shape has been chosen. Most of the MyObjectGrid2D are superior to the default ObjectGrid2D, due to MyObjectGrid2D maintaining the correct order of the local neighbourhood array.

(a) Heatmap for default ObjectGrid2D, neighbours distance two cell.

| Alg 22 | vN | Neighbourhood | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 822 | 900 | 900 | 883 | 894 | 889 | 889 | 887 | 895 | 893 |
| | 45*20 | | 897 | 900 | 897 | 889 | 896 | 158 | 147 | 900 | 900 | 894 |
| | 60*15 | | 821 | 900 | 895 | 900 | 895 | 900 | 116 | 900 | 900 | 900 |
| | 90*10 | | 747 | 900 | 900 | 900 | 891 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 659 | 900 | 900 | 897 | 183 | 900 | 900 | 900 | 900 | 146 |
| 40*40 | | 1600 | 1454 | 1584 | 1586 | 1598 | 1597 | 1600 | 286 | 1600 | 1591 | 1598 |
| | 50*32 | | 1502 | 1600 | 1597 | 1599 | 1585 | 1600 | 1600 | 1600 | 1600 | 1585 |
| | 80*20 | | 1451 | 1600 | 1557 | 1597 | 1600 | 1600 | 156 | 1600 | 1600 | 1560 |
| | 100*16 | | 1434 | 1600 | 1600 | 1600 | 1594 | 1600 | 133 | 16000 | 1600 | 1600 |
| | 160*10 | | 1062 | 1600 | 1600 | 1594 | 1597 | 1600 | 106 | 1600 | 1600 | 1600 |
| | 320*5 | | 999 | 1600 | 275 | 1600 | 227 | 1600 | 111 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2475 | 2500 | 2497 | 2499 | 2469 | 2500 | 2500 | 2500 | 2500 | 2479 |
| | 125*20 | | 2261 | 2500 | 2499 | 2497 | 2491 | 198 | 2497 | 2500 | 2500 | 2499 |
| | 100*25 | | 2312 | 2500 | 2497 | 2495 | 2493 | 2496 | 196 | 2496 | 2500 | 2427 |
| | 250*10 | | 1611 | 2500 | 2500 | 2498 | 2498 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 1552 | 2500 | 320 | 2500 | 317 | 2500 | 156 | 2500 | 2500 | 2500 |

(b) Heatmap for extended MyObjectGrid2D, neighbours distance two cell.

| Alg 22m | vN | Neighbourhood | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 900 | 900 | 900 | 900 | 893 | 900 | 900 | 900 | 900 | 900 |
| | 45*20 | | 900 | 900 | 900 | 900 | 895 | 900 | 900 | 900 | 900 | 900 |
| | 60*15 | | 900 | 900 | 900 | 900 | 897 | 900 | 900 | 900 | 900 | 900 |
| | 90*10 | | 900 | 900 | 900 | 900 | 898 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 899 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1590 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 50*32 | | 1600 | 1600 | 1600 | 1600 | 1592 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 80*20 | | 1600 | 1600 | 1600 | 1600 | 1595 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 100*16 | | 1600 | 1600 | 1600 | 1600 | 1593 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1598 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1599 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2495 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 125*20 | | 2500 | 2500 | 2500 | 2500 | 2495 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2500 | 2500 | 2500 | 2500 | 2494 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 250*10 | | 2500 | 2500 | 2500 | 2500 | 2498 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2499 | 2500 | 2500 | 2500 | 2500 | 2500 |

Figure 6.5.: von Neumann neighbourhood, Occupied cells for each simulation grid version.

Heatmap for a hop of two is shown with the default ObjectGrid2D in figure 6.5a and the extended MyObjectGrid2D in figure 6.5b. In comparison with each other figure 6.5b has more starting positions that are fully occupied than figure 6.5a. Thus MyObjectGrid2D is the better choice here.

(a) Heatmap for default ObjectGrid2D, neighbours distance four cell.

| Alg 24 | vN | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 896 | 900 | 897 | 897 | 900 | 896 | 900 | 898 | 896 | 897 |
| | 45*20 | | 896 | 900 | 900 | 900 | 900 | 900 | 896 | 897 | 897 | 897 |
| | 60*15 | | 898 | 900 | 896 | 900 | 900 | 897 | 900 | 897 | 900 | 895 |
| | 90*10 | | 898 | 898 | 900 | 900 | 900 | 900 | 900 | 896 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1571 | 1571 | 1596 | 1569 | 1595 | 1579 | 1584 | 1593 | 1596 | 1598 |
| | 50*32 | | 1589 | 1598 | 1575 | 1600 | 1579 | 1582 | 1592 | 1593 | 1596 | 1600 |
| | 80*20 | | 1595 | 1598 | 1600 | 1598 | 1589 | 1589 | 1600 | 1588 | 1590 | 1597 |
| | 100*16 | | 1598 | 1600 | 1597 | 1600 | 1600 | 1600 | 1598 | 1600 | 1597 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1598 | 1598 | 1600 | 1596 | 1598 | 1600 | 1597 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2474 | 2455 | 2495 | 2448 | 2469 | 2414 | 2496 | 2392 | 2494 | 2476 |
| | 125*20 | | 2497 | 2496 | 2497 | 2500 | 2492 | 2500 | 2500 | 2492 | 2495 | 2500 |
| | 100*25 | | 2494 | 2494 | 2488 | 2496 | 2490 | 2490 | 2495 | 2494 | 2500 | 2486 |
| | 250*10 | | 2500 | 2498 | 2498 | 2500 | 2500 | 2498 | 2500 | 2500 | 2496 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

(b) Heatmap for extended MyObjectGrid2D, neighbours distance four cell.

| Alg 24m | vN | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 45*20 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 60*15 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 90*10 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 50*32 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 80*20 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 16000 |
| | 100*16 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 125*20 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 250*10 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

Figure 6.6.: von Neumann neighbourhood, Occupied cells for each grid version.

Heatmap in figure 6.6 for a hop of four is shown with the default ObjectGrid2D in figure 6.6a and the extended MyObjectGrid2D in figure 6.6b. In comparison with each other figure 6.6b has all starting positions fully occupied whilst figure 6.6a has very high values but not as good. Thus MyObjectGrid2D is the best choice here.

(a) Heatmap for default ObjectGrid2D, neighbours distance eight cells.

| Alg 28 | vN | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 896 | 897 | 897 | 891 | 894 | 897 | 898 | 892 | 894 | 895 |
| | 45*20 | | 900 | 898 | 898 | 900 | 900 | 898 | 900 | 900 | 900 | 900 |
| | 60*15 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 898 |
| | 90*10 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1600 | 1600 | 1598 | 1600 | 1600 | 1600 | 1600 | 1598 | 1600 | 1600 |
| | 50*32 | | 1598 | 1600 | 1594 | 1594 | 1598 | 1600 | 1598 | 1598 | 1598 | 1598 |
| | 80*20 | | 1598 | 1600 | 1600 | 1600 | 1600 | 1600 | 1598 | 1598 | 1600 | 1598 |
| | 100*16 | | 1598 | 1600 | 1598 | 1599 | 1600 | 1600 | 1600 | 1598 | 1600 | 1598 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1598 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2500 | 2498 | 2494 | 2495 | 2500 | 2486 | 2487 | 2498 | 2498 | 2500 |
| | 125*20 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2498 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2498 | 2500 | 2498 | 2500 | 2498 | 2500 | 2500 | 2500 | 2498 | 2500 |
| | 250*10 | | 2494 | 2500 | 2498 | 2500 | 2498 | 2500 | 2500 | 2498 | 2498 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

(b) Heatmap for extended MyObjectGrid2D, neighbours distance eight cells.

| Alg 28m | vN | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 45*20 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 60*15 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 90*10 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 50*32 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 80*20 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 100*16 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 125*20 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 250*10 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

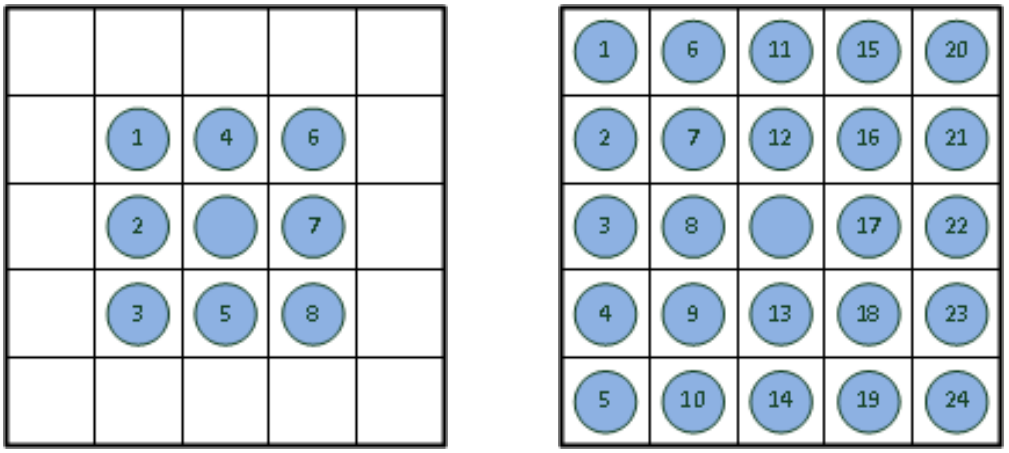
Figure 6.7.: von Neumann neighbourhood, Occupied cells for each grid version.

Heatmap in figure 6.7 for a hop of eight is shown with the default ObjectGrid2D in figure 6.7a and the extended MyObjectGrid2D in figure 6.7b. In comparison with each other figure 6.7b has all starting positions fully occupied whilst figure 6.7a is very close to fully occupied but not quiet as good. Thus MyObjectGrid2D is the best choice here.

It has become obvious that as the local neighbourhoods became larger in size fuller occupancy occurs. However, it is clear that MyObjectGrid2D provided better outcomes for all three hop sizes.

### 6.3.3. Moore neighbourhood

The Moore neighbourhood has eight cells surrounding the start cell for a hop of one as shown in figure 6.8a, while the neighbourhood for a hop of two has twenty four cells as figure 6.4b. Hops of four and eight have eighty and one hundred and ninety five cells respectively but are not shown here. The numbers within the cells for figures 6.8a, and 6.8b are the order in which the next empty cell is chosen with the blank central cell as the origin for this local neighbourhood.




(a) Neighbourhood for a hop of 1. (b) Neighbourhood for a hop of 2.

Figure 6.8.: Moore neighbourhoods.

There is also a film for the Moore neighbourhood with the same designated start cell as the von Neumann runs for comparison. Again, the film *M*1_*2*_*4e.wmv* shows that the larger hop more readily selects empty cells. Link to film location in Google Drive, *appendix D*. Also, the same runs were made for the grid structures with designated and random starts as mentioned previously in the von Neumann section. Comparison of the heat maps for hops of one for both versions of the grid structure are shown in the *appendix* B, heatmaps B.2.

