INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

GRADUATE PROGRAM ON ELECTRONICS, COMPUTER SCIENCE, INFORMATICS AND COMMUNICATIONS

A RECONFIGURABLE COMPUTING ARCHITECTURE BASED ON CELLULAR AUTOMATA

THESIS

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN ELECTRONICS ENGINEERING

POR

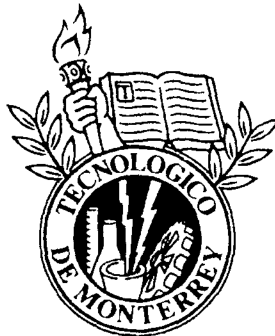
IOS ALBERTO CRUZ GUZMAN

Monterrey, N. L., Dec. 2003

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

GRADUATE PROGRAM ON ELECTRONICS, COMPUTER SCIENCE, INFORMATICS AND COMMUNICATIONS



A RECONFIGURABLE COMPUTING ARCHITECTURE BASED ON CELLULAR AUTOMATA

THESIS

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRONICS ENGINEERING

IOS ALBERTO CRUZ GUZMÁN

Monterrey, N.L., December 2003

A RECONFIGURABLE COMPUTING ARCHITECTURE BASED ON CELLULAR AUTOMATA

THESIS

IOS ALBERTO CRUZ GUZMÁN

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRONICS ENGINEERING

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

DECEMBER 2003

A mi mamá con amor y admiración.

Acknowledgements

I want to thank the ITESM and CONACyT for granting me an Excellence Scholarship for graduate studies.

To my advisor, Alfonso Avila, Ph.D., who oriented me and was a great support.

To the committee members, David Garza, Ph.D., and Manuel Valenzuela, Ph.D., for their time and reflections.

Abstract

This thesis proposes a reconfigurable computer architecture based on cellular automata capable of improving computing performance by exploiting the massive parallelism from its individual's interaction. Studies, in the last 15 years, proved that the complexity of the synthesis tools needed to exploit this parallelism increased as the architecture granularity was finer. An architecture based on cellular automata represents the finest granularity. Studies have proved that cellular automata granularity can be handle or programmed using genetic algorithms. It is time for a viable reconfigurable computing architecture based on cellular computing to be proposed.

The architecture is oriented to exploit nature's parallelism while using the semiconductor technology available nowadays. The philosophy behind is to make the hardware as simple as it may be, and make the software as complex as it is required to be in order to perform valuable computations. An evolutive approach is used to handle the software complexities. A road to make this computer architecture feasible is suggested. The first steps towards the implementation of useful cellular automaton computer architecture were explored, including the physical media selection, topology definition, basic programming tools development, and search for a cell's rule computationally efficient and universal. Examples on how this architecture can compute simple Boolean functions are presented.

- *Keywords:* computer architecture, reconfigurable computing, cellular automata, cellular computing, and parallel computing.

Contents

1 Introduction **1**

1.1 Traditional Computers . . . 2
1.2 Reconfigurable Computers . . . 2
1.3 Cellular Computers . . . 2
1.4 Living Computers . . . 3
1.5 Problem Description . . . 4
1.6 Objective and Contributions . . . 4
1.7 Development Stages . . . 4
1.8 Document Organization . . . 5

2 Background **6**

2.1 Cellular Automata . . . 6
2.2 Reconfigurable Computing . . . 8
2.3 Genetic Algorithms . . . 9
2.4 Related Work . . . 13
2.4.1 The Plastic Cell . . . 13
2.4.2 Universal Cellular Automata . . . 14
2.4.3 Cellular Automata Machines . . . 14
2.4.4 VLSI Design using Genetic Algorithms . . . 14
2.4.5 Evolving Cellular Automata with Genetic Algorithms . . . 14

3 Architecture **16**

3.1 Input/Output . . . 16
3.2 Programming and Execution . . . 16
3.3 Physical Media . . . 17
3.4 Cellular Automata Topology . . . 18
3.5 Cell Behavior . . . 19
3.5.1 Local Behaviors . . . 19
3.5.2 Global Behaviors . . . 25
3.6 Implementation . . . 25
3.6.1 Synchronous . . . 25
3.6.2 Asynchronous . . . 26
3.7 Machine Language . . . 26
3.8 Programming and Execution Example . . . 26

4 Compiler **36**
4.1 A Compiler based on Genetic Algorithms 36
4.2 Genetic Algorithm 37
4.3 Genotype Coding 37
4.4 Genetic Operators 37
4.5 Evaluating Function 38
4.6 Compiling Procedure 41

5 Methodology and Results **43**
5.1 Research Platform 43
5.2 Rule Search 43
5.2.1 Rotation Invariance Selection Criterion 44
5.2.2 Local Behaviors Selection Criterion 45
5.2.3 Global Behaviors Selection Criterion 47
5.3 Rule 48336 52
5.4 Program Verification 54
5.5 Simple Programs 54
5.6 Results Validity 57
5.7 Applications 57

6 Conclusions and Future Work **58**

List of Figures

1.1 Cellular Computing. 1

2.1 Frequently used 2D neighborhoods. 7
2.2 One-dimensional cellular automata transition function example. 7
2.3 First time steps of the one-dimension cellular automaton example. 7
2.4 Result on the one-dimension cellular automaton after 100 step. 8
2.5 Hardware implemented algorithm vs Software implemented algorithm 8
2.6 Typical FPGA topology. 8
2.7 Simple genetic algorithm flowchart. 10
2.8 One-point crossover. 12
2.9 Two-point crossover. 12
2.10 Uniform crossover. 12

3.1 Cellular automata input/output interface boundary. 16
3.2 Cellular automata programming and execution cycles. 17
3.3 Cell and cellular automata graphic representation convention. 19
3.4 Rule 48336 state transition table. 27
3.5 Cellular automaton initial configuration. 28
3.6 Cellular automaton programming example. 29
3.7 Cellular automaton execution example $o_5 = \overline{i_{18} i_{14}} = \overline{00} = 1$. 31
3.8 Cellular automaton execution example $o_5 = \overline{i_{18} i_{14}} = \overline{01} = 1$. 32
3.9 Cellular automaton execution example $o_5 = \overline{i_{18} i_{14}} = \overline{10} = 1$. 33
3.10 Cellular automaton execution example $o_5 = \overline{i_{18} i_{14}} = \overline{11} = 0$. 34
3.11 Cellular automaton program unloading example. 35

4.1 Genetic algorithm. 37
4.2 Crossover example with variable size genotypes. 38
4.3 Evaluating function block diagram. 38
4.4 Evaluating function flowchart diagram. 40
4.5 Genetic algorithm compiling procedure. 41
4.6 Fitness evolution samples. 42
4.7 Evolution in time of the fitness mean and standard deviation. 42

5.1 Rule analysis flowchart diagram. 46
5.2 Rotation invariant rule space. 47

5.3 Test programs. 48
5.4 Rule test flowchart diagram. 50
5.5 2-bit counter programming and execution. 56

List of Tables

1.1 Traditional vs Cellular Computing 3

3.1 Physical media and technology. 18
3.2 Machine code for the 2-input NAND Boolean function. 28
3.3 Truth table of the 2-input NAND function. 29

4.1 Comparison between a traditional compiler and the proposed genetic algorithm compiler. 36

5.1 Rotation invariant mapping. 44
5.2 Rule seed example. 44
5.3 Rule count and number tested rules per local behavior. 45
5.4 Genetic algorithm parameters. 49
5.5 Average and maximum scores per local behaviors. 49
5.6 Rules with the highest scores. 51
5.7 Mapping of rule 48336 and rule 62658. 52
5.8 Output Karnaugh maps for rule 48336 53
5.9 Best solutions for rule 48336. 53
5.10 Simple Boolean logic functions for 5×5 cellular automata using rule 48336. 54
5.11 Simple memory function for 5×5 cellular automata using rule 48336. 55
5.12 2-bit counter output in time. 55

Chapter 1

Introduction

Humans, capable of computing on their own, invented tools to facilitate this task. The abacus and algebra are examples. In time these tools evolved into computers and computer sciences. The complexity and reach of these tools has increased in time, becoming harder to handle.

Common computer architectures are the result of many years of constant refinement. The need for faster algorithm execution has increased computer hardware complexity, and increased the number of computing elements used. In time all kinds of computers architectures have emerged. These computer hardware can be classified according to the simplicity, parallelism and locality of its elements. Figure 1.1 illustrates computing artifacts cataloged under this classification. *Cellular Computing*, the ultimate computer architecture illustrated in the figure, accomplishes vast parallelism with the simplest computing elements interacting locally.

Figure 1.1: Cellular Computing.

1.1 Traditional Computers

Today's computers are mostly based on the fifty-year old Von Neumann architecture. This architecture matches human's thinking nature of solving problems in a sequential approach, due to lack of resources to perform common activities. For example, humans are required to perform a sequential set of steps to build a sand castle because they have only 2 hands and a couple of eyes. Von Neumann architecture based devices are conceived as execution tools of the sequential set of steps or instructions that we usually visualize to solve problems. Von Neumann computers can be interconnected to perform parallel computing; but the nature of these computing elements remains sequential.

The computer industry improvements come from increasing either the efficiency of computing elements of the number computing elements.

- **Complex Instruction Set Computers** - The first types of commercially successful computers were developed to execute programs written by humans. Programmers were supplied with a rich set of instructions.
- **Reduced Instruction Set Computers** - The advent of compilers enabled the creation of computers with a simple but efficient instruction set. Computer hardware was no longer defined directly by the programmer's abilities. The parallelism is gathered at instruction level.
- **Parallel Computers** - The increasing need of computer power leads to the use of several coordinated computing elements and the creation of computers with several functional units.

1.2 Reconfigurable Computers

One of the latest trends in the computer industry is called Reconfigurable Computing. This computing approach tries to take advantage of the hardware parallelism for data processing by embedding the algorithms in a flexible hardware. The Field Programmable Gate Arrays (FPGAs) are the core technology that has made Reconfigurable Computing possible [1][2]. Instead of solving a problem using a sequential set of instructions over a CPU, Reconfigurable Computing uses the capability of today's Field Programmable Gate Arrays to perform the desired behavior connecting hardware primitives.

1.3 Cellular Computers

Nature is concurrent as events may occur simultaneously in time but in different places. These events are interrelated in a causal and spatial sense as a consequence of the space and time invariance of physical laws. The quantum theory states that the universe is discrete at certain physical level and the relativity theory states that nothing can travel beyond the speed of light. Thus a computer model based on the above mentioned constrains is required to effiently exploit nature's parallelism. The cellular automata are such models, and *Cellular*

Computing is the name used to describe this type of computing architecture. A comparison between cellular computing models and traditional computers is shown in Table 1.1.

Aspect	Traditional	Cellular
Orientation	Human	Physical
Execution	Sequential	Parallel
Programming	Algorithmic	Behavioral
Processing	Centralized	Distributed
Memory	Centralized	Distributed
Hardware	Complex	Simple
Software	Complex	Highly Complex
Scalability	Hard	Easy

Table 1.1: Traditional vs Cellular Computing

The main characteristic of cellular automata are the massive parallelism, the element's simplicity, and the locality of the element's interactions. The difficulty of designing a cellular computer lies on how to program it. A solution to this problem may be found in nature.

A computer architecture based on cellular automata would be reconfigurable. Reconfigurable computing has shown better performance compared with traditional computing in several data processing applications[3].

1.4 Living Computers

The human beings emerged from a matter and energy evolutive process. An evolutive process has also created our perception of the universe. The ideas conforming the human's perception are going through a constant reproduction, mutation, and selection process. A person communicating an idea to other people is an effort resulting in the generation of instances of the idea in terms of each person's context.

When a person communicates an idea to other people, the idea is instantiated in everyone's mind, as a relation of the ideas that each one already has. Different perceptions create mutated ideas and relating old ideas creates new ideas again. When an idea is not useful any more it is forgotten and dies.

Computer Science, a revolutionary idea with significant impact in the world, has been broadly developed for 500 years. The act of computing has been performed since the beginning of life by any organism that has to solve a problem to ensure its own existence. As an example, a mouse is capable of moving objects to reach food. The act of solving a problem or computing a solution can be conceived as changing the state of reality to increase the survival opportunity of the computing entity. The field of genetic algorithms has successfully applied these concepts in the solution of problems on many areas[4].

1.5 Problem Description

The parallelism is abundant in nature but difficult to exploit. Computers are designed with a lot of legacy ideas behind, ideas that may shadow some better solutions. The cellular automata computing model promises to use this parallelism efficiently. A physical implementation of the cellular automaton, required to efficiently exploit the parallelism, has to simple enough to perform the most basic task with a small number of cells. A major impediment of cellular automata is the difficulty of using their complex behavior to perform useful computations.

The software for cellular automata computer architecture will be extremely complex to human interpretation. Its complexity can be compared with living organism's DNA information. The compiling tool used to generate the cellular automata software will have no precedence.

1.6 Objective and Contributions

The objective of this thesis is to propose a computer architecture based on cellular automata. The architecture is developed having in mind physical implementation. A fixed rule cellular automaton is used to simplify the architecture's hardware. The hardware complexity is transferred to the software layer using genetic algorithms to handle the software complexity. The computer complexity is laid in software instead of the hardware.

A method to program the architecture is proposed. The architecture and programming method are tested with simple problems. The proposed architecture and programming paradigm are the main contributions. Some concepts are defined for the first time, such as cellular automata boundary stimulus, mobility rate, retention rate, and genetic algorithm based compiler. Other concepts are opened for further research. One of the main difficulties of traditional computers is the memory bottleneck problem of the Von Neumann architecture. This problem could disappear in the proposed architecture, because the memory would be distributed all over the cellular automata like in data flow architectures. The performance of such architecture could be superior to traditional computer architectures in highly parallel applications.

1.7 Development Stages

The following path is suggested in the development of a general-purpose computer based on cellular automata.

1. Choose a physical media for cellular automata.
2. Define the architecture topology.
3. Develop the compiler.
4. Specify the cell behavior.

5. Design and evolve an operating system.

6. Achieve a self-containing system.

The first four aspects are explored in the present document.

1.8 Document Organization

This document starts with a background explanation in chapter 2 of the basic concepts needed on reconfigurable computing, cellular automata and genetic algorithms. The related work to these concepts is introduced. In chapters 3 and 4, the proposed architecture and compiler are defined in detail, with explanations on the main considerations followed. Chapter 5 describes the methodology followed to define the architecture parameters, and presents the results obtained. The last chapter includes the conclusions and describes the future work.

Chapter 2

Background

2.1 Cellular Automata

John Von Neumann and Stanislaw Ulam conceived cellular automata in theory in the late 40's. Cellular automata started to be used in the study and simulation of complex systems in the late 60's. Recently the cellular automata have been successfully used to solve problems[5].

Cellular automata are discrete dynamical systems often described as discrete counterpart to partial differential equations, which describe continuous dynamical systems. Discrete means a finite countable number of states for the space, time and cellular automaton properties. The basic idea is to describe a complex system by simulating the interaction of cells following a simple behavior. In other words, do not describe a complex system with complex equations, but let the complexity emerge by interaction of simple individuals following simple equations.

Cellular automata are discrete space and time logical universes, obeying their own "local" physics. Space in cellular automata is partitioned into discrete elements called "cells" and time progresses in discrete steps. "Local" means that the state of a cell at time $t + 1$ is a function only of its own state and the states of its immediate neighbors at time t, this function is commonly known as the cell's *rule*. Each cell is in one of a finite number of states at a specific time. The physics of this logical universe is deterministic meaning that the future evolution is uniquely determined once a rule and an initial state of cellular automata have been chosen.