(a) Heatmap for default ObjectGrid2D, neighbours distance two cell.

| Alg 6-2 | Moore | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 678 | 864 | 500 | 507 | 507 | 823 | 877 | 872 | 321 | 823 |
| | 45*20 | | 824 | 874 | 886 | 875 | 874 | 891 | 889 | 871 | 890 | 887 |
| | 60*15 | | 891 | 889 | 882 | 881 | 884 | 895 | 894 | 889 | 892 | 891 |
| | 90*10 | | 895 | 893 | 897 | 897 | 898 | 896 | 892 | 896 | 896 | 892 |
| | 150*6 | | 731 | 898 | 56 | 93 | 93 | 93 | 56 | 898 | 898 | 56 |
| 40*40 | | 1600 | 1505 | 1562 | 1303 | 1551 | 1545 | 1586 | 1525 | 1524 | 1590 | 1528 |
| | 50*32 | | 1574 | 1531 | 1564 | 1537 | 1575 | 1583 | 1584 | 1576 | 1586 | 1589 |
| | 80*20 | | 1578 | 1585 | 1597 | 1563 | 1576 | 1588 | 1591 | 1551 | 1591 | 1594 |
| | 100*16 | | 1590 | 1596 | 1592 | 1590 | 1592 | 1589 | 1583 | 1594 | 1594 | 1589 |
| | 160*10 | | 1594 | 1596 | 1593 | 1584 | 1584 | 1596 | 1593 | 1593 | 1593 | 1598 |
| | 320*5 | | 697 | 16 | 96 | 1600 | 1600 | 1600 | 96 | 16 | 16 | 96 |
| 50*50 | | 2500 | 2027 | 2029 | 2415 | 2383 | 2425 | 2455 | 2460 | 2406 | 2448 | 2394 |
| | 125*20 | | 2416 | 2494 | 2486 | 2475 | 2475 | 2489 | 2497 | 2497 | 2489 | 2489 |
| | 100*25 | | 2473 | 2487 | 2486 | 2399 | 2401 | 2472 | 2492 | 2460 | 2471 | 2478 |
| | 250*10 | | 2495 | 2496 | 2497 | 2497 | 2498 | 2496 | 2492 | 2493 | 2493 | 2492 |
| | 500*5 | | 878 | 16 | 141 | 2500 | 2500 | 2500 | 141 | 16 | 16 | 141 |

(b) Heatmap for extended MyObjectGrid2D, neighbours distance two cell.

| Alg 62m | Moore | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 900 | 900 | 900 | 900 | 892 | 900 | 900 | 900 | 900 | 900 |
| | 45*20 | | 900 | 900 | 900 | 900 | 895 | 900 | 900 | 900 | 900 | 900 |
| | 60*15 | | 900 | 900 | 900 | 900 | 896 | 900 | 900 | 900 | 900 | 900 |
| | 90*10 | | 900 | 900 | 900 | 900 | 897 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 898 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1590 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 50*32 | | 1600 | 1600 | 1600 | 1600 | 1592 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 80*20 | | 1600 | 1600 | 1600 | 1600 | 1595 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 100*16 | | 1600 | 1600 | 1600 | 1600 | 1596 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1597 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1599 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2487 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 125*20 | | 2500 | 2500 | 2500 | 2500 | 2495 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2500 | 2500 | 2500 | 2500 | 2494 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 250*10 | | 2500 | 2500 | 2500 | 2500 | 2497 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2499 | 2500 | 2500 | 2500 | 2500 | 2500 |

Figure 6.9.: Moore neighbourhood, Occupied cells for each grid version.

Heatmaps in figure 6.9 for a hop of two is shown with the default ObjectGrid2D in figure 6.9a and the extended MyObjectGrid2D in figure 6.9b. In comparison with each other figure 6.9b has nearly all starting positions fully occupied whilst figure 6.9a has very variable occupancy. Thus MyObjectGrid2D is the better choice here.

(a) Heatmap for default ObjectGrid2D, neighbours distance four cell.

| Alg 64 | Moore | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 881 | 884 | 884 | 878 | 869 | 889 | 888 | 880 | 877 | 884 |
| | 45*20 | | 892 | 892 | 892 | 894 | 893 | 892 | 893 | 892 | 892 | 892 |
| | 60*15 | | 896 | 896 | 896 | 897 | 896 | 896 | 896 | 896 | 893 | 896 |
| | 90*10 | | 896 | 896 | 896 | 897 | 896 | 896 | 896 | 896 | 896 | 896 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1574 | 1582 | 1581 | 1542 | 1540 | 1586 | 1564 | 1570 | 1582 | 1582 |
| | 50*32 | | 1580 | 1584 | 1573 | 1575 | 1574 | 1573 | 1589 | 1575 | 1590 | 1574 |
| | 80*20 | | 1591 | 1592 | 1592 | 1591 | 1590 | 1592 | 1592 | 1592 | 1592 | 1590 |
| | 100*16 | | 1596 | 1596 | 1596 | 1594 | 1594 | 1596 | 1596 | 1596 | 1596 | 1596 |
| | 160*10 | | 1597 | 1598 | 1598 | 1597 | 1596 | 1596 | 1598 | 1596 | 1596 | 1598 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2455 | 2463 | 2484 | 2478 | 2456 | 2483 | 2483 | 2449 | 2476 | 2483 |
| | 125*20 | | 2492 | 2492 | 2492 | 2494 | 2493 | 2492 | 2492 | 2492 | 2492 | 2492 |
| | 100*25 | | 2489 | 2492 | 2479 | 2494 | 2492 | 2492 | 2481 | 2488 | 2488 | 2481 |
| | 250*10 | | 2496 | 2496 | 2496 | 2497 | 2496 | 2496 | 2496 | 2496 | 2496 | 2496 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

(b) Heatmap for extended MyObjectGrid2D, neighbours distance four cell.

| Alg 64m | Moore | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 45*20 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 60*15 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 90*10 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 50*32 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 80*20 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 100*16 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 125*20 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 250*10 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

Figure 6.10.: Moore neighbourhood, Occupied cells for each grid version.

Heatmap in figure 6.10 for a hop of four is shown with the default ObjectGrid2D in figure 6.10a and the extended MyObjectGrid2D in figure 6.10b. In comparison with each other figure 6.10b has all starting positions fully occupied whilst figure 6.10b is close to fully occupied but not quiet as good. MyObjectGrid2D is the best choice here.

(a) Heatmap for default ObjectGrid2D, neighbours distance eight cells.

| Alg 68 | Moore | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 889 | 885 | 885 | 893 | 892 | 892 | 892 | 892 | 892 | 885 |
| | 45*20 | | 892 | 892 | 892 | 893 | 892 | 892 | 892 | 894 | 894 | 892 |
| | 60*15 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 90*10 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1582 | 1581 | 1584 | 1566 | 1566 | 1584 | 1581 | 1579 | 1564 | 1581 |
| | 50*32 | | 1589 | 1592 | 1592 | 1593 | 1592 | 1592 | 1592 | 1592 | 1592 | 1592 |
| | 80*20 | | 1593 | 1594 | 1594 | 1593 | 1592 | 1594 | 1594 | 1592 | 1592 | 1594 |
| | 100*16 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2480 | 2476 | 2477 | 2478 | 2477 | 2476 | 2482 | 2477 | 2477 | 2476 |
| | 125*20 | | 2493 | 2492 | 2492 | 2493 | 2492 | 2492 | 2492 | 2494 | 2494 | 2492 |
| | 100*25 | | 2492 | 2492 | 2492 | 2493 | 2492 | 2492 | 2492 | 2492 | 2492 | 2492 |
| | 250*10 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

(b) Heatmap for extended MyObjectGrid2D, neighbours distance eight cells.

| Alg 68m | Moore | Neighbourhood | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 45*20 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 60*15 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 90*10 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| | 150*6 | | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 | 900 |
| 40*40 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 50*32 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 80*20 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 100*16 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 160*10 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 125*20 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 250*10 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 500*5 | | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 | 2500 |

Figure 6.11.: Moore neighbourhood, Occupied cells for each grid version.

Heatmap in figure 6.11 for a hop of eight is shown with the default ObjectGrid2D in figure 6.11a and the extended MyObjectGrid2D in figure 6.11b. In comparison with each other figure 6.11b has all starting positions fully occupied whilst figure 6.11a is very close to fully occupied but not as good. Thus MyObjectGrid2D is the best choice here.

The extended class is shown to have preferable outcomes to that of the default grid class, due to the correct order of the neighbour coordinates which keep the last cores on

the outside edge of the local neighbourhood, this experiment has shown that both von Neumann and Moore neighbourhoods have comparable occupancy. Regardless of grid size or start location for a hop of two or more is very successful.

## 6.4. Experiment 3 (Multiple agent spawning).

This experiment focusses on the exploration of three in-built local neighbourhoods and an additional neighbourhood that is rule based and each of these have more than one agent placed randomly on initialisation. The same random number seed when used with the same simulation parameters always returns the same agent start positions.

The many-core system will have multiple tasks or applications in operation at the same time and this simulation is closer to this. At each subsequent step, they each create another agent in an empty core within their current local neighbourhood. The built-in neighbourhood's simulation terminate if any single agent becomes unable to locate a new empty core within their local neighbourhood. The additional rule based neighbourhood is called Selective Travel and uses a different grid class. Which allows some agents to continue even if one has stopped.

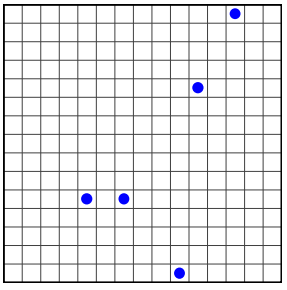
### 6.4.1. Radial multi agent

For the radial local neighbourhood six simulations are undertaken. All simulations have the form .ALL and Closedness is False and table 6.2 shows these option choices have the most complete cells within the neighbourhood.

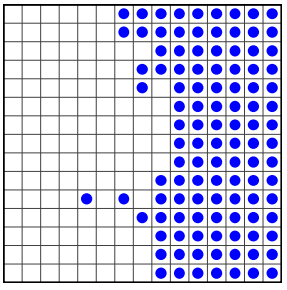
There are five agents randomly placed within the grid. The local neighbourhood has a hop of four. Figure 6.12a has a random seed of three and includes the starting cell positions and figure 6.12b the simulation end with the number of cells occupied. As all of the agents are the same colour it was difficult to identify how many and which agents were unable to find an empty core. To better understand why figure 6.12b ends with so many cells unoccupied. an image of each step with each of the five agents

coloured differently is shown in *appendix* C.1. Figure 6.12c has the same simulation parameters as used in figure 6.12a but with a different random seed of four. The end image figure 6.12d shows one hundred percent occupancy. This particular simulation was run many times whilst incrementing the random seed for each run. It was observed that some of those with an odd random number in particular produced start positions that ended before occupying all cores. This was the reason for choosing random seeds of three and four for these images shown in figure 6.12.

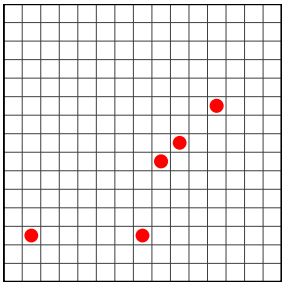
The film *R_all_H4_A5.wmv*, has a link to its location in Google Drive *appendix D*, shows several similar Radial neighbourhood simulations. Analysis of all of these radial simulations shows that random seed choice can be a critical to the start location of each agent. This start position can affect the final outcome by allowing one or more agent's to become distanced from the empty cores.



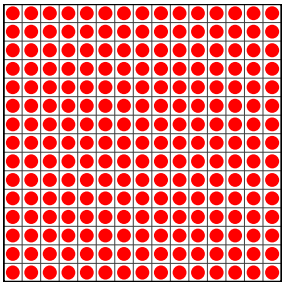
(a) Random seed of three, start.



(b) Random seed of three, end.



(c) Random seed of four, start.



(d) Random seed of four, end.

Figure 6.12.: Comparison of two Radial neighbourhood simulations, each with five agents and a hop of four. Figure 6.12a shows start positions for a random seed of three and figure 6.12c a random seed of four. The occupancy shown in figures 6.12b and 6.12d is quite different.

### 6.4.2. von Neumann multi agent

For the von Neumann neighbourhood figure 6.13 is the start position for four separate simulations each with the same random start position and three agents. Figures 6.14a, 6.14b, 6.14c, 6.14d show the final position for hops of one, two, four and eight.

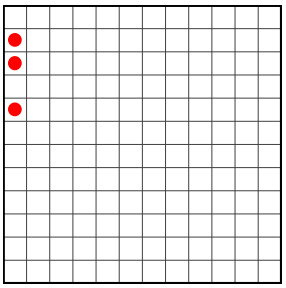
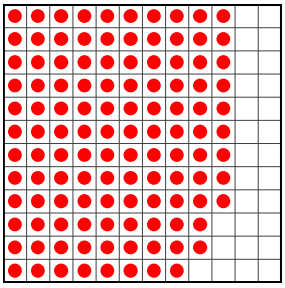
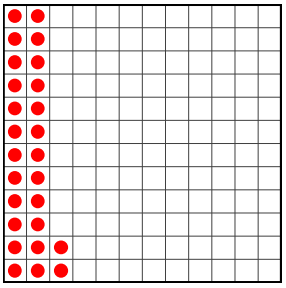


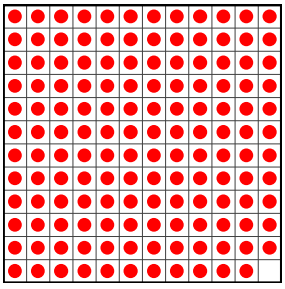
Figure 6.13.: von Neumann neighbourhood simulation with three agents and the same randomised start position for all four different hop sizes.