Each finite automaton consists of a finite set of cell states Σ, a finite input alphabet α, and a transition function Δ, which is a mapping from the set of neighborhood states to the set of cell states. Letting N be the number of neighbors:

$$\Delta : \Sigma^N \rightarrow \Sigma$$

A cellular automaton is a d-dimensional lattice with a finite automaton residing at each lattice site. Each automaton takes as input the states of the automata within some finite local region of the lattice, defined by a neighborhood. Typical 2-dimensional neighborhoods are shown in Figure 2.1. In Von Neumann's neighborhood any new state of cell at (i, j) depends on the previous state of cells at $(i, j - 1)$, $(i - 1, j)$, $(i + 1, j)$ and $(i, j + 1)$. In Moore's neighborhood any new state of cell at (i, j) depends on the previous state of cells

at $(i-1, j-1)$, $(i, j-1)$, $(i+1, j-1)$, $(i-1, j)$, $(i+1, j)$, $(i-1, j+1)$, $(i, j+1)$ and $(i+1, j+1)$.

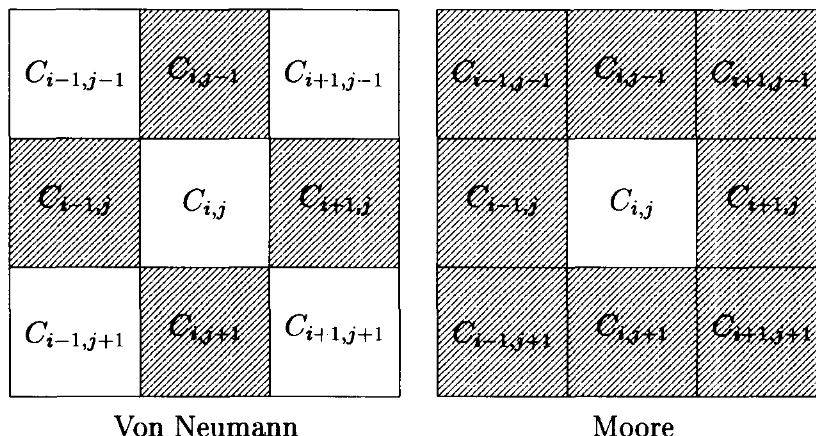


Figure 2.1: Frequently used 2D neighborhoods.

If all the cells in the cellular automaton share the same transition function we have a *homogeneous* cellular automaton. The transition function of this type of cellular automaton is named cellular automaton's *rule.*

A one-dimension cellular automaton is commonly used to illustrate how cellular automata work[6]. Figure 2.2 shows a one-dimensional cellular automata transition function. The vertical axis represents time progression starting from the top. The cellular automaton configuration is recorded in each time instant. The horizontal axis represents the one-dimensional space field. In this example neighbors, the one on the left, the one on the right and the cell itself define the cell's neighborhood. The cellular automaton initial configuration consists on a single black cell in the middle. Figure 2.3 shows the first time steps of evolution, and Figure 2.4 shows the result after 100 step.

Figure 2.2: One-dimensional cellular automata transition function example.

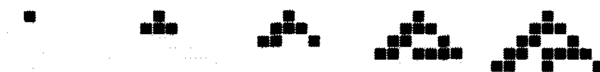


Figure 2.3: First time steps of the one-dimension cellular automaton example.

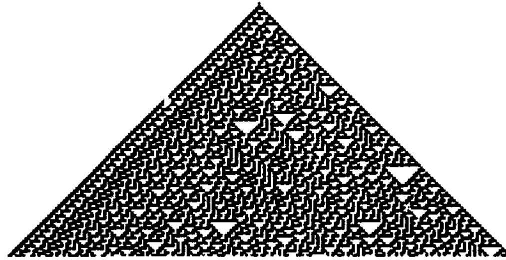


Figure 2.4: Result on the one-dimension cellular automaton after 100 step.

2.2 Reconfigurable Computing

Algorithms implemented in hardware run faster than if they were executed by software over the same hardware technology.

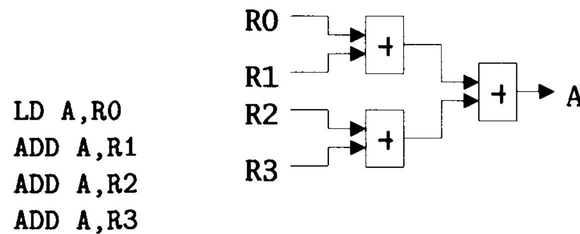


Figure 2.5: Hardware implemented algorithm vs Software implemented algorithm

Xilinx developed the first Field Programmable Gate Array (FPGA) in 1985. The FPGAs are electronic devices that can be programmed to implement a defined logic behavior. Field Programmable Gates Arrays make Reconfigurable Computing possible. Reconfigurable Computing is the presence of both hardware, that can be reconfigured to implement specific computing functionality (i.e. FPGA), and software with the ability to change the hardware data-path for optimizing the performance. A typical FPGA topology is shown on Figure 2.6.

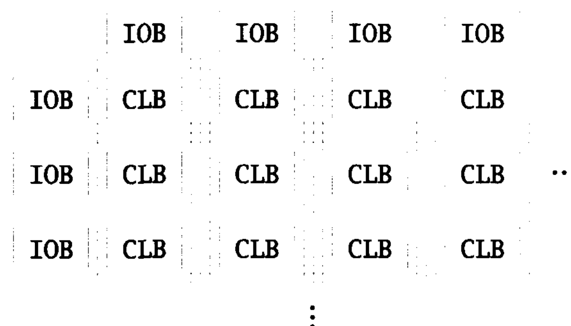


Figure 2.6: Typical FPGA topology.

An FPGA can be considered as a cellular automaton with a complex cell defined by the Control Logic Blocks (CLBs) and a non-homogeneous and almost global neighborhood as a consequence of the complex interconnection network. This interconnection network uses a big portion of the silicon die area. This fact suggests that cellular automata with local neighborhood and a simple rule will achieve higher transistor densities because no global communication interconnections are needed.

2.3 Genetic Algorithms

All genetic algorithms work on a population, a collection of several alternative solutions, to the given problem[4][7][8]. Each individual in the population is called a string, or chromosome, in analogy to chromosomes in natural systems. Often these individuals are coded as binary strings, and the individual characters or symbols in the strings are referred as genes. In each genetic algorithm iteration a new generation is evolved from the existing population in an attempt to obtain better solutions.

The population size determines the amount of information stored by the genetic algorithm. The genetic algorithm population is evolved over a number of generations. An evaluation function or fitness function is used to determine the fitness of each candidate solution. The *fitness* is an indicator of how suitable is an individual to solve a problem. The evaluation function is usually user-defined, and problem specific.

Individuals are selected from the population for reproduction, with the selection biased toward more highly fit individuals. Selection is one of the key operators on genetic algorithms that ensure survival of the fittest. The selected individuals form pairs, called parents.

- **Crossover** - It is the main operator used to reproduction. It combines portions of two parents genotypes to create two new individuals that may inherit a combination of the features of the parents. For each pair of parents, crossover is performed with a high probability.

- **Mutation** - It is an incremental change made to each member of the population with very small probability. Mutation enables the introduction of new features into a population. Mutation is performed probabilistically. The probability of a gene change in known as mutation probability.

The Simple Genetic Algorithm

The simple genetic algorithm is illustrated in Figure 2.7 flowchart diagram. The simple genetic algorithm is composed of individuals or chromosomes, and three evolutionary operators: selection, crossover, and mutation. The chromosomes are binary coded. Each chromosome is an encoded solution to the problem, and each individual has an associated application dependent fitness. The initial population can be either randomly generated or user supplied. A highly fit population is evolved through several generations by selecting, and crossing two individuals with a given mutation rate to improve the population. Selection is done probabilistically biased towards more highly fit individuals maintaining the population as an

unordered set. Distinct generations are evolved, and the process of selection, crossover, and mutation are repeated until all individuals in a new generation are defined. Then the old generation is discarded. New generations are evolved until some stopping criterion is met. The genetic algorithm may be limited to a fixed number of generations, or it may be terminated when all individuals in the population converge to the same string or no improvements in fitness values are found after a given number of generations. Since selection is biased toward more highly fit individuals, the fitness of the overall population is expected to increase in successive generations. However, the best individuals may appear in any generation.

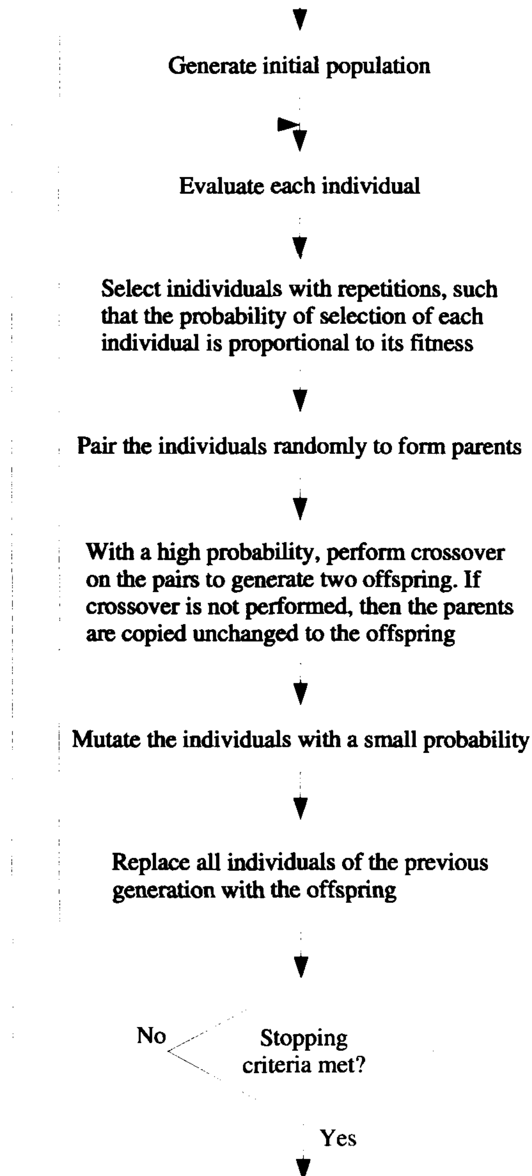


Figure 2.7: Simple genetic algorithm flowchart.

The genetic operators and their significance can now be explained. The description will be in terms of a traditional genetic algorithm without any problem-specific modifications. The operators to be discussed are the selection operator, the crossover operator, and the mutation operator.

Selection

The objective of the genetic algorithm is to converge to an optimal individual, and selection pressure is the driving force that determines the rate of convergence. A high selection pressure will cause the population to converge quickly, possibly at the expense of a sub optimal result.

- **Roulette wheel selection** - The roulette wheel selection is a proportionate selection scheme, the slots of a roulette wheel are sized according to the fitness of each individual in the population. An individual is selected by spinning the roulette wheel and noting the position of the marker. The probability of selection an individual is therefore proportional to its fitness.

- **Stochastic universal selection** - The stochastic universal selection is a less noisy version of roulette wheel selection, N equidistant markers are placed around the roulette wheel, where N is the number of individuals in the population. N individuals are selected in a single spin of the roulette wheel, and the number of copies of each individual selected is equal to the number of markers inside the corresponding slot.

- **Tournament selection** - A number N of individuals are taken at random from the population without replacement. The one with higher fitness is copied to the target population the others are set aside. When the population is empty it is refilled with the original population. The process is repeated until the target population size is collected. Therefore the best individual will be selected N times, and the worst will not be selected at all.

Crossover

Once two chromosomes are selected, the crossover operator is used to generate two offspring. One-point, two-points and uniform crossovers are commonly used. In one-point crossover a position is selected at random between the first and the Lth gene, where L is the chromosome length. The two parents are crossed at the selected point. See Figure 2.8. In two-point crossover a second position is selected at random. The crossover genes are those in between the two points. In uniform crossover, each chromosome position is crossed with some probability, typically one-half.

Parent 1:	1	1	0	1	1	0	1	0	0	0
Parent 2:	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**
Offspring 1:	1	1	0	**1**	**0**	**0**	**0**	**1**	**0**	**1**
Offspring 2:	**0**	**0**	**0**	1	1	0	1	0	0	0

Figure 2.8: One-point crossover.

Parent 1:	1	1	0	1	1	0	1	0	0	0
Parent 2:	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**
Offspring 1:	1	1	0	**1**	**0**	**0**	**0**	**1**	0	0
Offspring 2:	**0**	**0**	**0**	1	1	0	1	0	**0**	**1**

Figure 2.9: Two-point crossover.

Parent 1:	1	1	0	1	1	0	1	0	0	0
Parent 2:	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**
Offspring 1:	**0**	**0**	0	**1**	1	0	1	**1**	0	**1**
Offspring 2:	1	1	**0**	1	**0**	**0**	**0**	0	**0**	0

Figure 2.10: Uniform crossover.

Mutation

As new individuals are generated, each gene is mutated with a given probability. A mutation of a binary coded genetic algorithm mutation can be done flipping a bit. In a non binary coded genetic algorithm, mutation involves randomly generating a new alphabet character in a specified position.

2.4 Related Work

The proposed architecture relates to the Complex Adaptive Systems theory. This theory studies the emergent phenomena of systems made of many interacting elements. Complex Adaptive Systems theory includes topics such as neural networks, chaos theory, information physics, cellular automata, and genetic algorithms. Only a few publications on genetic algorithms and cellular automata were related with the proposed architecture and the proposed programming methodology.

The bibliography of Reconfigurable Computing is mostly focused on applications requiring existent hardware. The *Plastic Cell Architecture*[9] is the publication related to new hardware approaches. All related works differ with each other in objectives and contributions. The relationship between the objectives and contributions of each previous work, and this thesis is pointed out in the following sections. The differences of the previous works and this thesis highlight and clarifies the thesis contributions.

2.4.1 The Plastic Cell

The use of cellular automata in reconfigurable computing has been studied recently[9]. A new architecture reference based on programmable logic devices called Plastic Cell Architecture is proposed. The Plastic Cell Architecture is described as a reference for implementing a mechanism of fully autonomous reconfigurability. This reconfigurability is a further step toward general-purpose reconfigurable computing introducing variable-grain and programmable-grain parallelism to wired logic computing. The Plastic Cell Architecture is a fusion of an SRAM-based FPGA and cellular automata, where the cellular automata are dedicated to support the run time activities of the circuits configured on the architecture. The Plastic Cell Architectures follows the object-oriented paradigm, in that the circuits are regarded as objects. These objects can be described in a hardware description language that features the semantics of dynamic module instantiation.

The fundamental difference of the Plastic Cell Architecture and the architecture proposed in this thesis is the design philosophy. The Plastic Cell Architecture follows a top-down design philosophy, while the architecture proposed in this thesis follows a bottom-up approach. This means that the Plastic Cell Architecture defines the high level details "global behavior" prior the low level definition "local behavior". This restrictions implies a cellular automata cell with complex behavior oriented to facilitate the development of programming and synthesis tools. Higher element density can be obtained using the approach followed in the present thesis. The advantage are simpler cells, but at the expense of more a complex programming

problem. The architecture proposed is going to exhibit as emergent phenomena some of the high level attributes embedded in the Plastic Cell Architecture.

2.4.2 Universal Cellular Automata

Wolfram has proved the existence of cellular automata with universal computing capabilities[6]. He suggests that the existence of universal cellular automata is in fact common, and that if a cellular automaton has the universality property then it should be able to emulate any other cellular automaton. Perrier, Sipper, and Zahnd show the development of cellular automata capable of instantiating universal computers and capable of performing self-reproduction[10]. However the universal computer proposed is sequential. This means that the massive parallelism of the cellular automata is not exploited. Margolus found universal cellular automata based on physical models[11]. The approach used in these models is to mimic some natural phenomena and identify model's computational capability. The approach followed by the current thesis is to identify models with computational capabilities that can be implemented in physical artifacts, using an available technology.