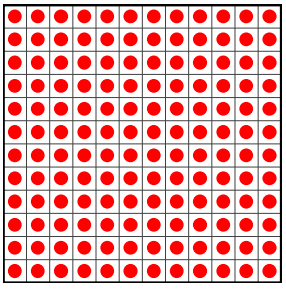
(a) von Neumann with a hop of one, end.



(b) von Neumann with a hop of two, end.



(c) von Neumann with a hop of four, end.



(d) von Neumann with a hop of eight, end.

Figure 6.14.: von Neumann neighbourhood simulations with three agents and hops of one, two, four and eight when finished .

In these four simulations it can be seen that the hop of two has early termination and reduced cell occupancy. This can be identified from figure 6.14b. As all the agents are

close to the left hand boundary the possible cores available are restricted. The agents travel in a downwards direction and when the bottom left hand cell is occupied the next empty cell is beyond the reach of that agent which terminates the simulation. The other three hops' agents more readily access much more cells due to the local neighbourhood becoming larger and allowing more empty core availability.

The film *vN_H2_A3.wmv*, has a link to its location in Google Drive *appendix D* has a simulation of three agents with a hop of two showing that this neighbourhood is very successful at accessing all of the available cells.

### 6.4.3. Moore multi agent

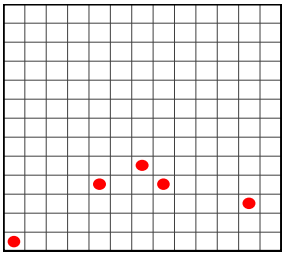
For the Moore neighbourhood figure 6.15 shows the start position for four separate simulations with each one having the same random start and five agents. Figures 6.16a, 6.16b, 6.16c, 6.16d show the final position for hops of one, two, four and eight.

The film *M_H2_8_A3_10.wmv*, has a link to its location in Google Drive *appendix D*, shows the full simulation run for both a hop of two and a hop of eight with three agents. The final section of this film includes a hop of eight with ten agents. Figure 6.16a shows a hop of one simulation and indicates that it produces reduced core occupancy. A Mann-Whitney U test showed that there was a significant difference (W = 235, p-value = 0.02145) between the even random number seed and the odd random number seed. The estimated effect size using Cohen's *d* calculation in R, shown in *Appendix* E, difference of *d* estimate: 0.5990212 (medium). Whereas the other three simulations access from most to all available cores in the neighbourhood.

### 6.4.4. Selective Travel multi agent

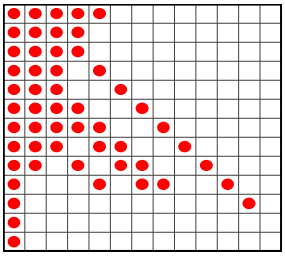
Selective travel simulation is a progression from the von Neumann and Moore multi agent simulations.

Rather than using ObjectGrid2d for this algorithm, SparseGrid2d is used because the

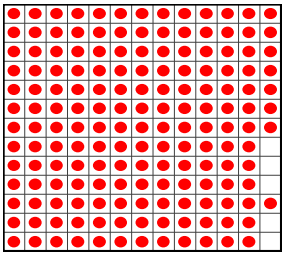


(a) Moore neighbourhood, five agents, randomised start positions.

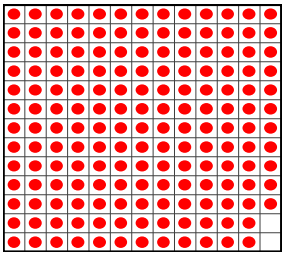
Figure 6.15.: Moore neighbourhood simulation with five agents and the same randomised start position for all four different hop sizes.



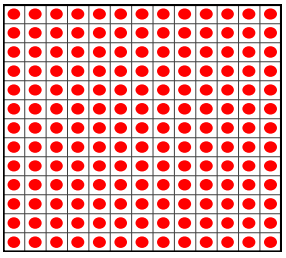
(a) Moore with a hop of one, end.



(b) Moore with a hop of two, end.



(c) Moore with a hop of four, end.



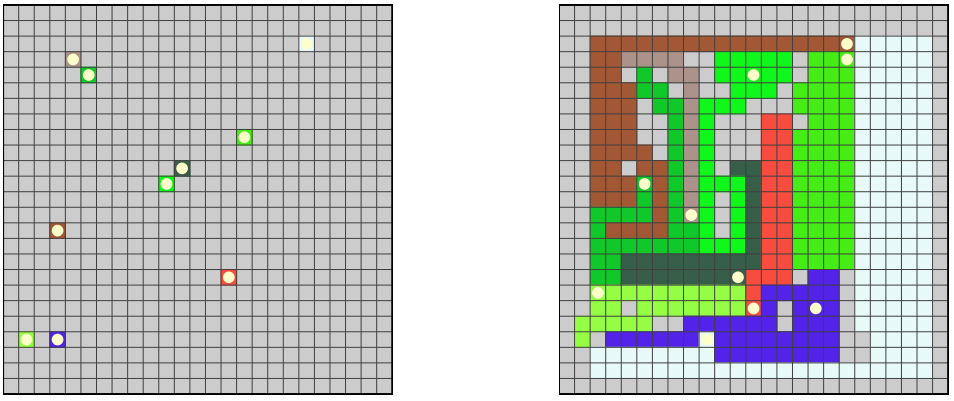
(d) Moore with a hop of eight, end.

Figure 6.16.: Moore neighbourhood simulations with five agents and hops of one, two, four and eight when finished .

coloured background counts as an object and the agent is also an object. ObjectGrid2D does not allow more than one object within a cell.

Each different colour denotes a particular agent grouping. The circular agent is the current cell selection for the background colour family grouping. Its position also identifies the possible future locations. This neighbourhood uses a one hop neighbourhood similar to von Neumann, that is at each time step after the start each agent examines the four directions.

The path directions are ordered from a maximum to a minimum number of empty cores. The direction with the longest empty path is selected for the next empty core. The agents act in series and if more than one agent chooses the same empty core then an agent is selected randomly to occupy that core with a new agent and the other will choose a different core if available. Should there be no empty cores, that agent finishes. The simulation continues until the last moving agent can move no further then the whole simulation ends. A sample simulation is shown in figure 6.17. The starting agents are the cream ovals and the coloured background is the complete core family for that agent. In the film *ST_H1_A10.wmv* the whole simulation can be observed with a link to its location in Google Drive *appendix D*.



(a) Selective travel neighbourhood start. (b) Selective travel neighbourhood end.

Figure 6.17.: Selective travel neighbourhoods with ten agents and a hop of one.

**Experiment 3 Conclusions**

It is assumed that all or nearly all cores should be occupied in each simulation. However, there are some which do not meet this criteria. In the simulations examined, the differing neighbourhoods all show similar behaviour when used for multi-agents. Random numbers for starting positions are influenced by the random number seed. In particular, there is a noticeable difference between odd and even number seeds as is shown in our statistical analysis in Appendix E. Thus careful selection of random number seed should be undertaken. These simulations also show that in general, bigger local neighbourhoods can access more cells and more readily occupy all available cells within

larger grids.

Selective Travel is the best of those tested, it has the ability to halt agents when they are blocked and allow others which can still move to do so. In addition the differing cores each agent occupies are identified with appropriate colouring. Lastly, is its relationship with von Neumann neighbourhood shape. von Neumann and Moore neighbourhoods are better than Radial of the built-in local neighbourhoods because Radial can identify partial cells within its neighbourhood and we are looking for whole cells. On the other hand both von Neumann and Moore neighbourhoods have very similar core occupancy.

For the proposed hardware, selection of the neighbourhood is a key parameter, as it provides the number of possible cell states and how many transition rules are available. Even with the small number of neighbourhoods examined, differences in symmetry and size of the neighbourhood can have greatly different results. Thus, depending upon the particular implementation neighbourhoods from both von Neumann and Moore could be suitable. The von Neumann neighbourhood always creates a common relationship between neighbours, whilst the Moore neighbourhood has multiple neighbour relationships, edges and corners, perhaps even different types of corners. From these relationships of neighbours it can be seen that each type of neighbour has a direct link to the number of physical connections that needs to be designed into any hardware.

## 6.5. Experiment 4 (Multiple agent moving, growing and spawning).

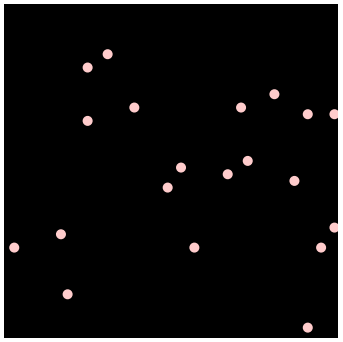
Experiment 4, is the most complex simulation undertaken. It brings together many of the threads of the current research and is an indication of future simulations for further research. This combines elements of the previous three experiments and ultimately ensures that all three actions can be carried out at the same time. It could be suggested

there should be an intermediate simulation that omits the move function but retains all other objectives. There are three different simulation runs, one Moore and two von Neumann. The two von Neumann ones have one similar to the Moore version and one on a much smaller grid to allow inspection of the transitions more clearly.

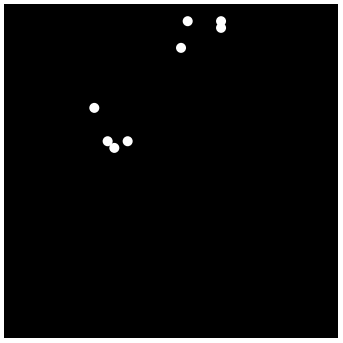
The rules for this algorithm are as follows:

1. Simulation start.
    - Agents have 1.0 units of processing power.
    - Cells have 0.0 units of processing power.
2. Simulation at each step.
    - Cells accumulate 0.1 units of processing power.
    - Agent moves to a local neighbouring cell.
    - Agent entering a cell has a transaction cost of minus 0.2 units of processing power.
    - Agent gathers from the cell available processing power up to a maximum of 1.0 unit.
    - Agent has a five percent chance of early termination and no new agents are created.
    - When Agent attains 10.0 units or greater of processing power:
        - Agent terminates and is removed.
        - New agents are created in the surrounding local neighbourhood
        - New agents have 0.0 units of processing power.

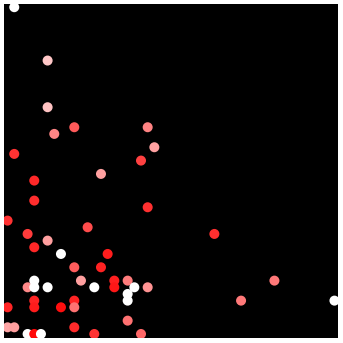
The neighbourhood bags of coordinates, occupancy and processing power for each agent are copied into temporary bags which are then sorted into higher to lower processing power. Hence when the agent is moved to a new core then the best one is chosen and if new agents are being created then the best cores are also chosen for them.



(a) Moore neighbourhood at step 1, agents initial placement.



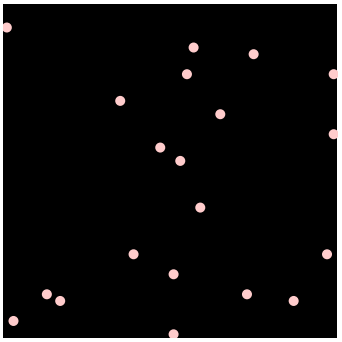
(b) Moore neighbourhood at step 36, first four new agents created from two terminated agents.



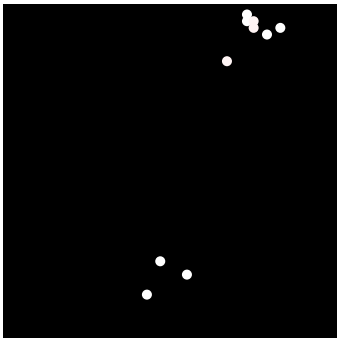
(c) Moore neighbourhood at step 500, agent colours indicate which are nearing termination. White is new agent, pink is transitioning and red is near termination.

Figure 6.18.: Moore neighbourhood simulations with twenty agents and a hop of four at different steps.