2.4.3 Cellular Automata Machines

The most successful cellular automata hardware has been the Cellular Automata Machines (CAM)[12]. Cellular Automata Machines are designed with off-the-shelf components as a hardware accelerator to simulate cellular automata; the hardware is not a cellular automaton itself. Conventional computer are ill suited to run cellular automata models, and so discourage their development. Nevertheless examples of physical models for which the best computational models are cellular automata exist. Low-cost cellular automata multiprocessor are possible arranging the same quantity and quality of hardware as one might find in low-end workstations, obtaining large cellular automata calculations as good as any existing supercomputer. This machine architecture is of performance at cellular automata calculations much superior to that of existing supercomputers, but vastly inferior to what a fully parallel cellular automata machine could achieve. The proposed architecture will be a genuine cellular automata implementation.

2.4.4 VLSI Design using Genetic Algorithms

Genetic algorithms have been successfully used in the development of VLSI designs tools[8]. Even commercially available Electronic Design Automation (EDA) software is starting to use genetic algorithms. The use of genetic algorithms to layout integrated circuits, and to do FPGA technology mapping is of special interest because the cellular automata programming problem faced in this thesis is related to technology mapping and layout problem of EDA.

2.4.5 Evolving Cellular Automata with Genetic Algorithms

A genetic algorithm was used to evolve cellular automata for two computational tasks: density classification and synchronization[5]. In both cases the genetic algorithm discovered rules

that gave rise to sophisticated emergent computational strategies. These strategies are analyzed using a "computational mechanics" framework in which "particles" carry information and interactions between particles effects information processing. The genetic algorithms are used to find cell rules that will allow the cellular automata to solve particular problems. The initial configuration of the cellular automaton represents the data and the cellular automaton rule is the program. When this rule is placed in a cellular automaton and executed repetitively, the cellular automaton will end up with a configuration in its cells that represents the solution to the problem that the rule was designed to solve. Mitchell does not considered implementation issues. These experiments demonstrate that with genetic algorithms, cellular automata can be commanded to perform useful massively parallel computing. This finding is the starting point to try using genetic algorithm to program the proposed architecture.

Chapter 3

Architecture

The proposed cellular automata architecture has a homogeneous neighborhood with a fixed rule. A homogeneous neighborhood was selected because heterogeneous cellular automata do not have a single rule or connectivity scheme making them more complex and harder to implement. If the architecture is supposed to be of a general purpose, then the cellular automata rule shall be universal.

3.1 Input/Output

A cellular automaton with a finite number of cells and array-like ordered cells always define a cellular automata boundary. The cell's neighbors in this boundary are undefined. The missing communication links of these elements are used as the I/O interface of the architecture. An example is shown in Figure 3.1.

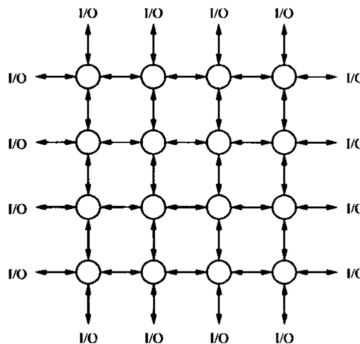


Figure 3.1: Cellular automata input/output interface boundary.

3.2 Programming and Execution

The data and program are stored in the configuration of the cellular automata (current state of the cells). Most of the computer models based on cellular automata assume an

arbitrary state on each cell as the initial state [11][6][12]. Thus, each cell is connected to the external world to obtain the initial state breaking the local nature of cellular automaton. Another approach is to have a rule with a transition state from the programming stage to the processing stage to enter data and program from the outermost cells as an information stream[9]. However these program/data cell behavior requires a cell with complex rule. This work proposes a simpler rule to make the programming and processing stages emerge as a cellular automata global behavior. Thus, global behaviors are important elements for the cellular automata rule selection.

The program is entered from the outermost cells boundary in the *programming cycle.* After the iterations, these outer stimuli take the cellular automaton to the target configuration state. The cellular automaton enters into the *execution cycle* and starts processing data stimulus received in the input/output boundary after reaching the target configuration state. The processing result is finally placed in the input/output boundary after an appropriate processing time. See Fig. 3.2.

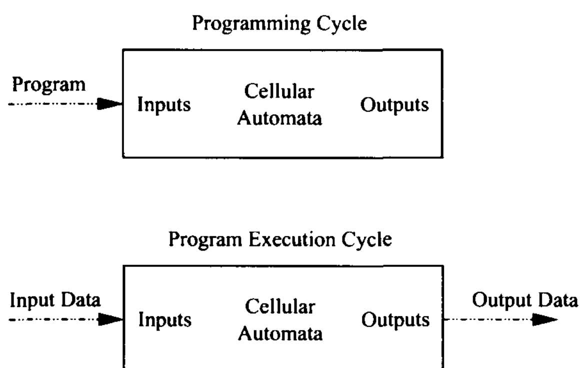


Figure 3.2: Cellular automata programming and execution cycles.

The cellular automaton has no distinction between data and program information. This fact opens the door to a general-purpose computing, in the same way that the Von Neumann architecture did by allowing the data and program to reside in the same memory space.

3.3 Physical Media

Truly cellular automata architecture should be implemented in a natural cellular lattice in order to exploit nature's parallelism. The selected physical media imposes restrictions to the architecture. Table 3.1 illustrates some examples of physical media, in most cases the technology to construct cellular automata is under development.

Physical Media	Technology
Organic Tissue	Development
Crystalline Structure	Development
Polymer	Development
Quantum Elements	Development
Silicon Integrated Circuits	Mature

Table 3.1: Physical media and technology.

The most feasible implementation of cellular automata with the current technology is in silicon integrated circuits. The CMOS silicon integrated circuits fabrication process enforces the use of 2 dimensional cellular automata lattice with rectangular components. Digital instead of analog elements are useful due to their deterministic behavior. The transistor density of such architecture would be superior to any other computer architecture.

3.4 Cellular Automata Topology

The cellular automata neighborhood that fits best with a 2D rectangular lattice is the Von Neumann neighborhood. Each cell has communication only with the upper, lower, left and right neighbors. The cell is kept simple to obtain the highest density. The communication between cells is 1-bit wide. The cell behavior is modeled by a 4-input, and 4-output combinational function with no hidden states. The total number of possible input combinations is 2^4, and the total number of possible output combinations is 2^4. The cell's behavior can be fully represented in a 16-entry lookup table with each input mapped to one of 16 possible output values.

The right side of Figure 3.3 shows the graphic convention used to represent a cell. The i label is located over the place from were the input signal is read, o label is situated over the place were the output data of the cell will be set. Each input and output is differentiated by an integer suffix starting from zero in the upper most position. The left side of Figure 3.3 shows a 3×3 cellular automaton. The cellular automata's I/O labels are placed on the CA boundary in the same way as in the cell representation, but the integer suffix starts from zero in the upper-left most position.

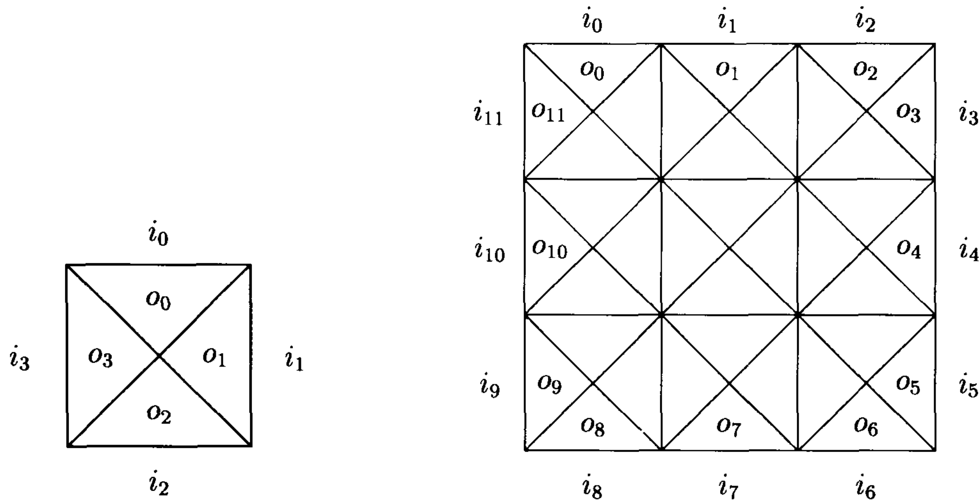


Figure 3.3: Cell and cellular automata graphic representation convention.

3.5 Cell Behavior

The local and global behaviors of homogeneous cellular automata are defined by the cell's state transition function commonly known as the cellular automaton's rule. A description of the local and global behaviors used in chapter 5 as rule selection criteria follows. Keep in mind that in order to improve element density the cellular automata cell shall be as simple as possible.

3.5.1 Local Behaviors

Local behaviors are characteristics that the cellular automata cell has by itself. The cell's local behaviors depend on the cell's rule exclusively. A description of the local behaviors of interest follows. Each local behavior includes a test procedure coded in Java to verify if a cell's rule has the local behavior. The test procedure assumes that the cell's rule is encoded in an input to output lookup table, `int Mapping[16]`. Each possible input combination has a unique integer representation that is used as the lookup table index (i_0 is the least significant bit and i_4 is the most significant bit). The lookup table has a corresponding output value for each input. The output is an integer number that represents each of the possible output combinations (o_0 is the least significant bit and o_4 is the most significant bit). The input space size and output space size are defined as follow.

```
int InputStateSpaceBitSize = 4;
int OutputStateSpaceBitSize = 4;
int InputStateSpaceSize = 16;
int OutputStateSpaceSize = 16;
```

- **Rotation Invariability** - A rotation invariant rule is defined as a rule that behaves equally after been rotated in any topological possible way.

```
public boolean isRotationInvariant()
{
    int or, ir;
    /* Assume the rule is rotation invariant by default. */
    boolean answer = true;
    /* Check each possible input entry. */
    for (int i = 0; (i < InputStateSpaceSize) && answer; i++)
    {
        ir = i;
        or = Mapping[i];
        /* Check each possible rotation. */
        do
        {
            /* Rotate the input. */
            ir = (ir*2) % InputStateSpaceSize + (ir*2) / InputStateSpaceSize;
            /* Rotate the output. */
            or = (or*2) % InputStateSpaceSize + (or*2) / InputStateSpaceSize;
            /* If the output of the rotated input is not equal
               to the original output rotated */
            if (Mapping[ir] != or)
            {
                /* then the rule is not rotation invariant. */
                answer = false;
            }
        } while ((ir != i) && answer);
    }
    return answer;
}
```

- **Reversibility** - A reversible rule has the property of relating each input state to a unique non-repeatable output state. This means that from any cellular automata configuration it is possible to go backwards in time.

```
public boolean isReversible()
{
    /* Assume the rule is reversible by default. */
    boolean answer = true;
    boolean[] used = new boolean[OutputStateSpaceSize];
    /* Mark each output combination as unused. */
    for (int i = 0; i < OutputStateSpaceSize; i++)
    {
        used[i] = false;
    }
    /* Check each possible input entry. */
```

```
    for (int i = 0; (i < InputStateSpaceSize) && answer; i++)
    {
        /* If the input entry output combination is not marked as used*/
        if (!used[Mapping[i]])
        {
            /* mark the output as used. */
            used[Mapping[i]] = true;
        }
        else /* If the output combination is already marked as used */
        {
            /* the rule is not reversible. */
            answer = false;
        }
    }
    return answer;
}
```

- **Value Conservation** - A Value Conservative rule is defined as a function that has the same number of 0's and 1's in the outputs as in the inputs for all possible inputs transition values. The value conservation local behavior is particularly important because it implies that the amount of 0's and 1's is conserved giving us some means to control the amount of activity that happens in the cellular automaton. This characteristic can be compared with the energy conservation law that rules our universe.

```
public boolean isValueConservative()
{
    /* Assume the rule is value conservative by default. */
    boolean answer = true;
    /* Check each possible input entry. */
    for (int i = 0; (i < InputStateSpaceSize) && answer; i++)
    {
        /* If the hamming magnitud (number of 1's) in the input entry
           is not equal to the hamming magnitud of the output
           combination then */
        if (getHammingMagnitud(i) != getHammingMagnitud(Mapping[i]))
        {
            /* the rule is not value conservative. */
            answer = false;
        }
    }
    return answer;
}
```

- **Value Symmetry** - In a value symmetric function,

$$f_j(\overline{i_{n-1}}, \ldots, \overline{i_1}, \overline{i_0}) = \overline{f_j(i_{n-1}, \ldots, i_1, i_0)}$$

for each $j = 0, 1, \ldots m$.

```
public boolean isValueSymmetric()
{
    int ivc,ic,icv;
    /* Assume the rule is value symmetric by default. */
    boolean answer = true;
    /* Check each possible input entry. */
    for (int i = 0; (i < InputStateSpaceSize) && answer; i++) {
        /* Obtain the complement of the output combination. */
        ivc = OutputStateSpaceSize-1 - Mapping[i];
        /* Calculate the complement of the input combination. */
        ic = InputStateSpaceSize-1 - i ;
        /* Obtain the output of input combination complement. */
        icv = Mapping[ic];
        /* If complement of the output combination is not equal to
           the output of input combination complement then */
        if (icv != ivc)
        {
            /* the rule is not value symmetric. */
            answer = false;
        }
    }
    return answer;
}
```

- **Complement Invariability** - In a complement invariant function,

$$f_j(i_{n-1}, \ldots, i_1, i_0) = f_j(\overline{i_{n-1}}, \ldots, \overline{i_1}, \overline{i_0})$$

for each $j = 0, 1, \ldots m$.

```
public boolean isComplementInvariant()
{
    int ivc,ic,icv;
    /* Assume the rule is complement invariant by default. */
    boolean answer = true;
    /* Check each possible input entry. */
    for (int i = 0; (i < InputStateSpaceSize) && answer; i++)
    {
        /* Obtain the output combination. */
        iv = Mapping[i];
        /* Calculate the complement of the input combination. */
        ic = InputStateSpaceSize-1 - i ;
```

```
        /* Obtain the output of input combination complement. */
        icv = Mapping[ic];
        /* If complement of the output combination is not equal to
           the output of input combination then */
        if (icv != iv)
        {
            /* the rule is not complement invariant. */
            answer = false;
        }
    }
    return answer;
}
```

- **Axial Symmetry** - A rule with axial symmetry has the property of behaving in the same way after rotating any of its spatial axes by 180 degrees. The difference with a rotation invariant rule in a 2D lattice is that the rotation invariant rule behaves equally when the normal axis is rotated.

```
/* Note: This test procedure is only valid on rotation invariant rules. */
public boolean isAxialSymmetric()
{
    int left, right;
    /* Assume the rule is complement invariant by default. */
    boolean answer = true;
    int j;
    /* Check each possible input entry. */
    for (int i = 0; (i < InputStateSpaceSize) && answer; i++)
    {
        /* Obtain the input value on the right. */
        right = i & 0x2;
        /* Obtain the input value on the left. */
        left = i & 0x8;
        /* Swap the left and right input values. */
        j = (i & 0x1) | (i & 0x4) | (left >> 2) | (right << 2);
        /* If the output 0 remains changes when the left and
           right input values are swapped then
        if ((Mapping[i] & 0x1) != (Mapping[j] & 0x1))
        {
            /* the rule has no axial symmetry. */
            answer = false;
        }
    }
    return answer;
}
```

- **Mobility Rate** - The number of possible single bit changes in the input state that result on a bit change in an output bit for all possible input states. This property has influence on the cellular automata ability to move data.

```
/* Note: This test procedure is only valid on rotation invariant rules. */
public int getMovilityRate()
{
    /* The bit changes counter is set to 0. */
    int count = 0;
    /* Check each possible input entry. */
    for (int i = 0; i < InputStateSpaceSize; i++)
    {
        /* Check each input bit on the input entry. */
        for (int j = 1; j < (1 << InputStateBitSize); j = j << 1)
        {
            /* If output 0 changes when the input bit is
               toggled then */
            if ((Mapping[i ^ j] & 0x1) != (Mapping[i] & 0x1))
            {
                /* add one to the bit changes counter. */
                count++;
            }
        }
    }
    /* Return the total number of two-entry Karnaugh groups that
       produce an output change. */
    return count / 2;
}
```

- **Retention Rate** - The number of possible single bit changes in the input state that does not result on a bit change in an output bit for all possible input states. This property has influence on the cellular automata ability to hold the program.

```
/* Note: This test procedure is only valid on rotation invariant rules. */
public int getRetentionRate() {
    /* The bit changes counter is set to 0. */
    int count = 0;
    /* Check each possible input entry. */
    for (int i = 0; i < InputStateSpaceSize; i++)
    {
        /* Check each input bit on the input entry. */
        for (int j = 1; j < (1 << InputStateBitSize); j = j << 1)
        {
            /* If output 0 does not change when the input bit is
```

```
                toggled then */
            if ((Mapping[i ^ j] & 0x1) == (Mapping[i] & 0x1))
            {
                /* add one to the bit no changes counter. */
                count++;
            }
        }
    }
    /* Return the total number of two-entry Karnaugh groups
       that does not produce an output change. */
    return count / 2;
}
```

3.5.2 Global Behaviors

The selected rule will have repercussion on the cellular automata processing and storage capabilities. The rule selection in chapter 5 shall be oriented to find cellular automata rules with these capabilities by enclosing them in desirable global behaviors.