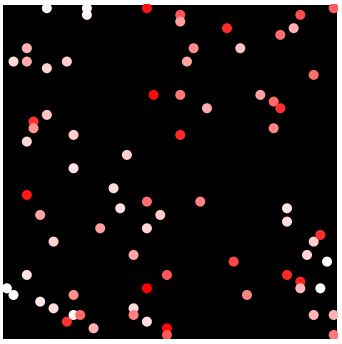
These three films have a link to their location in Google Drive *appendix D*. For the simulation film *M_H4_A50.wmv* fifty agents are placed in a ten thousand cells square grid. The neighbourhood is a hop of four and up to eight new agents are created upon agent termination. Whilst for the simulation film *vN7_H2_A50.wmv* the fifty agents



(a) von Neumann neighbourhood at step 1, agents initial placement.



(b) von Neumann neighbourhood at step 36, first ten new agents created from three terminated agents.



(c) von Neumann neighbourhood at step 500, agent colours indicate which are nearing termination. White is new agent, pink is transitioning and red is near termination.

Figure 6.19.: von Neumann neighbourhood simulations with twenty agents and a hop of four at different steps.

are placed in a two thousand five hundred cells square grid with a hop of two neighbourhood, up to five new agents are created as described above. Finally the smaller von Neumann simulation film *vN_H1_A5.wmv* has five agents starting in a one hundred cells square grid with a hop of one. It can also create up to five new agents but as the

neighbourhood only contains four cells this is the maximum that can be created at any one time.

Experiment 4 has all of the abilities that are ultimately required, multiple agents with the ability to relocate to different processors. At predefined stages, create new processes utilising the best processors that are available within a local neighbourhood. Finally the ability to transition to any available processor within the grid.

Figure 6.20 shows a system with two hundred and fifty thousand cores, starting with five hundred agents at an intermediate point within the simulation. It can be clearly seen that there are agents at different stages within their growth by their colour, starting at white then shades of pink to final red colour. In addition, the patches of agents indicates local neighbourhoods where red agents can terminate and new white agents are spawned.

In the field review, section 2.1.3, fault tolerance and energy efficiency from *dark silicon* are discussed, so within Figure 6.20 the black patches can be considered to be *dark silicon* and most of those cores are cooler than the occupied cores, although some of the cores may have had the agent transition to an empty core and be in the cooling down phase. When a defective core is identified it can be marked permanently as an occupied core and thus not used by any agent when the defective core appears within that agent's local neighbourhood as a method of gaining fault tolerance.

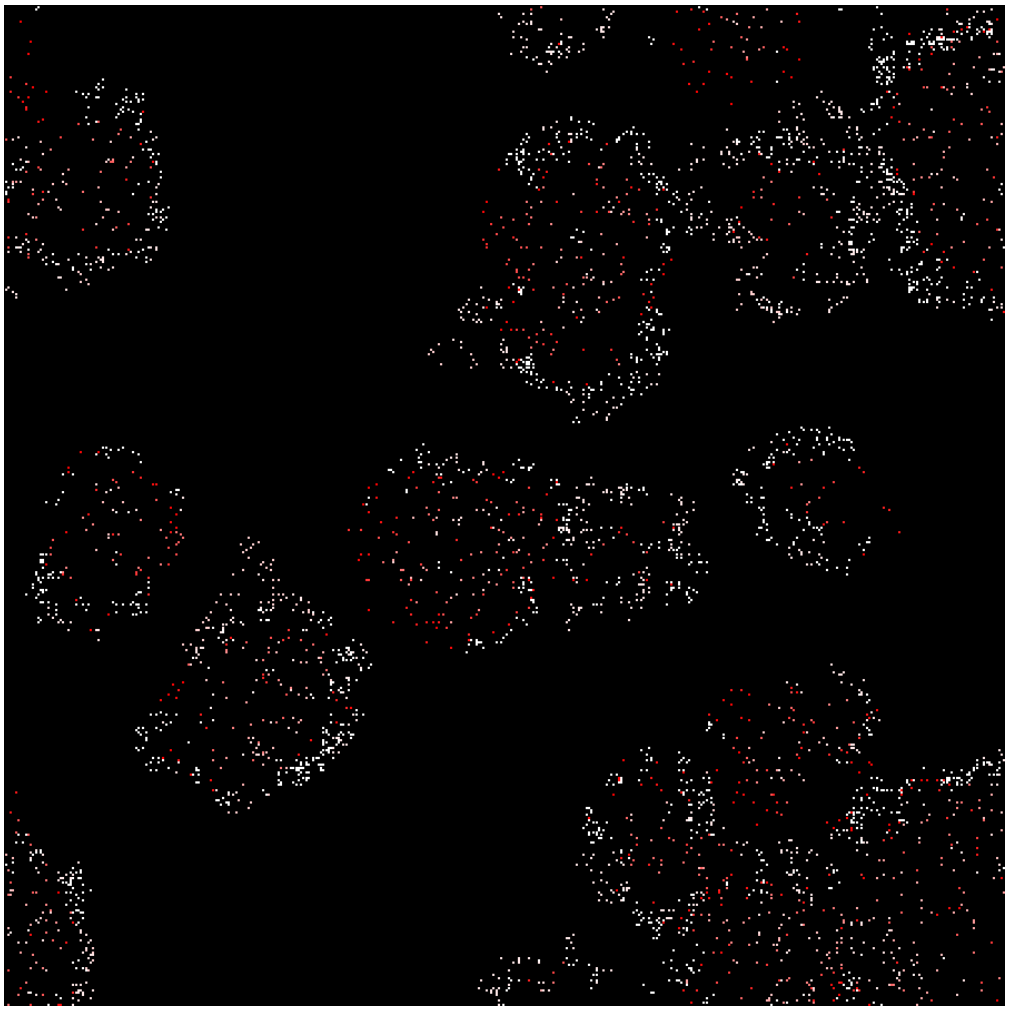


Figure 6.20.: von Neumann local neighbourhood simulation of a many-core system. Depicting agents at various stages and core occupancy. Each local neighbourhood covers those cores within a hop of four.

# 7. Conclusions and Future Work

## 7.1. Conclusions

In this thesis we have focused upon bio-inspired algorithms that may be applicable for direct implementation onto an FPGA. There are two main algorithms. Cellular automata, offer an effective massively fine grained parallel computation model and rewriting systems, of which, in particular L-systems. The self-replicating arena is widely based upon CA's but a few researchers are using L-systems in a similar way. The L-systems model is naturally suited for modelling growth processes and self-replication can be considered a special case of this. It can also be seen that both models could be combined to provide a very similar outcome to von Neumann, in that if the L-system was used to generate the string of characters which are then passed along a constructing arm to the CA implementation phase. However it does appear that neither of these systems are of a high enough level in the design chain for our possible usage. Whilst all of the local neighbourhood algorithms examined are capable of being implemented in a hardware environment some are more difficult than others. These algorithms should be considered in this order:

1. von Neumann local neighbourhood.
2. Moore local neighbourhood.
3. User created local neighbourhood.
4. Radial local neighbourhood.

von Neumann and Moore are considered both in the classical version of a single hop and larger hop versions. User created rules are algorithms that don't use the local

neighbourhoods in the same way as the others but may be derived from evolutionary algorithms.

## 7.2. Future Work

The experiments in chapter 6 have helped to identify good local neighbourhoods. The von Neumann and Moore neighbourhoods appear to be comparable in simulation.

The custom hardware known as SMBH has on-board four Xilinx Spartan 3E field programmable gate arrays each of which can be configured as master or slave. There are 8 of these boards connected by Ethernet. An Ethernet is a simple bus-like connection of wires which operates by 'carrier sensing' with collision detection known as (CSMA/CD) belonging to the class of contention bus networks. Access is managed by a medium access control (MAC) protocol. As a single link connects all hosts, the MAC protocol combines the functions of a data link layer protocol and a network protocol in a single protocol layer as in figure 7.1.


Figure 7.1.: Ethernet packet - physical layer, and Frame - data link layer.

The node to node interconnection can be considered as local neighbourhood within each FPGA. Four FPGAs as a local area network (LAN) on each board. A wide area network (WAN) over all the boards within the Intranet and an external connection to the Internet as presented in figure 7.4. This can be achieved by the use of a hybrid routing methodology, where the local local network uses a 2D-mesh topology and the wider network utilises Spidergon topology. The Spidergon STNoC allows the selection of these two routing strategies for a network.

Each of these papers have used either single FPGAs; [27], [30], [42], [46], [47], [81], [73], [58] and or multiple FPGAs; [25], [32], [64] to create a node by node grid. The size of the grids range from 3 by 3 up to 29 by 29; dependant upon the purpose of the

core. Also, the model of FPGA must be taken into consideration for the amount of cores created. As the model available on the SMBH is relatively small the number of cores are limited to the lower end of the scale. Thus, figures 7.2 and 7.3 display the smallest neighbourhood available. When hardware implementation is explored it's expected that the final grid size could be between 5 by 5 and 10 by 10.

As to which neighbourhood should be implemented in the custom hardware available for this purpose. Consideration of figure 7.2 and figure 7.3 highlights the problem in the circuit connectivity for the Moore neighbourhood. The wires for von Neumann can be implemented in a 2 layer circuit, the 1st layer could carry the horizontal wires and the 2nd layer the vertical wires. In the case of the Moore neighbourhood a 3rd or more layers is required to connect the 4 corners with the centre. However the current custom hardware does not allow for this. Thus the wires would need to be connected as depicted in figure 7.3. When the router look up table 7.1 and table 7.2 are compared, the Moore neighbourhood is more complex and has greater hop cost for many of the node to node connections.

| 1 | A | B | C | D | E | Out |
|---|---|---|---|---|---|---|
| A | 0 | • | • | • | • | • |
| B | • | 0 | | | | • |
| C | • | | 0 | | | • |
| D | • | | | 0 | | • |
| E | • | | | | 0 | • |
| Out | • | • | • | • | • | 0 |

Table 7.1.: Node to node router lookup table for FPGA 1 on Board A from figure 7.2, Out refers to routing beyond the local area network.

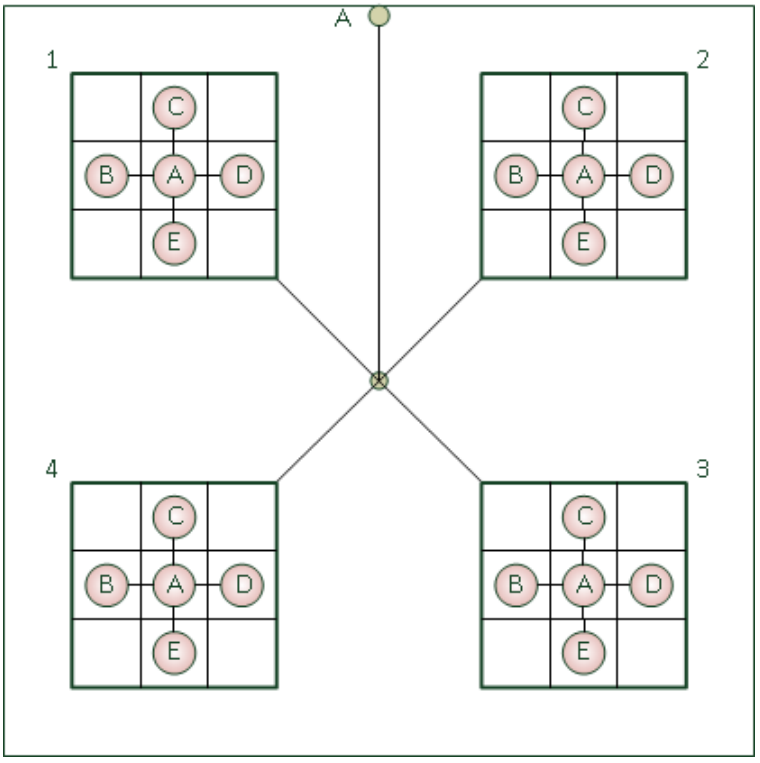



Figure 7.2.: SMBH custom board with 4 Spartan 3E FPGAs, von Neumann internal neighbourhood with Spidergon STNoC Architecture.

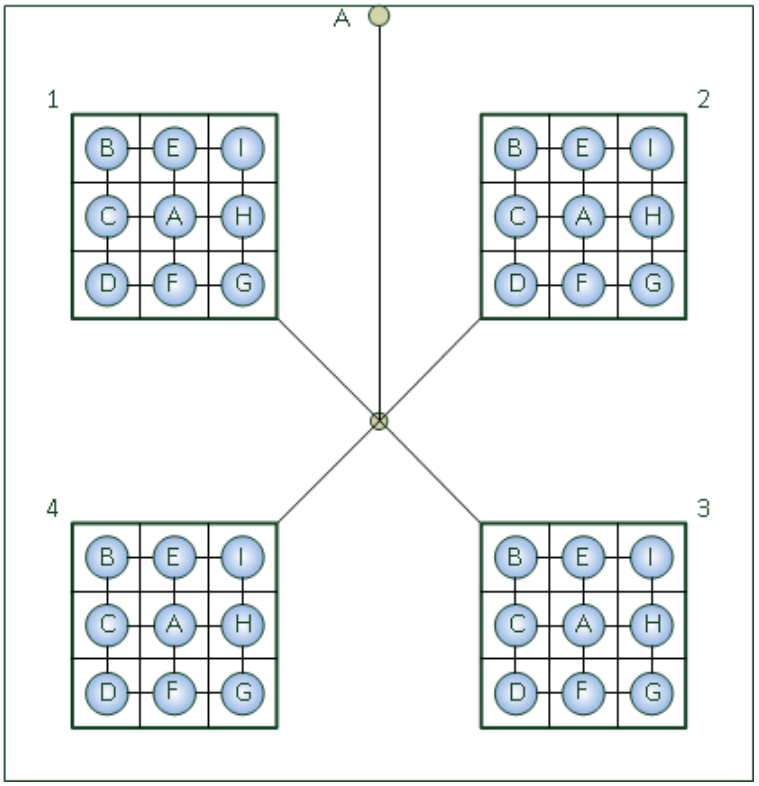



Figure 7.3.: SMBH custom board with 4 Spartan 3E FPGAs, Moore internal neighbourhood with Spidergon STNoC Architecture.

| 1 | A | B | C | D | E | F | G | H | I | Out |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 2 | •1 | 2 | •1 | •1 | 2 | •1 | 2 | •1 |
| B | 2 | 0 | •1 | 2 | •1 | 3 | 4 | 3 | 2 | •1 |
| C | •1 | •1 | 0 | •1 | 2 | 2 | 3 | 2 | 3 | •1 |
| D | 2 | 2 | •1 | 0 | 3 | •1 | 2 | 3 | 4 | •1 |
| E | •1 | •1 | 2 | 3 | 0 | 2 | 3 | 2 | •1 | •1 |
| F | •1 | 3 | 2 | •1 | 2 | 0 | •1 | 2 | 3 | •1 |
| G | 2 | 4 | 3 | 2 | 3 | •1 | 0 | •1 | 2 | •1 |
| H | •1 | 3 | 2 | 3 | 2 | 2 | •1 | 0 | •1 | •1 |
| I | 2 | 2 | 3 | 4 | •1 | 3 | 2 | •1 | 0 | •1 |
| Out | •1 | •1 | •1 | •1 | •1 | •1 | •1 | •1 | •1 | 0 |

Table 7.2.: Node to node router lookup table for FPGA 1 on Board A from figure 7.3, Out refers to routing beyond the local area network. The number is the hop cost between two nodes, those that are directly connected also have a bullet point next to the number.

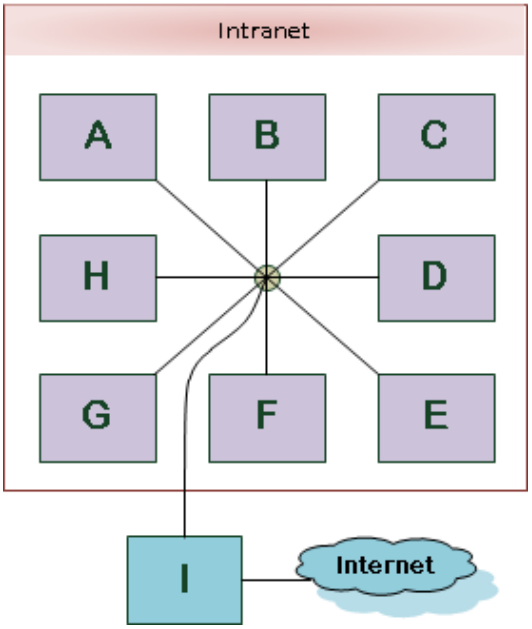



Figure 7.4.: 8 SMBH custom boards within local Intranet and external connection to Internet with Spidergon STNoC Architecture.

# A. JAVA Code.

## A.1. MyObjectGrid2D class extension

```
package sim.field.grid;
import sim.util.IntBag;
public class MyObjectGrid2D extends ObjectGrid2D {
   // Modifications to ObjectGrid2D class to prevent
   // disruption to the cell coordinate array.
private static final long serialVersionUID = 1L;
   public MyObjectGrid2D(int width, int height) {
      super( width, height);
      this.width = width;
      this.height = height;
      field = new Object[width][height];
   }
   protected void removeOrigin(int x, int y, IntBag xPos,
      IntBag  yPos)
   {
   int size = xPos.size();
   for(int i = 0; i <size; i++)
      {
      if (xPos.get(i) == x && yPos.get(i) == y)
         {
         xPos.removeNondestructively(i);
         yPos.removeNondestructively(i);
         return;
         }
      }
   }
// only removes the first occurrence
protected void removeOriginToroidal(int x, int y, IntBag xPos,
   IntBag yPos)
   {
   int size = xPos.size();
   int width = getWidth();
   int height = getHeight();
   x = tx(x, width, width*2, x+width, x-width);
   y = ty(y, height, height*2, y+height, y-height);
   for(int i = 0; i <size; i++)
   {
```

```
        if (tx(xPos.get(i), width, width*2, x+width, x-width) ==
           x &&  ty(yPos.get(i), height, height*2, y+height, y-
           height) == y)
           {
           xPos.removeNondestructively(i);
           yPos.removeNondestructively(i);
           return;
           }
        }
    }
}
```

## A.2. Code for Radial Neighbourhood

```
package sim.alg;

import sim.field.grid.*;
import sim.util.*;

public class RadialObjectTest {

    public static void bagOut(IntBag xs, IntBag ys)
    {
    if (xs.size() != ys.size()) System.err.println("WARNING,
       sizes not  same");
    for(int i = 0; i < xs.size(); i++)
       System.err.println("" + i + ": (" + xs.get(i) + ", " +
          ys.get(i) + ")");
    }
public static void main(String[] args)
    {
        {
           System.err.println("Bounded 5x5 Object Cell: (2,2) :
              Radius 1, ALL");
           ObjectGrid2D grid = new ObjectGrid2D(5, 5);
//         Bag result = new Bag();
           IntBag xs = new IntBag();
           IntBag ys = new IntBag();
           grid.getRadialLocations(2, 2, 1, Grid2D.BOUNDED, true,
               Grid2D.ALL, true, xs, ys);
           bagOut(xs,ys);
        }

        {
           System.err.println("Bounded 5x5 Object Cell: (2,2) :
              Radius 1, ANY");
```

```
            ObjectGrid2D grid = new ObjectGrid2D(5, 5);
            Bag result = new Bag();
            IntBag xs = new IntBag();
            IntBag ys = new IntBag();
            grid.getRadialNeighbors(2, 2, 1, Grid2D.BOUNDED, true,
                Grid2D.ANY, true, xs, ys);
            bagOut(xs,ys);
        }

        {
            System.err.println("Bounded 5x5 Object Cell: (2,2) :
               Radius 1, CENTER");
            ObjectGrid2D grid = new ObjectGrid2D(5, 5);
            Bag result = new Bag();
            IntBag xs = new IntBag();
            IntBag ys = new IntBag();
            grid.getRadialLocations(2, 2, 1, Grid2D.BOUNDED, true,
                Grid2D.CENTER, true, xs, ys);
            bagOut(xs,ys);
            }
    }

}
```

## A.3. Code for D0L Simple Model

```
package combug.first;

import javax.swing.*;
import java.awt.event.*;
import java.util.*;

public abstract class TimeCycle extends JFrame implements
   ActionListener {

  //private static Object celltype;

  public static void timeCycle() {
  // Open a window to get the time cycle length and convert from
      a string
  // to an integer.
     String tc = JOptionPane
          .showInputDialog("Enter number of growth cycles, must
             be < 38.");
     int tcycle = Integer.parseInt(tc);
```

```
// create an ArrayList with seed cell A.
 ArrayList<Character> cellType = new ArrayList<Character>();
 cellType.add('A');
 System.out.println("cellType: " + cellType);
 ArrayList<Character> newType = new ArrayList<Character>();

 while (tcycle > 0) {

   int procid = (cellType.size() - cellType.size());

   while (procid != cellType.size()) {
     char type = cellType.get(procid).charValue();

     switch (type) {
     case 'A':
       newType.add('C');
       newType.add('B');
       break;
     case 'B':
       newType.add('A');
       break;
     case 'C':
       newType.add('D');
       newType.add('A');
       break;
     case 'D':
       newType.add('C');
       break;
     default:
       System.out.println("Invalid cell type." + type);
       break;
       }
   procid++;
   }
   List<Character> temp = new ArrayList<Character>(cellType);
   cellType.clear();
   cellType.addAll(newType);
   newType.clear();
   // newType.addAll(temp);

   System.out.println("procid: " + procid);
   System.out.println("cellType: " + cellType);
   /*
    * When using this program do not input a number for growth
       cycles
    * greater than 37, as my computer ran out of resources.
    * For inputs greater than 8 one of the println line should
       be // out
    * dependent on which is more important.
```

```
      *
      */
    tcycle--;
    }
    }

    public static void main(String[] args) {
    // Schedule a job for the event-dispatching thread:
    // creating and showing this application's GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
        timeCycle();
        }
         }
      );
    }
}
```

# B. Moore and von Neumann heat-maps.

Heatmaps for both von Neumann and Moore for hops of 2, 4, and 8 are shown in 6.3, the heatmaps for a hop of 1 are shown below.

1. The Heatmap transitions from low values through the 50th percentile to the highest values with the following colours:
   a) Pale yellow is the lowest values.
   b) Pale blue is the 50th percentile.
   c) Dark blue is the highest values.
2. Each Heatmap consists of sixteen different simulations in three sets. Each set of simulations has one random average of one hundred simulation runs. There are also nine designated start points:
   - the four corners,
   - the four edge centres,
   - the central cell.

   a) von Neumann Heatmap for ObjectGrid2D hops 1, 2, 4 and 8.
   b) von Neumann Heatmap for extended ObjectGrid2D hops 1, 2, 4 and 8.
   c) Moore Heatmap for ObjectGrid2D hops 1, 2, 4 and 8.
   d) Moore Heatmap for extended ObjectGrid2D hops 1, 2, 4 and 8.

(a) Heatmap for default ObjectGrid2D, neighbours distance one cell.

| Alg 21 | vN | Neighbourhood | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 821 | 900 | 900 | 900 | 480 | 900 | 900 | 900 | 900 | 900 |
| | 45*20 | | 732 | 900 | 900 | 900 | 495 | 900 | 900 | 900 | 900 | 474 |
| | 60*15 | | 759 | 900 | 885 | 900 | 480 | 900 | 871 | 900 | 848 | 877 |
| | 90*10 | | 746 | 900 | 900 | 900 | 540 | 900 | 900 | 900 | 900 | 406 |
| | 150*6 | | 868 | 900 | 900 | 900 | 600 | 900 | 900 | 900 | 900 | 376 |
| 40*40 | | 1600 | 1239 | 1600 | 800 | 1600 | 840 | 1600 | 1600 | 1600 | 1600 | 821 |
| | 50*32 | | 802 | 1600 | 1600 | 1600 | 850 | 1600 | 1600 | 1600 | 1600 | 826 |
| | 80*20 | | 1315 | 1600 | 1600 | 1600 | 880 | 1600 | 1600 | 1600 | 1600 | 841 |
| | 100*16 | | 1169 | 1600 | 1600 | 1600 | 900 | 1600 | 1600 | 1600 | 1600 | 850 |
| | 160*10 | | 1339 | 1600 | 1600 | 1600 | 960 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 320*5 | | 1237 | 1600 | 1445 | 1600 | 960 | 1600 | 1441 | 1600 | 1600 | 801 |
| 50*50 | | 2500 | 2061 | 2500 | 2500 | 2500 | 1300 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 125*20 | | 2048 | 2500 | 2500 | 2500 | 1375 | 2500 | 2500 | 2500 | 2500 | 1314 |
| | 100*25 | | 2099 | 2500 | 2475 | 2500 | 1230 | 2500 | 2450 | 2500 | 2500 | 1250 |
| | 250*10 | | 2111 | 2500 | 2500 | 2500 | 1250 | 2500 | 2500 | 2500 | 2500 | 1126 |
| | 500*5 | | 1953 | 2500 | 2255 | 2500 | 1500 | 2500 | 2250 | 2500 | 2500 | 1250 |