- **Program Retention** - The programmed behavior shall not change when the data is entered.
- **Data Mobility** - Ability of data to move within the cellular automaton without change.
- **Data Independence** - Data streams should not interact unless some kind of interaction is wanted.

3.6 Implementation

When a rule is finally selected it defines most of the integrated circuit hardware because it can be synthesized once, and the resulting layout can be copied in a mesh pattern to fill all die area. The way in which cell communication takes place shall be delineated. Two approaches are suggested.

3.6.1 Synchronous

- **Global Clock** - A global clock signal is distributed to each cell. Some cells will update its outputs with a high level phase, why others will respond to the low level phase of the clock signal in a chess check board layout.
- **Local Synchronization Signals** - Each cell is synchronized only with its neighbors, resulting in global synchronization because they are all interconnected. The cellular automata step time is driven by the boundary.

3.6.2 Asynchronous

An asynchronous approach results after placing the combinational logic that executes the cell's rule connected directly with the input of the combinational logic of the neighbor cells without any kind of buffering. This approach has broad implications so it is leaved behind for future research.

3.7 Machine Language

The machine language of the proposed architecture is the set of all possible stimuli that the cellular automaton can receive on the input boundary. The machine language is not explicitly designed, it emerges as the global behaviors of the selected rule executed in the cellular automaton.

A machine instruction $I[t]$ is defined as the stimulus space vector entered to the cellular automaton at a given time instant.

$$I[t] = (i_0[t], i_1[t], i_2[t], \ldots, i_{n-2}[t], i_{n-1}[t])$$

where n is the number of I/O boundary communication links.

A program is then the set of machine instructions applied in time to the cellular automaton to make it behave as desired.

$$P = (I[0], I[1], \ldots, I[N-2], I[N-1])$$

where N is the total number of instructions (time steps) required to program the cellular automaton.

The convention used to represent programs in following examples, is to write them in binary representation, separating each instruction with a single space, starting the program with the first instruction in time $I[0]$ and starting each instruction with the first input in the boundary space i_0.

3.8 Programming and Execution Example

This section shows an example of how to load a program and execute it. The example's purpose is to clarify how the architecture actually works. The example uses a 5×5 cellular automaton with a fixed homogeneous rotation invariant rule. The rule's seed number is 48336. The procedure used to select the cellular automaton's rule is detailed in chapter 5. Figure 3.4 shows the rule's state transition table, were each possible input state is mapped to next cell's output state.

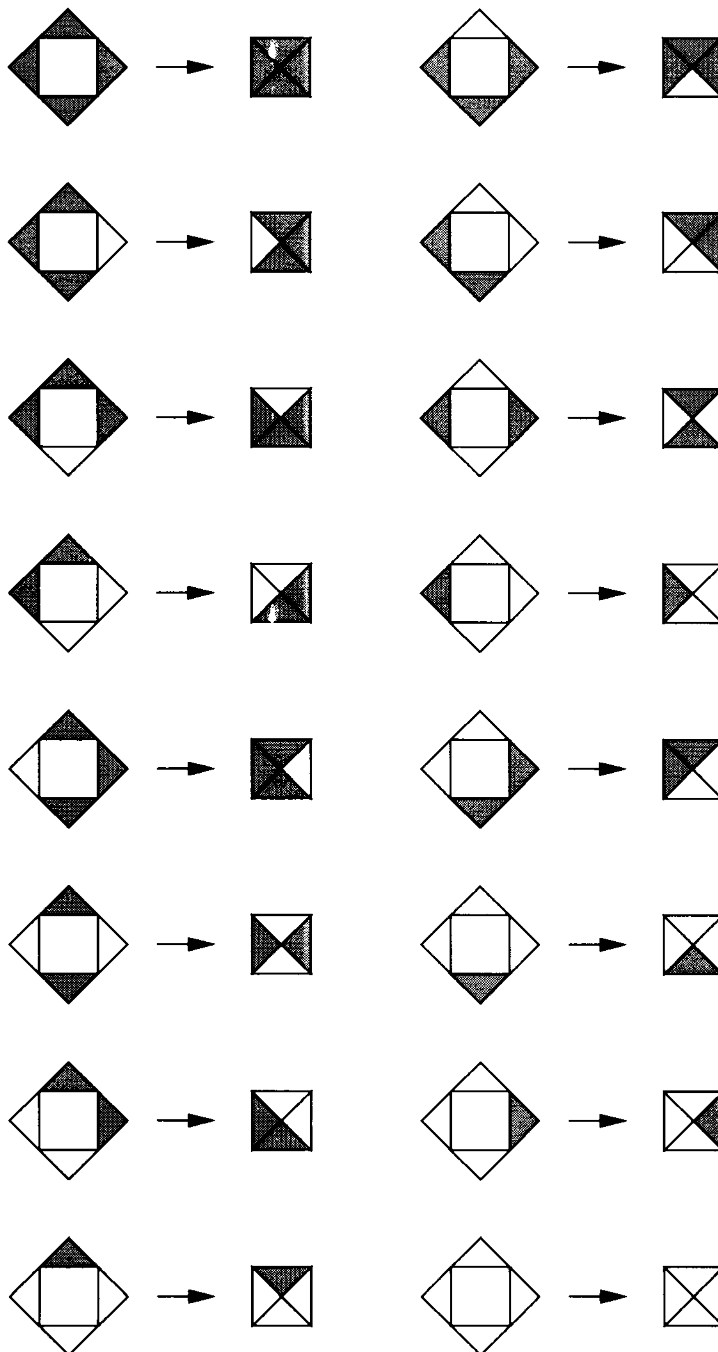


Figure 3.4: Rule 48336 state transition table.

The example consists of a simple program. The architecture will be commanded to execute a two input NAND Boolean function. The function returns the result on output o_5, using as input the data entered at locations i_{14} and i_{18}. The desired behavior was translated to cellular automaton programming stimulus (machine instructions) using the procedure detailed in chapter 4. The resulting object code is presented on table 3.2, it has only one machine instruction.

t	I[t]
0	10000000000100111101

Table 3.2: Machine code for the 2-input NAND Boolean function.

The example is divided in three steps.

1. **Apply programming stimulus**
The first step is to feed the object code in the cellular automaton. The cellular automaton shall be in a known initial state. In this case the cellular automaton cells have 0's in all their outputs. See Figure 3.5. The object code depends on this condition to work properly.

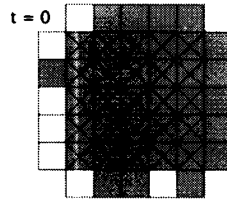


Figure 3.5: Cellular automaton initial configuration.

The last stimulus entered is kept constant from now on. Only the input terminals i_{14} and i_{18} will eventually change. The instruction is kept in the cellular automaton's input boundary during various time steps until the cellular automaton reaches a steady state. The steady state is actually the result of executing the NAND function on the default input data in the last machine instruction programmed.

$$o_5 = \overline{i_{18} i_{14}} = \overline{10} = 1 \tag{3.1}$$

The output terminal o_5 exhibits the previous result by time $t = 5$. The programming sequence is showed in Figure 3.6. The cellular automaton goes into steady state in time $t = 12$.

2. **Enter test data and verify results**
In order to verify that the cellular automaton behaves correctly, all possible input data combinations will be introduced. The output signal o_5 can be compared against the 2-input NAND Boolean function truth Table 3.3.

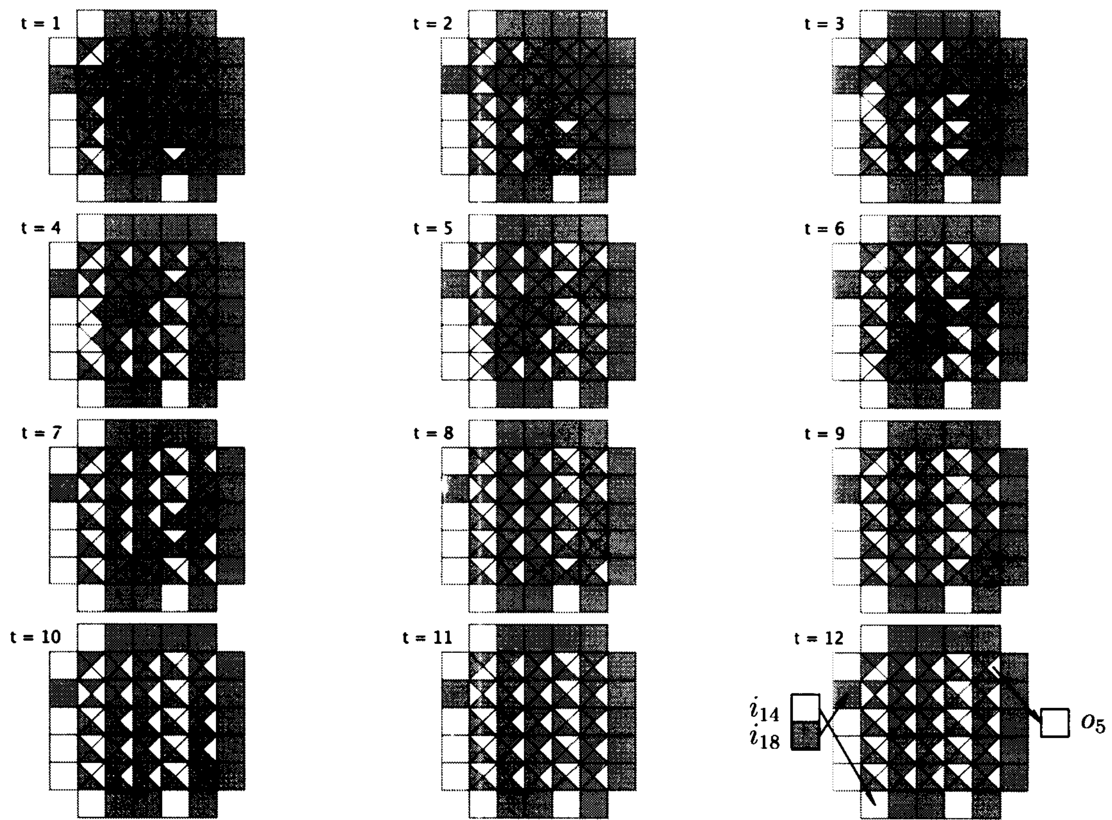


Figure 3.6: Cellular automaton programming example.

i_{18}	i_{14}	o_5
0	0	1
0	1	1
1	0	1
1	1	0

Table 3.3: Truth table of the 2-input NAND function.

The answer is not generated immediately after a new input combination is entered. It takes a delay time of 9 steps to observe the expected answer. The output takes the value of the NAND function evaluated with input values at i_{18}, 5 steps earlier, and i_{14}, 9 step earlier. Equation 3.2 summarizes this notion.

$$o_5[t] = \overline{i_{18}[t-5]i_{14}[t-9]} \tag{3.2}$$

Figures 3.7, 3.8, 3.9 and 3.10 show the evolution in time of the cellular automaton for in each input combination.

The first input combination, $i_{18} = 0$ and $i_{14} = 0$, is entered at time $t = 13$. See Figure 3.7. Only i_{14} changes since it was equal to 1 at time $t = 12$. This single change leads to 8 steps of cellular automaton activity from $t = 13$ to $t = 20$. The cellular automaton is considered in steady state at $t = 21$, because no configuration change is visible at $t = 22$. The value on output o_5 is considered the desired result once the cellular automaton reaches steady state at $t = 21$, this value can be verified with the first row $\overline{00} = 1$ in Table 3.3.

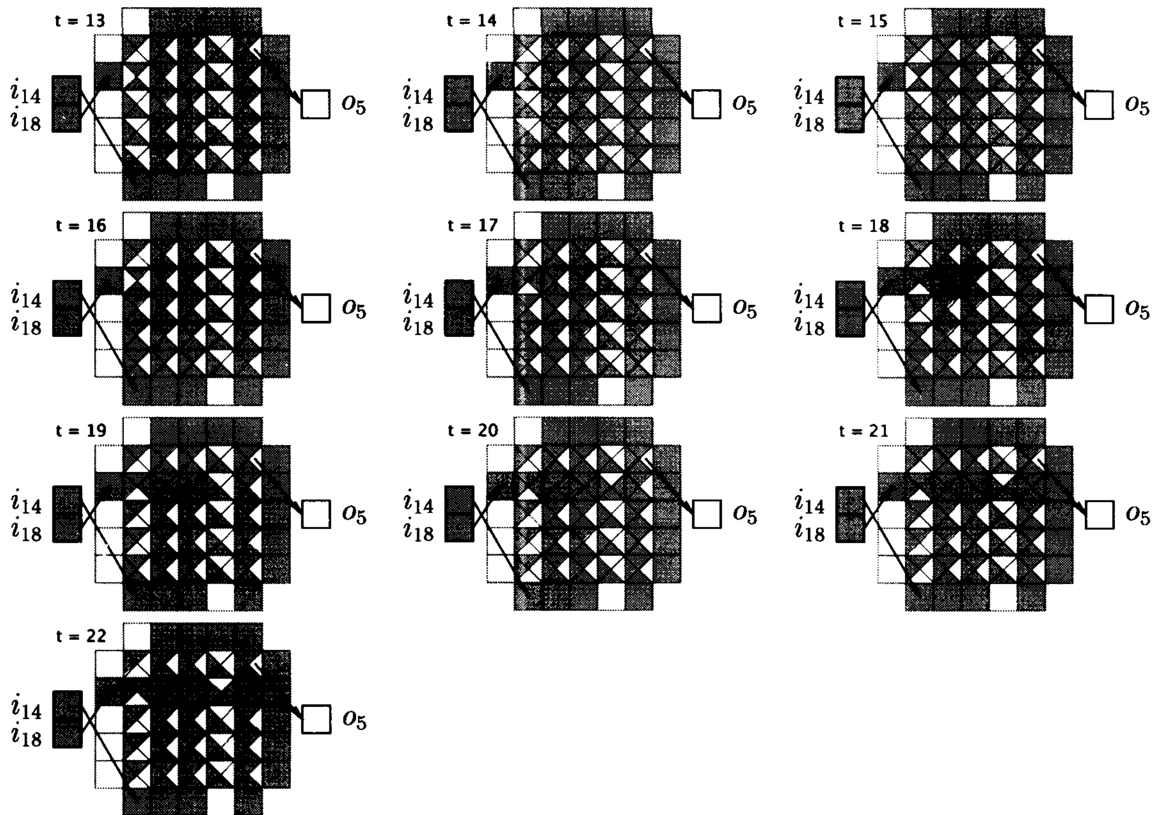


Figure 3.7: Cellular automaton execution example $o_5 = \overline{i_{18} i_{14}} = \overline{00} = 1$.