(b) Heatmap for extended ObjectGrid2D, neighbours distance one cell.

| Alg 21m | vN | Neighbourhood | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 774 | 900 | 900 | 900 | 450 | 900 | 450 | 900 | 480 | 900 |
| | 45*20 | | 749 | 900 | 900 | 900 | 450 | 900 | 480 | 900 | 200 | 640 |
| | 60*15 | | 711 | 900 | 900 | 900 | 420 | 900 | 480 | 900 | 120 | 600 |
| | 90*10 | | 628 | 900 | 900 | 900 | 450 | 900 | 500 | 900 | 60 | 500 |
| | 150*6 | | 790 | 900 | 900 | 900 | 450 | 900 | 900 | 900 | 24 | 468 |
| 40*40 | | 1600 | 1576 | 1600 | 16000 | 1600 | 800 | 1600 | 880 | 1600 | 1600 | 1600 |
| | 50*32 | | 1146 | 1600 | 1600 | 1600 | 800 | 1600 | 1600 | 1600 | 1600 | 1600 |
| | 80*20 | | 1476 | 1600 | 1600 | 1600 | 800 | 1600 | 840 | 1600 | 1600 | 1040 |
| | 100*16 | | 1408 | 1600 | 1600 | 1600 | 800 | 1600 | 832 | 1600 | 1600 | 960 |
| | 160*10 | | 1267 | 1600 | 1600 | 1600 | 800 | 1600 | 820 | 1600 | 60 | 1600 |
| | 320*5 | | 1084 | 1600 | 820 | 1600 | 960 | 1600 | 810 | 1600 | 1600 | 820 |
| 50*50 | | 2500 | 2410 | 2500 | 2500 | 2500 | 1250 | 2500 | 2500 | 2500 | 1300 | 2500 |
| | 125*20 | | 2488 | 2500 | 2500 | 2500 | 1250 | 2500 | 2500 | 2500 | 2500 | 2500 |
| | 100*25 | | 2240 | 2500 | 1850 | 2500 | 1300 | 2500 | 1300 | 2500 | 2500 | 1600 |
| | 250*10 | | 2268 | 2500 | 2500 | 2500 | 1250 | 2500 | 2500 | 2500 | 2500 | 1300 |
| | 500*5 | | 1814 | 2500 | 1270 | 2500 | 1500 | 2500 | 1260 | 2500 | 2500 | 1270 |

Figure B.1.: von Neumann neighbourhood, Occupied cells for each grid.

Heatmap for a hop of one is shown with the default ObjectGrid2D in figure B.1a and the extended MyObjectGrid2D in figure B.1b. In comparison with each other figure B.1a has more starting positions that are fully occupied than figure B.1b. Thus ObjectGrid2D is the better choice here.

(a) Heatmap for default ObjectGrid2D, neighbours distance one cell.

| Alg 6-1 | Moore | Neighbourhood | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 77 | 59 | 74 | 88 | 88 | 87 | 87 | 88 | 73 | 73 |
| | 45*20 | | 66 | 39 | 61 | 83 | 83 | 83 | 61 | 58 | 48 | 61 |
| | 60*15 | | 47 | 29 | 59 | 88 | 88 | 88 | 59 | 43 | 35 | 59 |
| | 90*10 | | 63 | 19 | 19 | 64 | 108 | 108 | 64 | 28 | 23 | 64 |
| | 150*6 | | 108 | 11 | 86 | 160 | 160 | 160 | 86 | 16 | 896 | 86 |
| 40*40 | | 1600 | 108 | 79 | 99 | 118 | 118 | 117 | 117 | 118 | 98 | 98 |
| | 50*32 | | 90 | 63 | 88 | 112 | 112 | 112 | 93 | 94 | 78 | 88 |
| | 80*20 | | 94 | 39 | 79 | 118 | 118 | 118 | 79 | 58 | 48 | 79 |
| | 100*16 | | 72 | 31 | 81 | 130 | 130 | 130 | 81 | 46 | 38 | 81 |
| | 160*10 | | 95 | 19 | 99 | 178 | 178 | 178 | 99 | 28 | 23 | 99 |
| | 320*5 | | 184 | 9 | 167 | 328 | 328 | 328 | 169 | 13 | 1598 | 169 |
| 50*50 | | 2500 | 130 | 99 | 124 | 148 | 148 | 147 | 147 | 148 | 123 | 123 |
| | 125*20 | | 98 | 39 | 101 | 163 | 163 | 163 | 101 | 58 | 48 | 101 |
| | 100*25 | | 90 | 49 | 99 | 148 | 148 | 148 | 99 | 73 | 60 | 99 |
| | 250*10 | | 151 | 19 | 144 | 268 | 268 | 268 | 144 | 28 | 23 | 144 |
| | 500*5 | | 282 | 9 | 259 | 508 | 508 | 508 | 259 | 13 | 2498 | 259 |

(b) Heatmap for extended MyObjectGrid2D, neighbours distance one cell.

| Alg 61m | Moore | Neighbourhood | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Square X*Y | Rectangle X*Y | Max Cells | Cells Occupied Random | Min x, Min y | Mid x, Min y | Max x, Min y | Max x, Mid y | Max x, Max y | Mid x, Max y | Min x, Max y | Min x, Mid y | Mid x, Mid y |
| 30*30 | | 900 | 899 | 899 | 899 | 899 | 899 | 900 | 900 | 900 | 899 | 899 |
| | 45*20 | | 891 | 900 | 900 | 900 | 889 | 900 | 899 | 899 | 889 | 889 |
| | 60*15 | | 899 | 899 | 899 | 899 | 899 | 900 | 899 | 900 | 899 | 899 |
| | 90*10 | | 899 | 899 | 899 | 899 | 899 | 1600 | 900 | 900 | 899 | 899 |
| | 150*6 | | 900 | 899 | 899 | 899 | 899 | 900 | 900 | 900 | 899 | 899 |
| 40*40 | | 1600 | 1599 | 1599 | 1599 | 1599 | 1599 | 1600 | 2499 | 1600 | 1599 | 1599 |
| | 50*32 | | 1599 | 1599 | 1599 | 1599 | 1599 | 1600 | 1600 | 1600 | 1599 | 1599 |
| | 80*20 | | 1599 | 1599 | 1599 | 1599 | 1599 | 1600 | 1599 | 1600 | 1599 | 1599 |
| | 100*16 | | 1599 | 1599 | 1599 | 1599 | 1599 | 1600 | 1599 | 1600 | 1599 | 1599 |
| | 160*10 | | 1599 | 1599 | 1599 | 1599 | 1599 | 1600 | 1599 | 1600 | 1599 | 1599 |
| | 320*5 | | 1599 | 1599 | 1599 | 1599 | 1600 | 1600 | 1599 | 1600 | 1600 | 1600 |
| 50*50 | | 2500 | 2499 | 2499 | 2499 | 2499 | 2499 | 2500 | 2500 | 2500 | 2499 | 2499 |
| | 125*20 | | 2491 | 25000 | 2500 | 2500 | 2500 | 2500 | 2499 | 2499 | 2489 | 2489 |
| | 100*25 | | 2499 | 2499 | 2499 | 2499 | 2499 | 2500 | 2499 | 2500 | 2499 | 2499 |
| | 250*10 | | 2499 | 2499 | 2499 | 2499 | 2499 | 2500 | 2500 | 2500 | 2499 | 2499 |
| | 500*5 | | 2499 | 2499 | 2499 | 2499 | 25000 | 2500 | 2499 | 2500 | 2500 | 2500 |

Figure B.2.: Moore neighbourhood, Occupied cells for each grid.

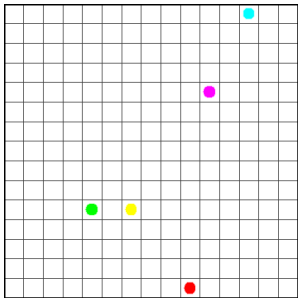
Heatmaps in figure B.2 for a hop of one is shown with the default ObjectGrid2D in figure B.2a and the extended MyObjectGrid2D in figure B.2b. In comparison with each other figure B.2b has nearly all starting positions nearly fully occupied whilst figure B.2a has very variable and low occupancy. Thus MyObjectGrid2D is the better choice here.

# C. Other figures.

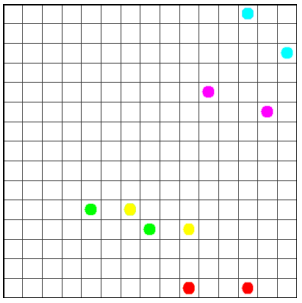
## C.1. Radial Neighbourhood step by step

The figures below are for the radial simulation discussed in 6.4.1.

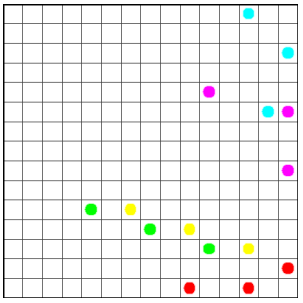
Figure C.1.: Simulation of a radial neighbourhood with a hop of four. Starting with five agents and increasing by a further five at each step. Random seed of 3 and limited cell occupation shown at step twenty. The cell at coordinates $(10, 0)$, coloured cyan with a black border is the one that is blocked from further movement and stops the whole simulation.



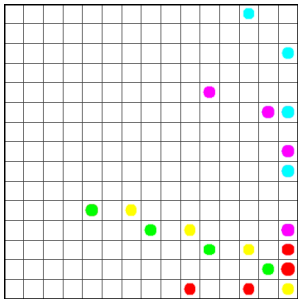
(a) Step one.



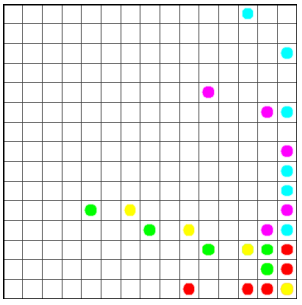
(b) Step two.



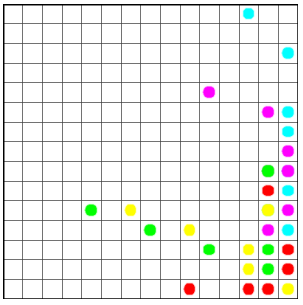
(c) Step three.



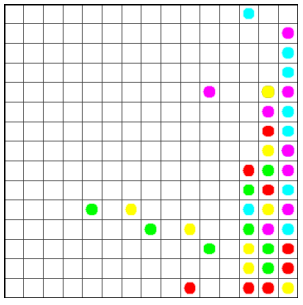
(d) Step four.



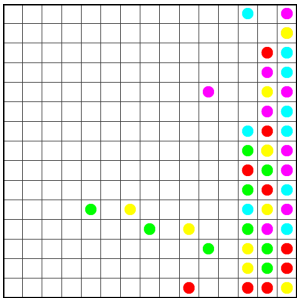
(e) Step five.



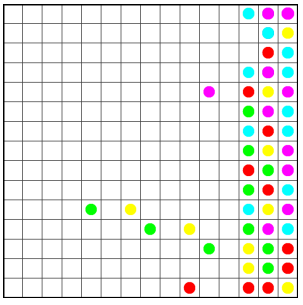
(f) Step six.



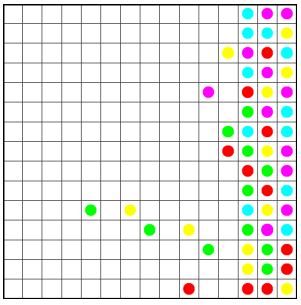
(g) Step seven.



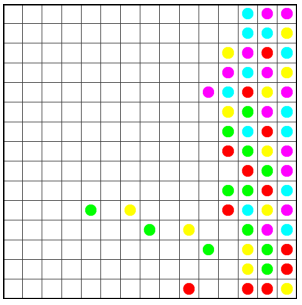
(h) Step eight.



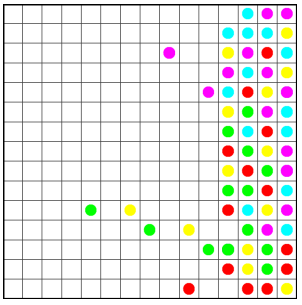
(i) Step nine.



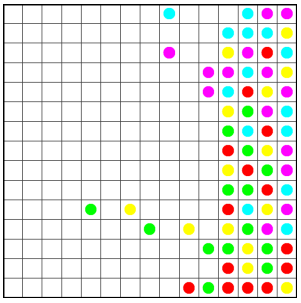
(j) Step ten.



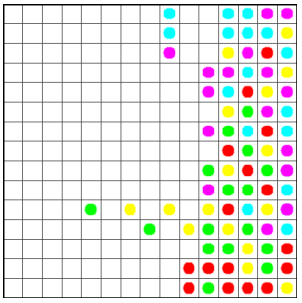
(k) Step eleven.



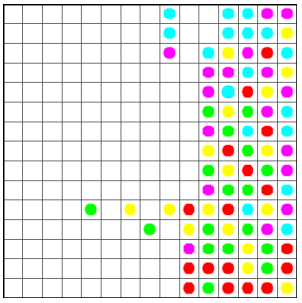
(l) Step twelve.



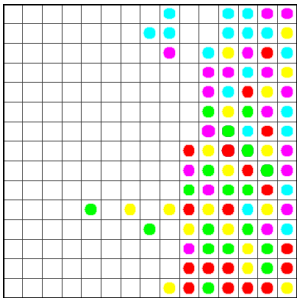
(m) Step thirteen.



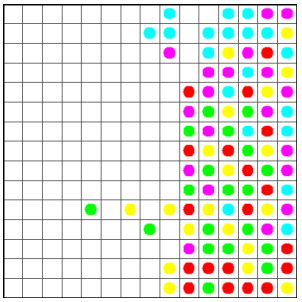
(n) Step fourteen.



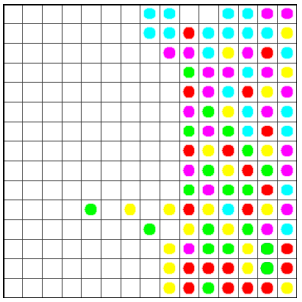
(o) Step fifteen.



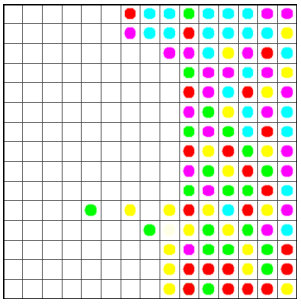
(p) Step sixteen.



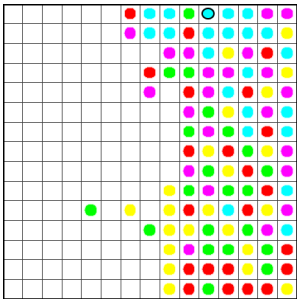
(q) Step seventeen.



(r) Step eighteen.



(s) Step nineteen.



(t) Step twenty.

# D. Google drive for media.

## D.1. Google Drive link for Simulation films:

https://drive.google.com/drive/folders/13K_vl0gLHzQ9m5d_kAnEWO6lbdqJHq5m?usp=sharing

## D.2. Simulation film listing:

1. Films of simulation runs

    a) Experiment 1.

        - ZigZag0.wmv
        - ZigZag1a.wmv

    b) Experiment 2.

        - M1_2_4e.wmv
        - R_all_H4_A5.wmv
        - vN1_2_4e.wmv

    c) Experiment 3.

        - M_H2_8_A3_10.wmv
        - R_all_H4_A5.wmv
        - vN_H2_A3.wmv
        - ST_H1_A10.wmv

    d) Experiment 4.

- M_H4_A50.wmv
- vN_H2_A50.wmv
- vN_H1_A5.wmv

# E. Cohens'*d* calculation

## E.1. Statistical analysis of Radial local neighbourhood core occupancy

Statistically p-values are often used to determine if there is a significant difference between two groups. However, it does not tell the size of the impact. To understand this, we need to know the *effect size*. Effect size can tell how large this difference actually is. In practice, effect sizes are much more interesting and useful to know than p-values.

To calculate the effect size through a standardised mean difference using Cohen's *d*, which is calculated as:

$$\text{Cohen}'\text{s } d = (\overline{x}_1 - \overline{x}_2) \,/\, s$$

where $\overline{x}_1$ and $\overline{x}_2$ are the sample means of group even and group odd, respectively, and s is the standard deviation of the population from which the two groups were taken. The effect size with a *d* of 0.2 or smaller is considered to be small effect size, a *d* of around 0.5 is considered to be a medium effect size and a *d* of 0.8 or larger is considered to be large effect size. The larger the effect size, the larger the difference between the average core in each group. If the means of two groups don't differ by at least 0.2 standard deviations the difference is trivial, even if the p-value is statistically significant.

Table E.1 shows the two samples data and the calculation and result for *d* which in this case is 0.597596. This is a medium effect size difference. Figure E.1 shows two boxplots, one for even and one for odd, where it can be seen that even has the bulk of high core occupancy within a narrow range, whereas odd core occupancy is much more

wider spread.

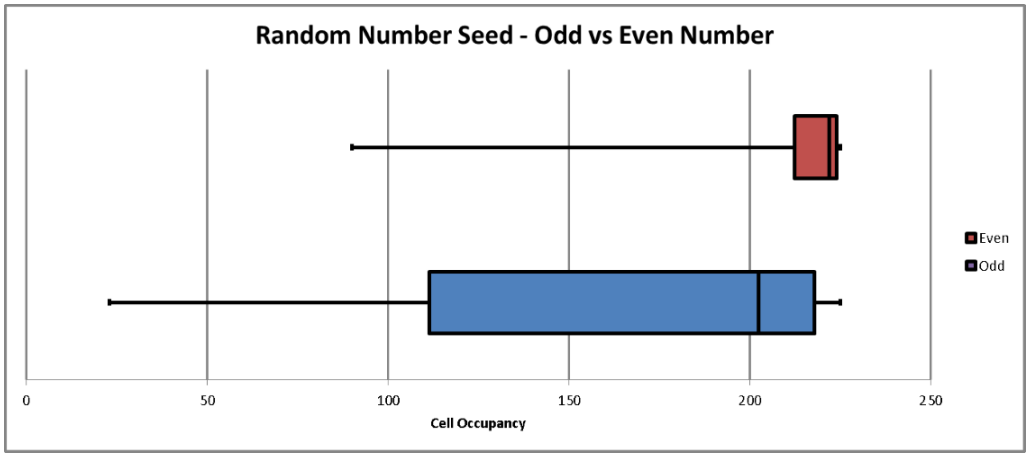



Figure E.1.: Boxplot of cell occupancy for odd random number seed versus even random number seed. Even random start positions lead to higher core occupancy than odd random start positions.

Table E.1.: Random number generator seed, Odd and Even comparison

| odd | even |
|---|---|
| 23 | 222 |
| 116 | 225 |
| 200 | 220 |
| 105 | 222 |
| 95 | 225 |
| 219 | 222 |
| 224 | 224 |
| 224 | 224 |
| 110 | 120 |
| 120 | 223 |
| 85 | 223 |
| 195 | 90 |
| 215 | 225 |
| 210 | 200 |
| 205 | 220 |
| 215 | 90 |
| 219 | 210 |
| 225 | 224 |

| | Mean | SD | n |
|---|---|---|---|
| Odd | 166.9444 | 63.98113 | 18 |
| Even | 200.5 | 47.03472 | 18 |
| | | | |
| M1-M2 | 33.55556 | | |
| Pooled SD | 56.15091 | | |
| Cohen's d | 0.597596 | | |

# F. List of Acronyms

## F.1. Acronyms

**SoC** System on Chip

**NoC** Network on Chip

**FPGA** Field Programmable Gate Array

**2D** two dimensional

**SISD** Single Instruction Single Data

**SIMD** Single Instruction Multiple Data

**MISD** Multiple Instruction Single Data

**MIMD** Multiple Instruction Multiple Data

**CPU** Central processing Unit

**MPSoC** Multi-Processor Systems

**RISC** Reduced Instruction Set Computer

**IP** Intellectual Property

**DSP** Digital Signal Processing

**TTA** Transport Triggered Architecture

**BMM** Bit manipulating Machines

**VLIW** Very Long Instruction Word

**FU** Functional Units

**VEHW** Virtual Evolvable Hardware

**OSI** Open Systems Interconnection

**LAN** Local Area Network

**WAN** Wide Area Network

**CA** Cellular Automata

**BRAM** Block RAM

### F.1.1. Acronyms that cannot be linked

Due to a problematic interaction between the Acronym package and the Koma-Script document layout some acronyms are warned as unreferenced and fail to link. These are the acronyms that this happened to.

**SPMD** Single Program Multiple Data shown in chapter 2.1, Basic architectures.

**IN** Interconnection Network shown in chapter 2.1, Custom processor.

**GUI** Graphical User Interface shown in chapter 2.2, Simulation tools.

**IDE** Integrated Development Environment shown in chapter 2.2, Simulation tools.

**CSMA/CD** Carrier Sense Multiple Access / Carrier Detection, shown in chapter 7.2, Future Work.

# Bibliography


[1] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13–33, 2017.

[2] Susumu Adachi, Ferdinand Peper, Jia Lee, and Hiroshi Umeo. Occurrence of gliders in an infinite class of life-like cellular automata. In *8th international conference on Cellular Automata for Research and Industry*, ACRI '08, pages 32–41, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Farhan Ahammed and Pablo Moscato. Evolving L-systems as an intelligent design approach to find classes of difficult-to-solve traveling salesman problem instances. In *2011 international conference on Applications of evolutionary computation - Volume Part I*, EvoApplications'11, pages 1–11, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] P. Anghelescu, E. Sofron, C. I. Rincu, and V. G. Iana. Programmable cellular automata based encryption algorithm. *Cas: 2008 International Semiconductor Conference, game-of-life environment*, 1:351–354, 2008.

[5] J. L. Beuchat and J. O. Haenni. Von neumann's 29-state cellular automaton: A hardware implementation. *IEEE Transactions on Education*, 43(3):300–308, 2000.

[6] William A. Beyer, Peter H. Sellers, and Michael S. Waterman. Stanislaw M. Ulam's Contributions to Theoretical Theory. *Letters in Mathematical Physics*, 10 1985.

[7] François Blanchard, Petr Kurka, and Alejandro Maass. Topological and measure-theoretic properties of one-dimensional cellular automata. *Physica D: Nonlinear Phenomena*, 103(1-4):86 – 99, 1997.

[8] Colin Andrew Bonney. *Fault Tolerant Task Mapping in Many-Core Systems*. Phd thesis, University of York, Electronic Engineering, May 2016.

[9] Arthur W. Burks. *Essays on cellular automata*. University of Illinois Press, Urbana, 1970.

[10] John Byl. Self-reproduction in small cellular automata. *Physica D: Nonlinear Phenomena*, 34(1 - 2):295 – 299, 1989.

[11] E.F. Codd. *Cellular Automata*. Academic Press Inc., 1968.

[12] M. Cook. Universality in elementary cellular automata,. *Complex Systems*, 15:1–40, 2004.

[13] M. Cook. A concrete view of rule 110 computation. *EPTCS*, 1:31–55, 2008.

[14] Henk Corporaal. *Microprocessor Architectures from VLIW to TTA*. John Wiley, 1998.

[15] P. Corsonello, G. Spezzano, G. Staino, and D. Talia. Efficient implementation of cellular algorithms on reconfigurable hardware. In *10th Euromicro conference on Parallel, distributed and network-based processing*, EUROMICRO-PDP'02, pages 211–218, Washington, DC, USA, 2002. IEEE Computer Society.

[16] Frederica Darema. The SPMD model: Past, present and future. In Yiannis Cotronis and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 1–1, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[17] Pedro P.B. de Oliveira, José C. Bortot, and Gina M.B. Oliveira. The best currently known class of dynamically equivalent cellular automata rules for density classification. *Neurocomputing*, 70(1-3):35–43, 2006.

[18] Nachum Dershowitz and Laurent Vigneron. Rewriting home page. `http://rewriting.loria.fr/`, December 2011.

[19] R Doursat. The growing canvas of biological development: multiscale pattern generation on an expanding lattice of gene regulatory networks. *InterJournal Complex Systems*, 1809, 2006.

[20] R Doursat. Programmable architectures that are complex and self-organized: From morphogenesis to engineering. In *11th International Conference on the Simulation and Synthesis of Living Systems*, 2008.