The second input combination, $i_{18} = 0$ and $i_{14} = 1$, is entered at time $t = 23$. See Figure 3.8. Only i_{14} changes since it was equal to 0 at time $t = 22$. This single change leads to 11 steps of cellular automaton activity from $t = 23$ to $t = 33$. The cellular automaton is considered in steady state at $t = 34$, because no configuration change is visible at $t = 35$. The value on output o_5 is considered the desired result once the cellular automaton reaches steady state at $t = 34$, this value can be verified with the second row $\overline{01} = 1$ in Table 3.3.

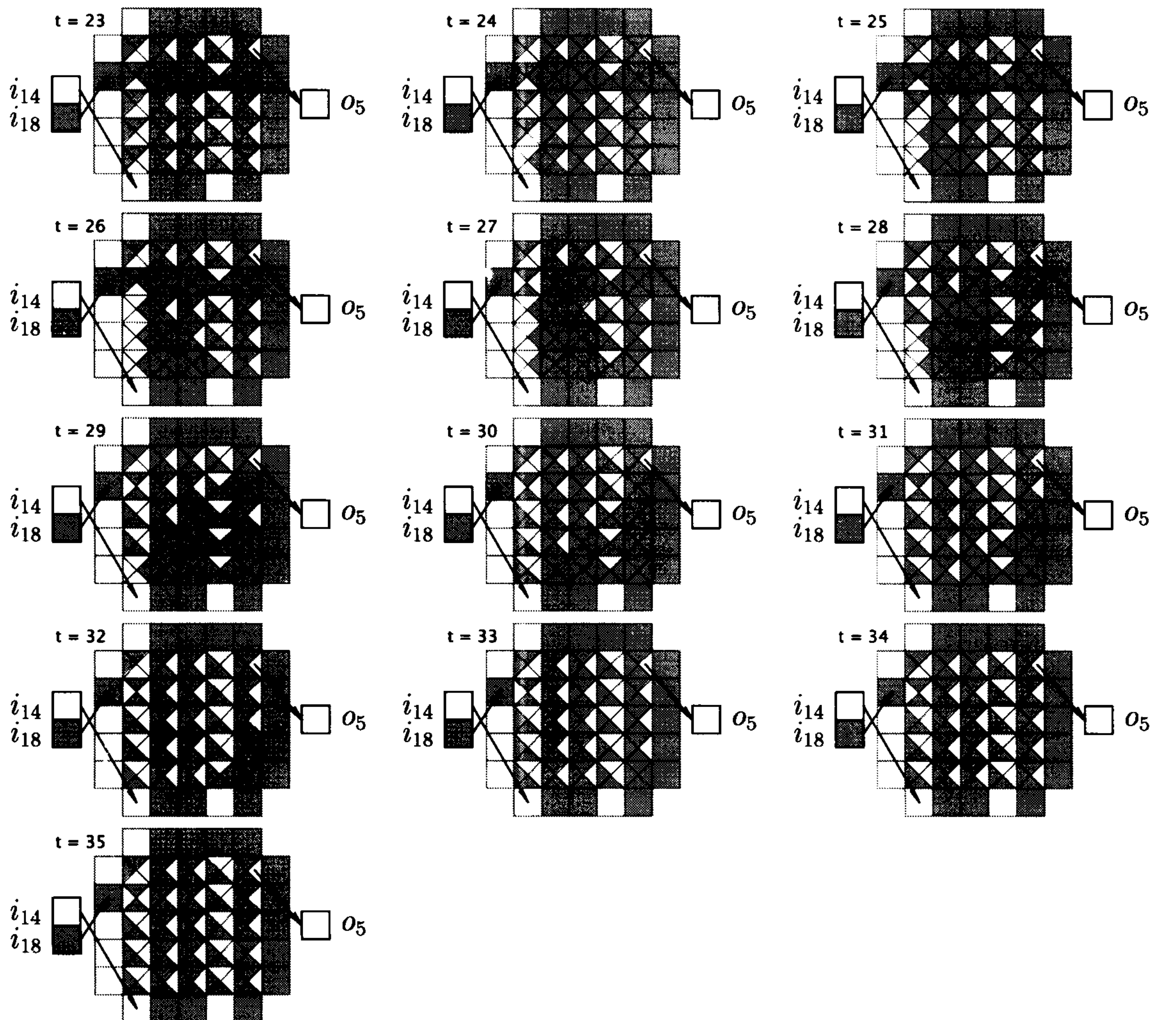


Figure 3.8: Cellular automaton execution example $o_5 = \overline{i_{18}i_{14}} = \overline{01} = 1$.

The third input combination, $i_{18} = 1$ and $i_{14} = 0$, is entered at time $t = 36$. See Figure 3.9. Both inputs i_{18} and i_{14} changed since they had 0 and 1 respectively at $t = 35$. This change leads to 9 steps of cellular automaton activity from $t = 36$ to $t = 44$. The cellular automaton is considered in steady state at $t = 45$, because no configuration change is visible at $t = 46$. The value on output o_5 is considered the desired result once the cellular automaton reaches steady state at $t = 45$, this value can be verified with the third row $\overline{10} = 1$ in Table 3.3.

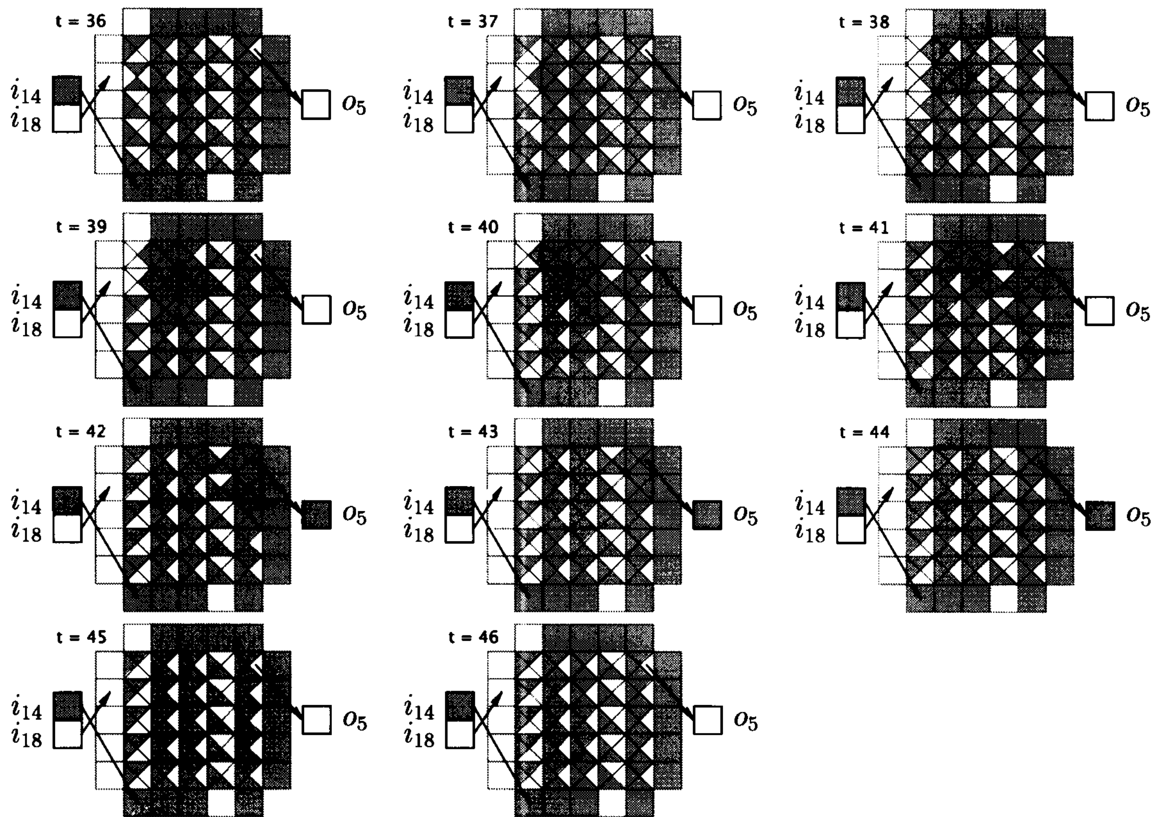


Figure 3.9: Cellular automaton execution example $o_5 = \overline{i_{18}i_{14}} = \overline{10} = 1$.

The last input combination, $i_{18} = 1$ and $i_{14} = 1$, is entered at time $t = 47$. See Figure 3.10. Only i_{14} changes since it was equal to 0 at time $t = 46$. This single change leads to 9 steps of cellular automaton activity from $t = 47$ to $t = 55$. The cellular automaton is considered in steady state at $t = 56$, because no configuration change is visible at $t = 57$. The value on output o_5 is considered the desired result once the cellular automaton reaches steady state at $t = 56$, this value can be verified with the last row $\overline{11} = 0$ in Table 3.3.

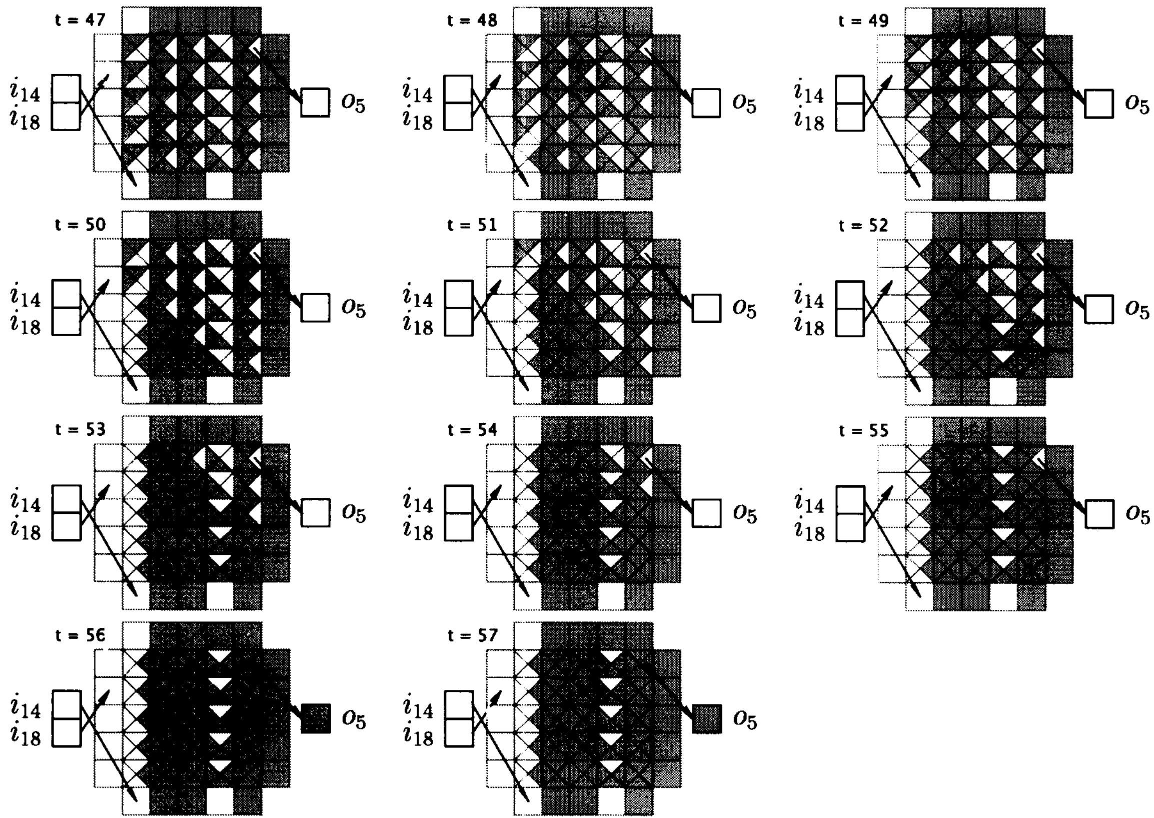


Figure 3.10: Cellular automaton execution example $o_5 = \overline{i_{18} i_{14}} = \overline{11} = 0$.

3. **Program unloading**
   The last step of the current example will show how the cellular automaton configuration can be restored to initial state. This probes the cellular automata capability of been reconfigured, meaning that cellular automaton configuration can be changed once it is already configured. Applying new stimulus to the inputs performs the reconfiguration. There are two reconfiguration stages. The first stage consists on a set of stimulus that shall return the cellular automaton to a known initial configuration. In the second stage the programming stimulus of the new configuration are applied, just as if it was programmed for the first time.

   In this example the cellular automaton will be returned to the initial configuration applying zeros on all the inputs, Figure 3.11. This configuration flush property is a direct consequence of using a cell rule with the Value Conservation local behavior.

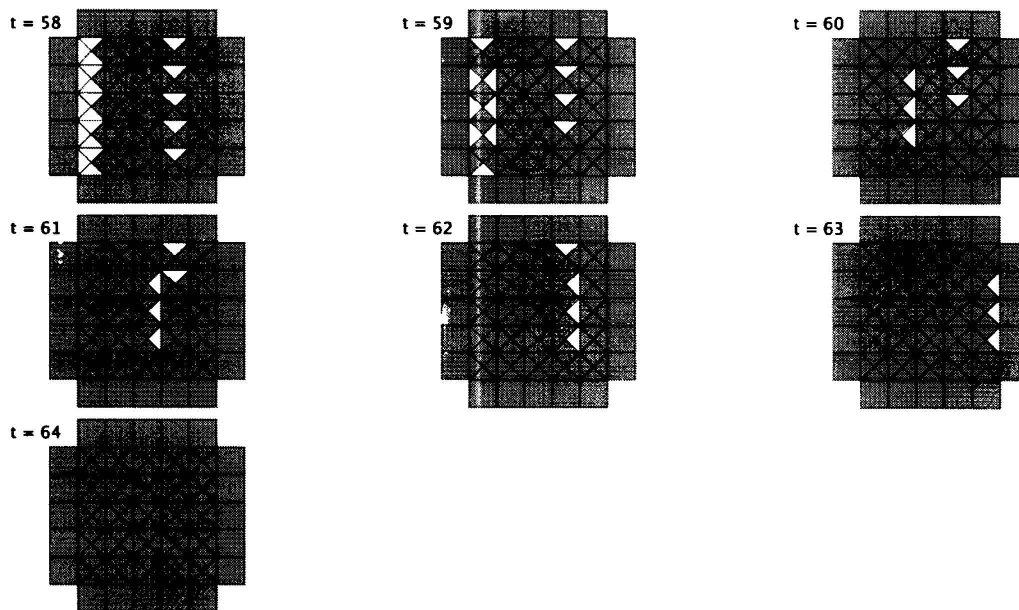


Figure 3.11: Cellular automaton program unloading example.

This chapter introduced the guideline concepts of the proposed architecture, but many details have not been defined yet. Further research is needed to understand the implications of such decisions. The following chapter will describe a programming methodology that can be used to start programming the architecture. This activity will be a cornerstone in the definition of the missing characteristics.

Chapter 4

Compiler

The compiler's objective is to translate the user's algorithms into a set of bits required to configure the target architecture. The proposed architecture has no instruction set. The machine language for the proposed architecture emerges from the cell's interaction of the cellular automaton. The compiling task, determine a set of bits to make cells interaction result in a specific behavior, becomes very complex. Traditional compilers and programming tools are unable to perform this complex task. This chapter presents the approach based on a genetic algorithm to perform such a complex compiling task.

4.1 A Compiler based on Genetic Algorithms

The genetic algorithm takes an important role in the compiler tool chain for this type of architecture. The genetic algorithm's objective is to find the machine language code that makes the cellular automata behave as wanted. A comparison between the proposed genetic algorithm compiler and traditional compiler is shown in table 4.1.

Characteristics	Traditional	Genetic Algorithm
Can handle cellular automata complexity	No	Yes
Deterministic	Yes	No
Amount of computational resources needed	Low/Medium	High
Correctness	Always	Not always

Table 4.1: Comparison between a traditional compiler and the proposed genetic algorithm compiler.

The proposed genetic algorithm based compiler will not always return correct solutions in a strict logical sense. The fact that it will return candidate solutions is good enough for the scope and objectives of the present research. Future work can be addressed to overcome this disadvantage.

4.2 Genetic Algorithm

Figure 4.1 shows the flow chart of the proposed genetic algorithm. The algorithm starts with a random initial population. Each individual represents a possible solution to the problem in the form of machine language object code. The population's individuals are evaluated. The genetic algorithm evaluates an individual giving and indicator of its performance. Evaluation results are the guideline to create a new population with higher survival probabilities assigned to best individual's genotypes. If the genetic algorithm's ending criteria is meet the algorithm will return the population at hand. Otherwise it will generate a new population applying the tournament, crossover, resizing and mutation genetic operators.

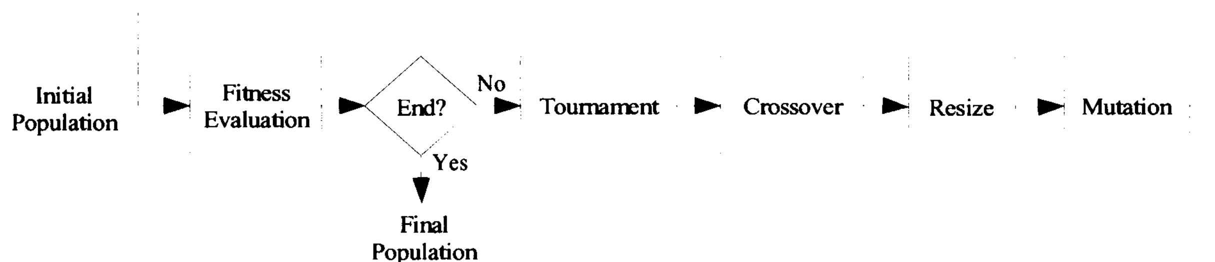


Figure 4.1: Genetic algorithm.

4.3 Genotype Coding

The cellular automata programs are coded in a bit vector (the genotype) starting the program with the first instruction in time $I[0]$ and starting each instruction with the first input in the boundary space i_0. See section 3.7. A variable size genotype is used to store programs, because programs may be of different lengths. The initial population is set at random, with a random number of instructions on each individual, and random bit values. The initial object code size is uniformly distributed between 1 and 5 machine instructions.

4.4 Genetic Operators

The genetic operators modify the population to induce the evolution of a solution. Only the *resize* and *crossover* operators are non-standard. A description of the genetic operators used follows:

- **Tournament** - This operator scrambles the population in a circular list, and passes a fixed size window in all possible places of the list. In each window place, the individuals within the window are compared and a new population is created adding a copy of the individual with best fitness.
- **Crossover** - The population is scrambled on a list. Individuals are paired starting from the beginning of the list. Couples are selected at a fixed crossover rate. The

Parent 1:	1	1	0	1	1	0	1	0	0	0	0	0	1	0	1	1
Parent 2:	**1**	**0**	**0**	**1**	**0**	0	0	1								
Offspring 1:	**1**	**0**	**0**	**1**	**0**	0	1	0	0	0	0	0	1	0	1	1
Offspring 2:	1	1	0	1	1	0	0	1								

Figure 4.2: Crossover example with variable size genotypes.

first instruction is the reference to align the couple of genotypes. A cross point is picked at random somewhere in the middle of the individual with shorter genotype (genotype boundaries position are also considered). All the bits before the cross point are interchanged between the couple members. See an example in Figure 4.2. In the example, the cross point is located to the right of $i_4[0]$. All the bits on the left of the cross point are swapped to generate the offspring.

- **Resize** - Individuals are resized at a fixed resize rate. Size increment and decrement are equally possible. Genotype growth is done duplicating the last machine instruction at the end of the genotype. Genotype shrink is performed eliminating the first machine instruction.

- **Mutation** - Each bit on each genotype of the population is toggled at a fixed mutation rate.

4.5 Evaluating Function

The genetic algorithm evaluating function must return an indicator or fitness amount on how well an individual behaves as desired. The indicating value is obtained by correlating in time the cellular automata output with the desired output after applying a random data test signal in the cellular automata inputs, see the block diagram in Figure 4.3. The use of large test signals is encouraged to obtain similar fitness values after each try, and to test the program under a broader input sample space.

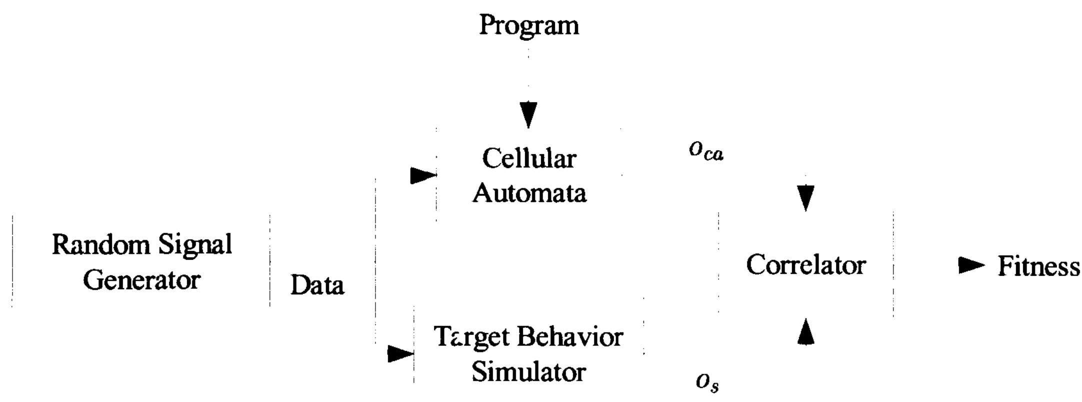


Figure 4.3: Evaluating function block diagram.

The source code is entered as a part of the evaluating function. It is coded in the Target Behavior Simulation block of Figure 4.3. The Target Behavior Simulator drives the genetic algorithm to find machine language (object code) that closely emulates the target behavior. The desired behavior is expressed in the Target Behavior Simulator as coded Boolean functions. These type of functions are enough to describe the test behaviors needed in this research.

The fitness is measured as the correlation peak in time,

$$fitness = \max\left(\sum_{k=-\infty}^{\infty} \overline{o_s[k] \oplus o_{ca}[k-t]}\right) \tag{4.1}$$

where o_s is the simulator output, o_{ca} is the cellular automaton output, and max() is a function that returns the maximum value of the argument evaluated in all possible time instants. For example $\max(\sin(t)) = 1$. If test signals of variable length are used, the fitness shall be normalized dividing equation 4.1 by the test signal length,

$$fitness_{normalized} = \max\left(\frac{\sum_{k \in T} \overline{o_s[k] \oplus o_{ca}[k-t]}}{n}\right) \tag{4.2}$$

where T is the set of time values, and n is the number of elements in T.

The length in time of the random signal ensures that the cellular automaton preserves the same behavior. It is important to reset the cellular automata and reload the same program several times to guarantee that the input data has no effect in the fitness. The fitness is calculated as the arithmetic mean of all the executions. See Figure 4.4. This flow chart shows that the final fitness value is the result of testing the same program with many input conditions.

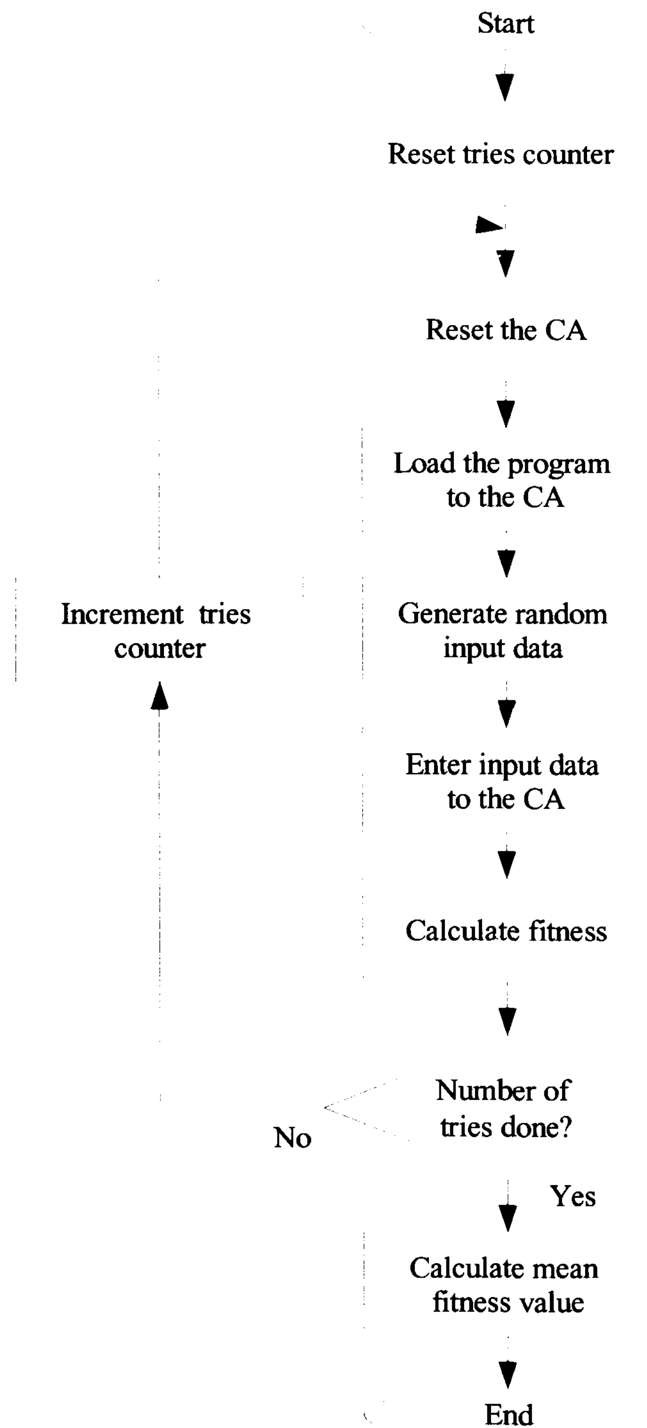


Figure 4.4: Evaluating function flowchart diagram.

4.6 Compiling Procedure

The genetic algorithm compilation procedure is shown in Figure 4.5. A verification stage is suggested after executing the genetic algorithm to ensure an acceptable solution. This verification stage can be done by executing some test cases, or by performing a correctness probe as explained in section 5.4.

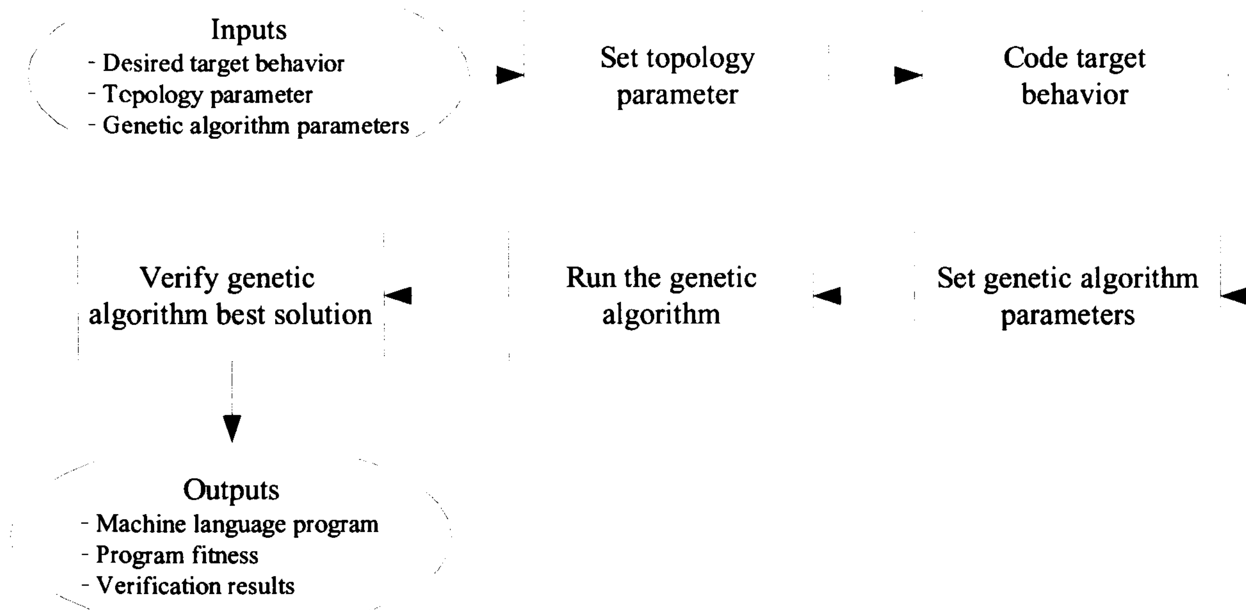


Figure 4.5: Genetic algorithm compiling procedure.

Figure 4.6 shows eight samples of the fitness evolution in time of the genetic algorithm after 20 generations. The genetic algorith was coded to solve the NAND function in section 5.5 using the same genetic algorithm parameters (Table 5.4). Only 3 samples reached almost optimal solutions, because those populations were able to construct the genotype information required to solve the problem. One hundred runs of the genetic algorithm were performed to obtain statistical regularity, the mean μ and standard deviation σ are presented in Figure 4.7. The fitness of the population evolves as expected. The standard deviation is significant as a consequence of the big selective pressure imposed in this genetic algorithm test. A tournament window of 5 individuals was used. The figures prove that the genetic algorithm is finding solutions in less than 20 generations. Optimal solutions are within $\mu + \sigma$ after 20 generations.

The programming methodology proposed in this chapter will be used in the following one to test different cellular automata rules. The results of such tests will help to ultimate some of the details that where leftover in the previous chapter, in particular the definition of the cell's rule.

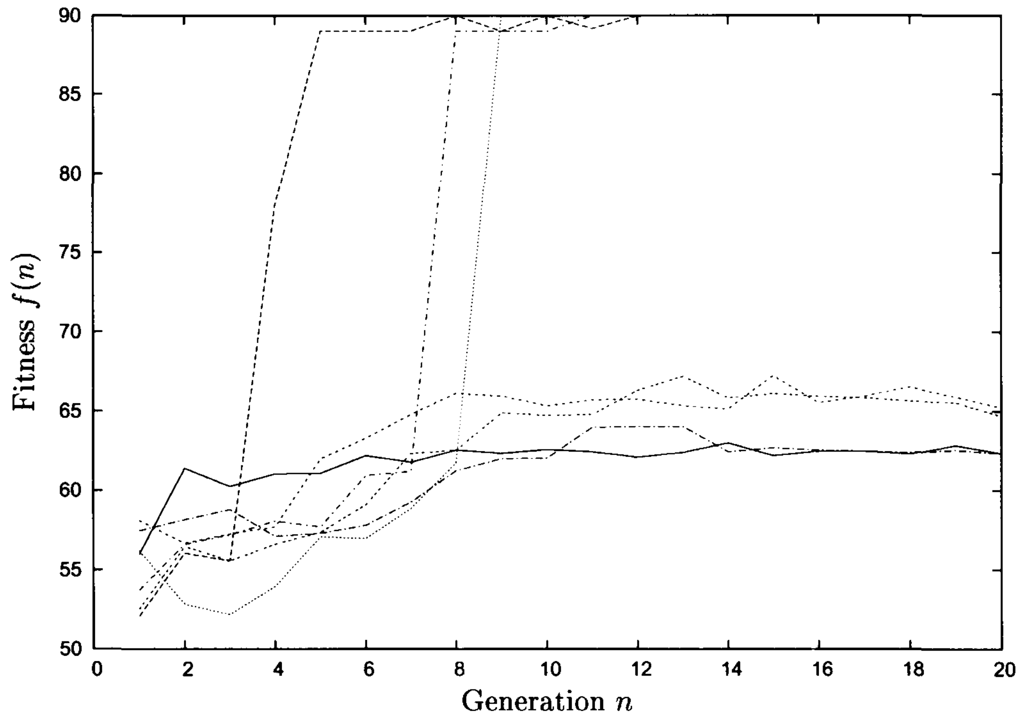


Figure 4.6: Fitness evolution samples.