[21] R Doursat. Morphogenetic engineering weds bio self-organization to human-designed systems. *PerAda Magazine*, 2011.

[22] Rene Doursat. *Organic Computing: Chapter 8*. Springer-Verlag, 2008. Chapter 8: Organically grown architectures: Creating decentralized, autonomous systems by embryomorphic engineering.

[23] René Doursat and Mihaela Ulieru. Emergent engineering for the management of complex situations. In *2nd International Conference on Autonomic Computing and Communication Systems*, Autonomics '08, pages 14:1–14:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[24] Rene Doursat and Mihaela Ulieru. Guiding the emergence of structured network topologies: A programmed attachment approach. In *Dynamics On and Of Complex Networks Workshop*. 5th European Conference on Complex Systems (ECCS 2008), 2008.

[25] G. M. Du, D. L. Zhang, Y. S. Yin, L. Ma, L. F. Geng, Y. K. Song, and M. L. Gao. Fpga prototype design of network on chips. *2008 2nd International Conference on Anti-Counterfeiting, Security and Identification*, 1:348–351, 2008.

[26] Criteria for Conway's rules. Criteria for conway's rules. `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`, July 2012.

[27] N. Fujita. A system design using FPGA for large scale computing. *Wmsci 2008: 12th World Multi-Conference on Systemics, Cybernetics and Informatics, Vol Iii*, Vol Iii:144–147, 2008.

[28] Martin Gardner. Mathematical games the fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[29] Allan Gottlieb George S. Almasi. *Highly Parallel Computing*. Benjamin/Cummings Pub. Co., 1994.

[30] J. A. Gomez-Pulido, J. M. Matas-Santiago, F. Perez-Rodriguez, M. A. Vega-Rodriguez, J. M. Sanchez-Perez, and F. F. de Vega. Hardware modelling of cellular automata: The game of life case. *Computer Aided Systems Theory- Eurocast 2007*, 4739:589–595, 2007.

[31] MicroBlaze Processor Reference Guide. *MicroBlaze Processor Reference Guide Embedded Development Kit EDK 14.1 UG081 (v14.1)*. Xilinx.Inc, xilinx edk 14.1 release edition, apr 2012.

[32] J. I. Hidalgo, F. Fernandez, J. Lanchares, J. M. Sanchez, R. Hermida, M. Tomassini, R. Baraglia, R. Perego, and O. Garnica. Multi-FPGA systems synthesis by means of evolutionary computation. *Genetic and Evolutionary Computation - Gecco 2003, Pt Ii*, 2724:2109–2120, 2003.

[33] Derek F. Holt, Sarah Rees, and Claas E. Röver. *Groups, Languages and Automata, Rewriting Systems*, chapter 4, pages 117–124. London Mathematical Society Student Texts. Cambridge University Press, 2017.

[34] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Science/Engineering/Math, 1992.

[35] Ahmed Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, 2004.

[36] Jarkko Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334:3 – 33, 2005.

[37] Ryan C. Kennedy, Glen E. P. Ropella, and C. Anthony Hunt. Implementing a cell-centered, agent-based framework with flexible environment granularities using mason and vtk. *ADS '16: Proceedings of the Agent-Directed Simulation Symposium*, 2016.

[38] F. Khalili and H. R. Zarandi. A fault-tolerant core mapping technique in networks-on-chip. In *IET Computers & Digital Techniques*, volume 7.6, 2013.

[39] Nikolaos Kyparissas and Apostolos Dollas. An FPGA-based architecture to simulate cellular automata with large neighborhoods in real time. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 95–99, Sep. 2019.

[40] Nikolaos Kyparissas and Apostolos Dollas. Large-scale cellular automata on FPGAs: A new generic architecture and a framework. *ACM Transactions on Reconfigurable Technology and System*, 14(1), 12 2020. Article 5.

[41] C. G. Langton. Self-reproduction in cellular automata. *Physica D*, 10(1-2):135–144, 1984.

[42] Y. F. Liu, P. Liu, Y. T. Jiang, M. Yang, K. J. Wu, W. D. Wang, and Q. D. Yao. Building a multi-FPGA-based emulation framework to support networks-on-chip design and verification. *International Journal of Electronics*, 97(10):1241–1262, 2010.

[43] S. Luke et al. *The MASON Simulation Toolkit: Past, Present, and Future.*., volume 11463 of . *Lecture Notes in Computer Science*. Springer, Cham., (2019).

[44] Sean Luke. *The ECJ Owner's Manual*. Department of Computer Science George Mason University, 27 edition, August 2019.

[45] Sean Luke. *Multiagent Simulation and the MASON Library*. George mason University, Department of Computer Science, 20 edition, August 2019.

[46] J. C. Lyke, G. W. Donohoe, and S. P. Karna. Cellular automata-based reconfigurable systems as transitional approach to gigascale electronic architectures. *Journal of Spacecraft and Rockets*, 39(4):489–494, 2002.

[47] P. Mal and F. R. Beyette. Development of an FPGA for multi-technology applications. *2002 45th Midwest Symposium on Circuits and Systems, Vol Iii*, Vol Iii:172–175, 2002.

[48] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Toward self-repairing and self-replicating hardware: The embryonics approach. In *Evolvable Hardware, 2000. The Second NASA/DoD Workshop*, Second Nasa/Dod Workshop on Evolvable Hardware, Proceedings, 2000.

[49] D. Mange, A. Stauffer, E. Petraglio, and G. Tempesti. Self-replicating loop with universal construction. *Physica D-Nonlinear Phenomena*, 191(1-2):178–192, 2004.

[50] P. Marchal. John von Neumann: The founding father of artificial life. *Artificial Life*, 4(3):229–235, 1998.

[51] N. Margolus. An FPGA architecture for dram-based systolic computations. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 2–11, April 1997.

[52] Marc Martin, Bastien Chopard, and Paul Albuquerque. Formation of an ant cemetery: swarm intelligence or statistical accident? *Future Gener. Comput. Syst.*, 18:951–959, August 2002.

[53] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3 to 30, January 1998.

[54] B. H. Mayoh. Multidimensional Lindenmayer organisms. In :Rozenberg, G., Salomaa, A. (eds) L Systems. *Lecture Notes in Computer Science, vol 15.*, pages 302–326, London, UK, 1974. Springer.

[55] Oleg Mazonka and Alex Kolodin. A simple multi-processor computer based on subleq, May 2011.

[56] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. Randomly failed! the state of randomness in current Java implementations. In Ed Dawson, editor, *Topics in Cryptology – CT-RSA 2013*, pages 129–144, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[57] Pierre-Andre Mudry. *A hardware-software codesign framework for cellular computing*. PhD thesis, Ecole Polytechnique Federal De Lausanne, 2009.

[58] S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot. Cellular automata simulations on a fpga cluster. *The International Journal of High Performance Computing Applications*, 25(2), 2011.

[59] Gabriela Ochoa. On genetic algorithms and lindenmayer systems. In *PARALLEL PROBLEM SOLVING FROM NATURE V*, pages 335–344. Springer-Verlag, 1998.

[60] Cesar Ortega and Andy Tyrrell. A hardware implementation of an embryonic architecture using virtex® fpgas. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pages 155–164, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[61] N. H. Packard and S. Wolfram. Two-dimensional cellular automata. *Journal of Statistical Physics*, 38(5-6):901–946, 1985.

[62] J. Y. Perrier, M. Sipper, and J. Zahnd. Toward a viable, self-reproducing universal computer. *Physica D*, 97(4):335–352, 1996.

[63] U Pesavento. An implementation of von neumann's self-reproducing machine. *Artificial Life*, 2(4):337–354, 1995.

[64] Lucian Prodan, Gianluca Tempesti, Daniel Mange, and André Stauffer. Embryonics: Artificial cells driven by artificial DNA. In Yong Liu, Kiyoshi Tanaka, Masaya Iwata, Tetsuya Higuchi, and Moritoshi Yasunaga, editors, *Evolvable Systems: From Biology to Hardware*, volume 2210 of *Lecture Notes in Computer Science*, pages 100–111. Springer Berlin / Heidelberg, 2001.

[65] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Virtual laboratory. Springer-Verlag, New York, 1990.

[66] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, 82(9):609–623, 2006.

[67] B. Naresh Kumar Reddy, M. H. Vasantha, and Y.B. Nithin Kumar. A gracefully degrading and energy-efficient fault tolerant noc using spare core. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 146–151, July 2016.

[68] J.A. Reggia, S.L. Armentrout, H. Chou, and Y. Peng. Simple systems exhibiting self-directed replication. *Science*, 259:1282–1286, May 1993.

[69] R. Santoro, S. Roy, and O. Sentieys. Search for optimal five-neighbor FPGA-based cellular automata random number generators. *2007 International Symposium on Signals, Systems and Electronics,*, Vols 1 and 2:332–335, 2007.

[70] J. Signorini. How a SIMD machine can implement a complex cellular automata? a case study: von Neumann's 29-state cellular automaton. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 175–186, New York, NY, USA, 1989. ACM.

[71] M. Sipper. Fifty years of research on self-replication: An overview. *Artificial Life*, 4(3):237–257, 1998.

[72] M. Sipper and M. Tomassini. An introduction to cellular automata. In D. Mange and M. Tomassini, editors, *Bio-inspired Computing Machines: Towards Novel Computational Architectures*, chapter 3, pages 49 – 98. Presses Polytechniques et Universitaires Romandes., Lausanne, Switzerland, 1998.

[73] Moshe Sipper. Simple parallel local cellular computing. In Agoston Eiben, Thomas BÃck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 653–662. Springer Berlin / Heidelberg, 1998.

[74] A. Stauffer and M. Sipper. L-hardware: Modeling and implementing cellular development using L-systems. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Towards Novel Computational Architechtures.*, chapter 10, pages 269 – 286. Presses Polytechniques et Universitaires Romandes., Lausanne, Switzerland., 1998.

[75] A. Stauffer and M. Sipper. Modeling cellular development using l-systems. *Evolvable Systems: From Biology to Hardware*, 1478:196–205, 1998.

[76] A. Stauffer and M. Sipper. On the relationship between cellular automats and l-systems: The self-replication case. *Physica D-Nonlinear Phenomena*, 116(1-2):71–80, 1998.

[77] A. Stauffer and M. Sipper. Emergence of self-replicating loops in an interactive, hardware-implemented game-of-life environment. In S. Chopard B. Tomassini M. Bandini, editor, *"ACRI'02",*, volume 2493 of *Lecture Notes in Computer Science*, pages 123–131, 2002.

[78] Konstantinos Tatas, Kostas Siozios, Dimitrios Soudris, and Axel Jantsch. *Designing 2D and 3D Network-on-Chip Architectures*. Springer, 2014.

[79] Gianluca Tempesti. A new self-reproducing cellular automaton capable of construction and computation. In *in ECAL95: Proceedings of the Third European Conference on Artificial Life*, pages 555–563. Springer-Verlag, 1995.

[80] C. Teuscher, D. Mange, A. Stauffer, and G. Tempesti. Bio-inspired computing tissues: towards machines that evolve, grow, and learn. *Biosystems*, 68(2-3):235–244, 2003.

[81] Gunnar Tufte and Pauline Haddow. Building knowledge into developmental rules for circuit design. In Andy Tyrrell, Pauline Haddow, and Jim Torresen, editors, *Evolvable Systems: From Biology to Hardware*, volume 2606 of *Lecture Notes in Computer Science*, pages 69–80. Springer Berlin / Heidelberg, 2003.

[82] Bala Dhandayuthapani Veerasamy. Concurrent approach to flynn's mpmd classification through java. *IJCSNS*, 10(2):164, 2010.

[83] Eric Weisstein. Still life - from Eric Weisstein's Treasure Trove of Life C.A. URL: http://www.ericweisstein.com/encyclopedias/life/StillLife.html, July 2012.

[84] S. Wolfram. Statistical-mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, 1983.

[85] S. Wolfram. Cellular automata as models of complexity. *Nature*, 311(5985):419–424, 1984.

[86] S. Wolfram. Computation theory of cellular automata. *Communications in Mathematical Physics*, 96(1):15–57, 1984.

[87] S. Wolfram. Universality and complexity in cellular automata. *Physica D*, 10(1-2):1–&, 1984.

[88] Stephen Wolfram. *Cellular automata and complexity : collected papers*. Westview Press, [S.l.], 1994.