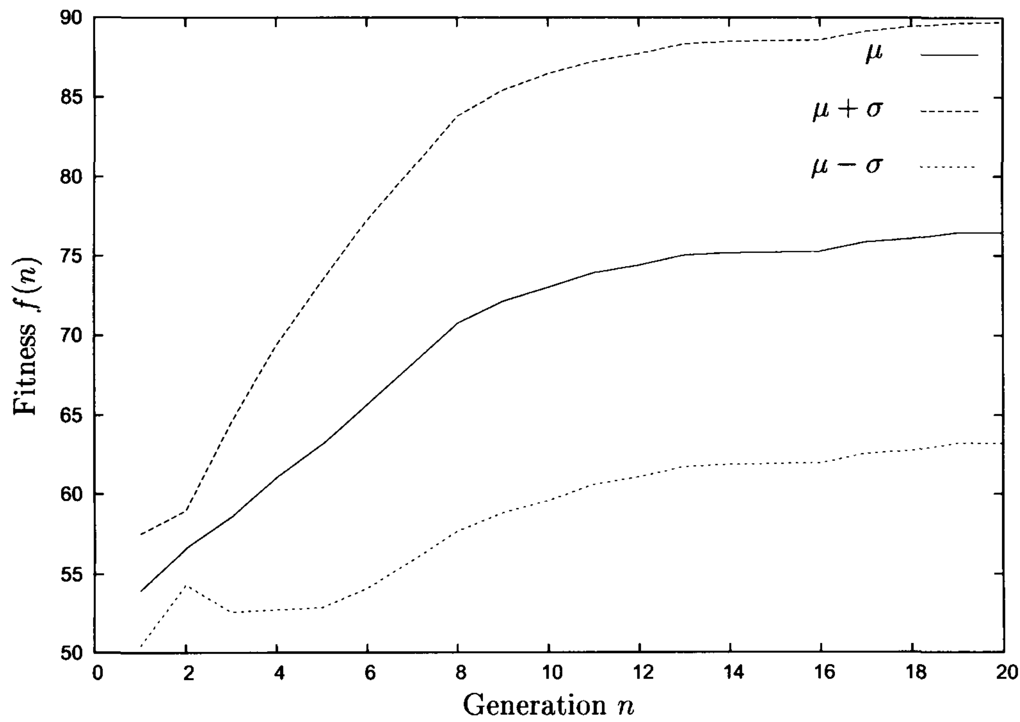


Figure 4.7: Evolution in time of the fitness mean and standard deviation.

Chapter 5

Methodology and Results

5.1 Research Platform

The rule analysis, rule search and simple programs tests of the following sections were performed on a Linux Pentium 4 workstation running at 2 GHz with 512 MB of RAM.

The cellular automata's, genetic algorithm and program verification packages were programmed in the Java language. A PostgreSQL database was created to store all the rule analysis, rule search and tests data. The following Java packages were developed:

- `gaf` - The Genetic Algorithm Framework, an application independent package to execute genetic algorithms.
- `careco` - The Cellular Automaton Reconfigurable Computing package. This package simulates the cellular automaton, analyzes the cellular automaton rules and uses the `gaf` package as a compiler.
- `symbolic` - The Symbolic Boolean logic package was developed to verify the genetic algorithm solutions in a strict logic sense.

5.2 Rule Search

The proposed architecture is based on a cellular automaton with a fixed rule. A rule defining the possible local behaviors, global behaviors and cellular automaton's computational efficiency must be selected carefully. The architecture topology presented in section 3.4 defines the space of possible rules from were one shall be selected. The criteria to select the fixed rule presented in this chapter made possible to select one rule from the large number of rules available for a cell with n inputs and m outputs. The number k of possible rules can be obtained from the equation shown below. The total number of rules is 2^{64} for a cell with $m = 4$ outputs and $n = 4$ inputs.

$$k = 2^{m2^n} \tag{5.1}$$

5.2.1 Rotation Invariance Selection Criterion

The *first criterion* focused on searching for the rotation invariant rules. The function for a specific output defines the function for all the other outputs under this criterion. The following equations show the rotation invariant function assignment.

$$o_0 = f_r(i_0, i_1, i_2, i_3) \quad (5.2)$$
$$o_1 = f_r(i_1, i_2, i_3, i_0) \quad (5.3)$$
$$o_2 = f_r(i_2, i_3, i_0, i_1) \quad (5.4)$$
$$o_3 = f_r(i_3, i_0, i_1, i_2) \quad (5.5)$$

A 16-bit number, known as seed, is assigned to output o_0, in order to represent a rule in the rotational invariant rule space. The truth table shown in Table 5.1 illustrates the seed assignment for each cell output and the definition of the other outputs using rotation invariant mapping. An example of a rule's seed is illustrated in Table 5.2.

i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0
0	0	0	0	s_0	s_0	s_0	s_0
0	0	0	1	s_2	s_4	s_8	s_1
0	0	1	0	s_4	s_8	s_1	s_2
0	0	1	1	s_6	s_{12}	s_9	s_3
0	1	0	0	s_8	s_1	s_2	s_4
0	1	0	1	s_{10}	s_5	s_{10}	s_5
0	1	1	0	s_{12}	s_9	s_3	s_6
0	1	1	1	s_{14}	s_{13}	s_{11}	s_7
1	0	0	0	s_1	s_2	s_4	s_8
1	0	0	1	s_3	s_6	s_{12}	s_9
1	0	1	0	s_5	s_{10}	s_5	s_{10}
1	0	1	1	s_7	s_{14}	s_{13}	s_{11}
1	1	0	0	s_9	s_3	s_6	s_{12}
1	1	0	1	s_{11}	s_7	s_{14}	s_{13}
1	1	1	0	s_{13}	s_{11}	s_7	s_{14}
1	1	1	1	s_{15}	s_{15}	s_{15}	s_{15}

Table 5.1: Rotation invariant mapping.

Seed	Binary Representation															
	s_{15}	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0
4660	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0

Table 5.2: Rule seed example.

The total number of rules k shows a significant reduction after applying the rotation invariability criterion. The new number of rotation invariant rules, calculated using the equation shown below, is 2^{16} for a cell with $m = 4$ outputs and $n = 4$ inputs.

$$k = 2^{2^n} \tag{5.6}$$

5.2.2 Local Behaviors Selection Criterion

The *second criterion* to select the rule takes account of the local behaviors previously defined in section 3.5.1, the selection of these behaviors is based on the need to reduce as much as possible the rule search space. The criterion is to select rules that satisfy two or more of the following local behaviors: reversible, value symmetric, value conservative and complement invariant. The axial symmetry local behavior was not included to reduce the number of rules under test.

An application program, called `careco.RuleAnalyze`, analyzed the rotational invariant rule space, and found the rules with two or more local behaviors. The application stores the results in the Rule Properties database, the flowchart diagram is shown in Figure 5.1. The application starts analyzing Rule 0 of the rotation invariant rule space. Each local behavior test is performed on the rule, and the results are stored in the rule properties database. Then, the next rule seed is generated and the rule analyzed. The `careco.RuleAnalyze` application keeps going until there are no more possible rules to analyze.

Local Behavior	Counted	Tested
Reversible	1536	224
Value Conservative	192	128
Value Symmetric	256	128
Complement Invariant	256	0
Axial Symmetric	4096	40

Table 5.3: Rule count and number tested rules per local behavior.

The application to analyze the rule obtained the rotation invariant space statistics illustrated in Table 5.3. The table shows the number of rules per local behavior in the rotation invariant rule space. The first column shows the name of the local behavior test performed. The count of rules that passed the local behavior test is in the second column. A total of 224 rules passed the *second criterion*; the statistics of these rules are shown in the third column. The rules that passed the *second criterion* will go through final selection tests. Note that all the rules that comply with the *second criterion* are reversible.

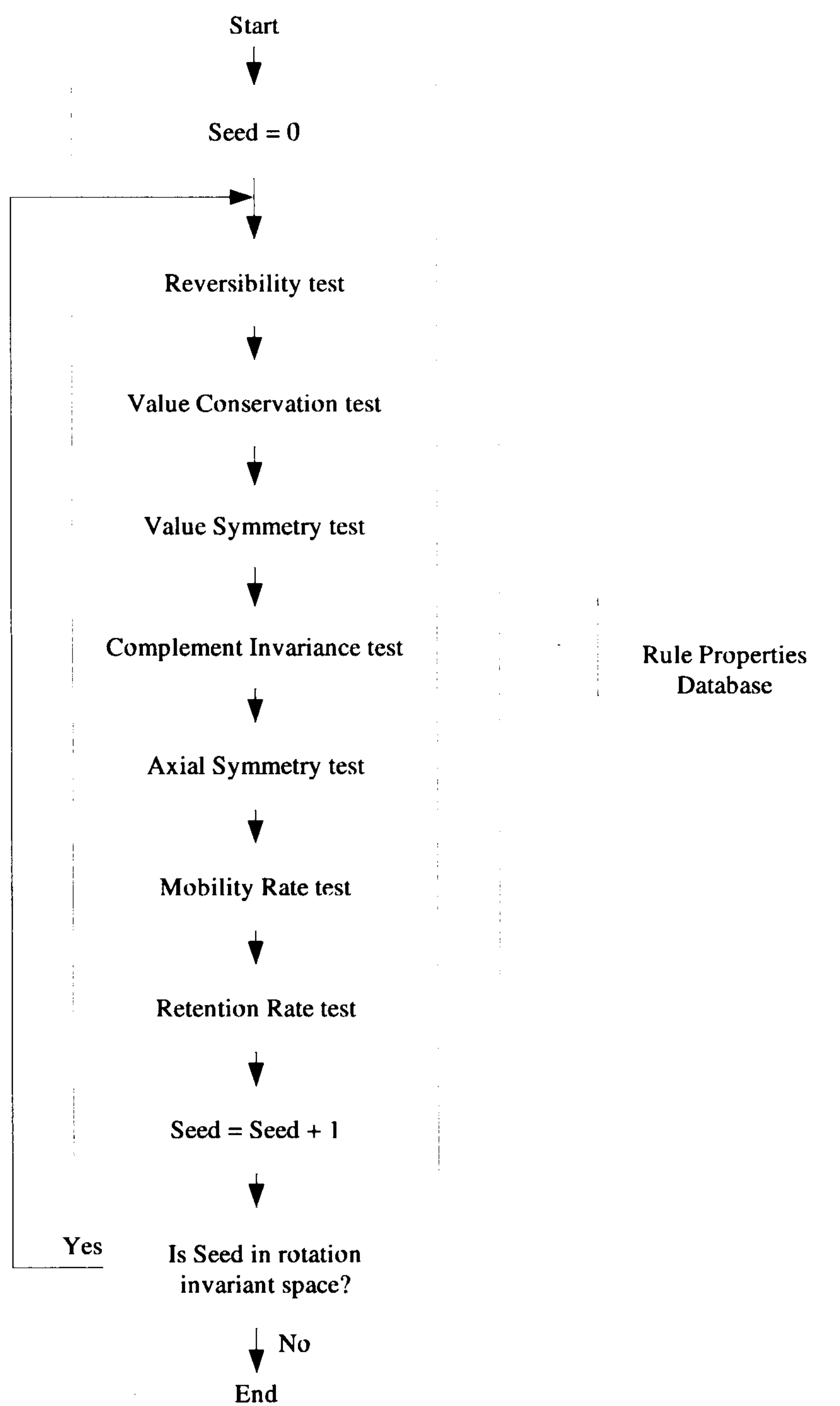


Figure 5.1: Rule analysis flowchart diagram.

Figure 5.2 shows the mix of local behaviors in the rotational invariant rule space. Only the 3.6% of the tested rule space had four of the local behaviors: reversible, value symmetry, value conservative and axial symmetry. Rules with complement invariant behavior (0.4%) were discarded; they had no match with the other behaviors included in the *second criterion.* The total number of rules k is reduced to 224 rules after applying the second criterion.

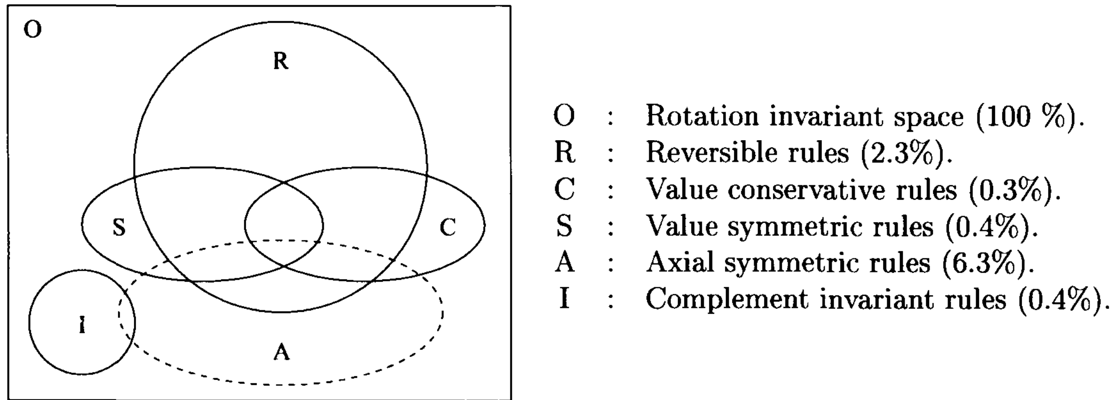


Figure 5.2: Rotation invariant rule space.

5.2.3 Global Behaviors Selection Criterion

The last criterion uses a rule score to select a single rule for the cellular automata architecture. The rule with the highest score is selected. The rule score is obtained after running a set of test programs on the cellular automaton for each of the 224 rules that passed the second criterion. The rule score is the sum of the maximum fitness of the last generation of each test program. The test programs, shown in Figure 5.3, validate the global behaviors described in section 3.5.2: data independence, program retention, and data mobility. The programs will move data from one place to another. The information on each signal shall be independent when more than one signal is involved. The complement function is used to manifest the simplest data processing form. An application named `careco.RuleTests` tries the test programs on the previously selected rules. The programs that handle one signal where tested with 4×4 cells cellular automata, while the programs that handle more than one signal use 5×5 cells cellular automata.

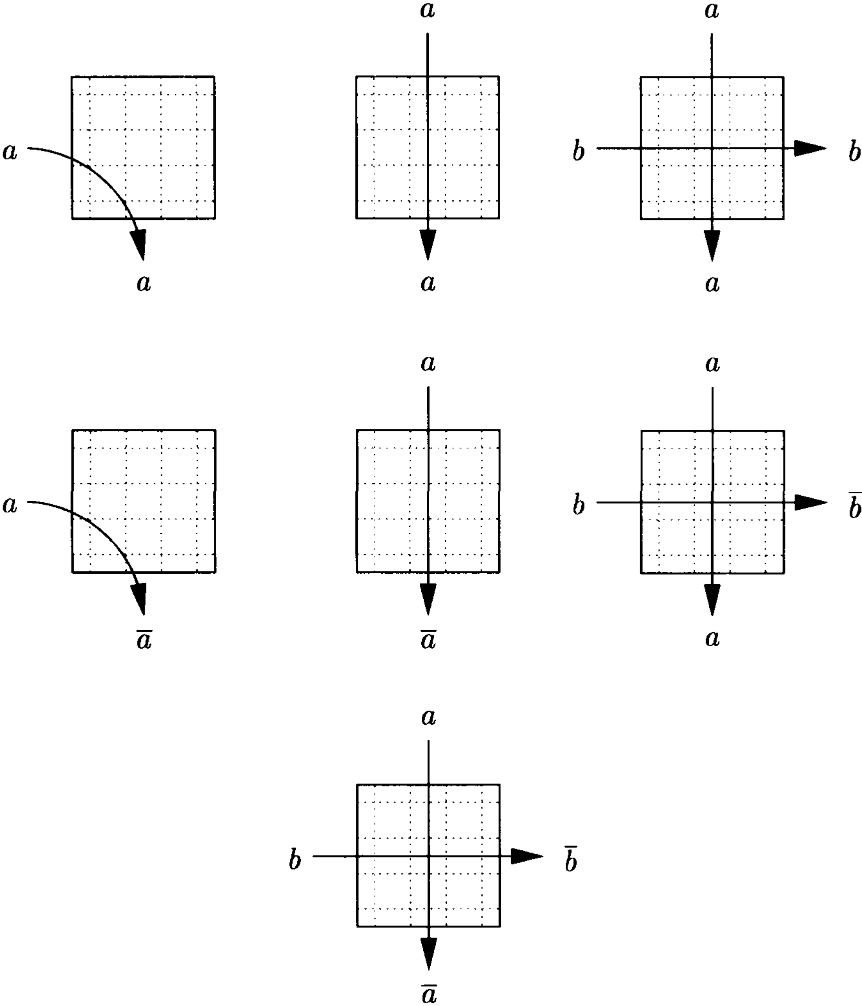


Figure 5.3: Test programs.

A genetic algorithm is run twice for every rule. The genetic algorithm reads in the rule and test program and outputs a fitness result, using the parameters listed in Table 5.4. These parameters were selected based on the experience gathered after many trials. The length of the test signals in the genetic algorithm evaluating function depends on the number of input signals, 100 samples for one input signal, and 50 samples for two input signals.

Parameter	Value
Population	100 individuals
Tournament	5 individuals
Finish criteria	20 generations
Crossover rate	90 %
Resizing rate	30 %
Mutation rate	7 %
Test signal length	100/50 samples

Table 5.4: Genetic algorithm parameters.

The genetic algorithm results are stored in the Rule Tests database see Figure 5.4. The `careco.RuleTests` application starts coding the first test program in the Target Behavior Simulator at the genetic algorithm evaluating function. Then the application makes a database query to obtain all the rules that passed the second criterion. The genetic algorithm runs twice on each rule. The genetic algorithm result are stored in the Rule Test database. Once all rules are tested, the next test program is coded in Target Behavior Simulator and the process is repeated until all test programs are tried.

Table 5.5 shows `careco.RuleTests` results analysis per local behavior. The first column refers to the local behavior property analyzed. The second column presents the average score of the rules per local behavior property. The third column shows the rules maximum score per local behavior. All tested rules have the reversible local behavior property, thus the reversible local behavior row presents the statistics of all tested rules. The table shows that in general rules with value symmetry local behavior perform below average, while value conservative rules perform above average.

Local Behaviors	Average Score	Maximum Score
Reversible	561.54	1083.45
Value Conservative	612.44	1083.45
Value Symmetric	499.03	892.85
Axial Symmetric	596.98	1083.45

Table 5.5: Average and maximum scores per local behaviors.

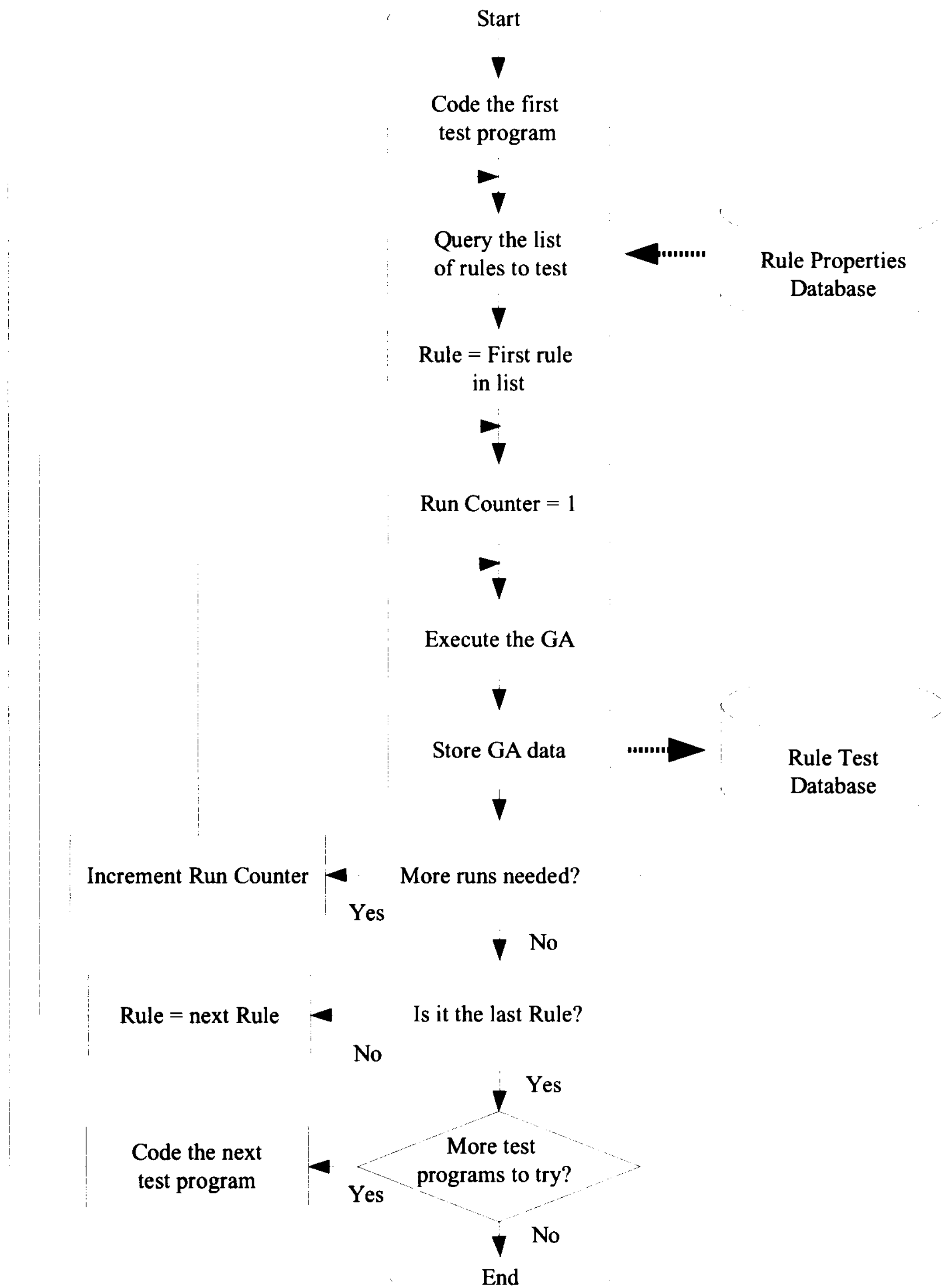


Figure 5.4: Rule test flowchart diagram.

The rules with higher scores are shown in Figure 5.6. Most of the high score rules have the value conservative local behavior property. Another common property in top score rules is the balance between mobility rate and retention rate. Both rates are almost the same. The value symmetry and complement invariance local behaviors seem to influence in a negative way the rule's performance, because almost none of the top score rules have those properties. The rules 62658 and 48336 had the highest scores. It can be proved that both rules are equivalent. One is the complement of the other, so any one can be selected. Rule 48336 was chosen. It should be notice that this rule does not have all the local behaviors used in the selection criterion.

Seed	R	VS	CI	VC	AS	MR	RR	Score
62658	√	×	×	√	√	16	16	1083.45
48336	√	×	×	√	√	16	16	1079.55
63600	√	×	×	√	×	12	20	1035.2
55536	√	×	×	√	×	12	20	1028.9
61920	√	×	×	√	×	12	20	1024.1
51896	√	×	×	√	×	16	16	1001.45
61668	√	×	×	√	×	12	20	998.85
59512	√	×	×	√	×	16	16	990.1
55984	√	×	×	√	×	16	16	989.4
59960	√	×	×	√	×	16	16	988.85
47344	√	×	×	√	√	12	20	957.1
62116	√	×	×	√	×	16	16	949.25
61666	√	×	×	√	√	12	20	941.8
57832	√	×	×	√	×	16	16	935.8
62912	√	×	×	√	×	12	20	926.3
62660	√	×	×	√	×	12	20	912.95
56528	√	×	×	√	×	12	20	910
64592	√	×	×	√	×	12	20	901.95
62672	√	√	×	√	√	12	20	892.85
49980	√	√	×	×	√	24	8	888.2

R : Reversible
VS : Value Symmetry
CI : Complement Invariant
VC : Value Conservative
AS : Axial Symmetry
MR : Mobility Rate
RR : Retention Rate

Table 5.6: Rules with the highest scores.

5.3 Rule 48336

The rule 48336 has the following local characteristics: reversible, value conservative and axial symmetry. The value symmetric local behavior does not seem to play an important role in the rule's ability to solve the test programs. The input to output rule mapping is shown in the left side of Table 5.7. A rule 62658 upside down mapping is shown on the right side of the same table. Its easy to see that one rule is the complement of the other.

Rule 48336

i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	1	1	0	0
0	1	0	0	0	0	0	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	1
0	1	1	1	0	1	1	1
1	0	0	0	0	0	1	0
1	0	0	1	0	1	1	0
1	0	1	0	0	1	0	1
1	0	1	1	1	0	1	1
1	1	0	0	0	0	1	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	1

Rule 62658

i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0
1	1	1	1	1	1	1	1
1	1	1	0	1	0	1	1
1	1	0	1	0	1	1	1
1	1	0	0	0	0	1	1
1	0	1	1	1	1	1	0
1	0	1	0	0	1	0	1
1	0	0	1	0	1	1	0
1	0	0	0	1	0	0	0
0	1	1	1	1	1	0	1
0	1	1	0	1	0	0	1
0	1	0	1	1	0	1	0
0	1	0	0	0	1	0	0
0	0	1	1	1	1	0	0
0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0

Table 5.7: Mapping of rule 48336 and rule 62658.

The simplified output functions of the rule 48336 are presented in Table 5.8. The rotation invariant property is reflected on the output functions, they all have the same structure. The axial symmetry property is also present because i_1 and i_3 can be swapped without altering the output function, both valid output expression are shown at the bottom of each output Karnaugh map.

i_3i_2 \ i_1i_0	00	01	11	10
00	0	0	0	0
01	1	0	1	1
11	1	1	1	0
10	0	0	1	1

$$o_0 = i_3\overline{i_2}i_1 + i_3i_2i_0 + i_2\overline{i_1}\,\overline{i_0} + \overline{i_3}i_2i_1$$
$$o_0 = i_3\overline{i_2}i_1 + i_2i_1i_0 + \overline{i_3}i_2\overline{i_0} + i_3i_2\overline{i_1}$$

i_3i_2 \ i_1i_0	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	1	0	1	1
10	1	1	1	0

$$o_1 = \overline{i_3}i_2i_0 + i_3i_1i_0 + i_3\overline{i_2}\,\overline{i_1} + i_3i_2\overline{i_0}$$
$$o_1 = \overline{i_3}i_2i_0 + i_3i_2i_1 + i_3\overline{i_1}\,\overline{i_0} + i_3\overline{i_2}i_0$$

i_3i_2 \ i_1i_0	00	01	11	10
00	0	1	1	0
01	0	0	1	0
11	0	1	1	1
10	0	1	0	1

$$o_2 = i_3i_1\overline{i_0} + i_2i_1i_0 + \overline{i_3}\,\overline{i_2}i_0 + i_3\overline{i_1}i_0$$
$$o_2 = i_3i_1\overline{i_0} + i_3i_2i_0 + \overline{i_2}\,\overline{i_1}i_0 + \overline{i_3}i_1i_0$$

i_3i_2 \ i_1i_0	00	01	11	10
00	0	0	1	1
01	0	1	0	1
11	0	1	1	1
10	0	0	1	0

$$o_3 = i_2\overline{i_1}i_0 + i_3i_2i_1 + \overline{i_3}i_1\overline{i_0} + \overline{i_2}i_1i_0$$
$$o_3 = i_2\overline{i_1}i_0 + i_3i_1i_0 + \overline{i_3}\,\overline{i_2}i_1 + i_2i_1\overline{i_0}$$

Table 5.8: Output Karnaugh maps for rule 48336

The best solutions to the test programs for rule 48336 are shown in Table 5.9. The first column shows the test programs. The second column has the maximum fitness obtained. The third column presents the genotype that generated the maximum fitness. The maximum possible fitness is 46 for single signal test programs, and 91 for double signal test programs. Most of the test programs performed close to the maximum fitness possible. Integer fitness values are likely to be strictly correct solutions.

Program	Fitness	Genotype
$o_{13} = i_1$	46	0110100100100010
$o_{13} = \overline{i_1}$	45.8	1100011111100001
$o_6 = \overline{i_2}$	43.6	0011000001111111
$o_6 = i_2$	46	0010111111010000
$o_9 = i_2$	46	0001000010001000
$o_9 = \overline{i_2}$	45	1001100110000001 1101100000000001
$o_{12} = i_2; o_7 = i_{17}$	91	00010010100001110100
$o_{12} = i_2; o_7 = \overline{i_{17}}$	90	10100000000001111011 10100000000001111111
$o_{12} = i_2; o_{17} = \overline{i_7}$	62.5	00001000001100000000
$o_{12} = \overline{i_2}; o_{17} = \overline{i_7}$	56.95	11001100000000010011 11111101100000010011

Table 5.9: Best solutions for rule 48336.

5.4 Program Verification

To have machine language code performing as expected under every circumstance is one of the most important drawbacks on using genetic algorithms. This issue can be treated by applying a verification stage on the genetic algorithm solutions. The verification stage consists on a strict Boolean logic conformity check of the solution program with the target behavior. The verification is performed by simulating the cellular automaton with symbolic logic. In the symbolic cellular automaton simulation, the program is represented with constants while the data is represented with symbolic values. The cell outputs are obtained evaluating the rule function using the symbolic input from neighbor cells and from the boundary. The output equations are simplified in each cellular automaton time step. The output locations of the cellular automaton are going to have Boolean expressions showing the cellular automaton actual behavior as time goes on. If this behavior is equal to the target behavior and it remains steady in time then the solution is correct in a strict logical sense. The test programs in Figure 5.3 were correctly verified for rule 48336.

5.5 Simple Programs

Simple Boolean programs helped test the rule 48336. These programs include the implementation of 2-input Boolean functions: AND, OR, XOR, NAND and XOR. The genetic algorithm had difficulties to find a solution with fixed input/output locations. It became clear that the genetic algorithm should not be restricted to find solutions with fixed input/output locations. The workaround was to include the input/output data locations in the genotype, coded as a header or preamble to the machine code. In this way the genetic algorithm found the place were the input/output data signal should be. Table 5.10 shows the obtained results. The first column has the Boolean expression tested. The second column shows the maximum fitness obtained. The next three columns tell the location of the inputs and the outputs. The last column shows the genotype that obtained the maximum fitness. The fitness values obtained are near the maximum possible value (100). Indicating a cellular automaton behavior close to the target behaviors. The NOR function was the only function with an irregular behavior under some circumstances. The state of the cellular automaton was traced manually after applying random input data to determine this irregular behavior.

Function	Fitness	a	b	c	Machine Code
$c = ab$	92.4	i_3	i_5	o_4	00011010010001000001 00010000000111101110
$c = a + b$	91.8	i_0	i_1	o_{14}	00001101000010001111 11110000000001111111
$c = a \oplus b$	93	i_5	i_3	o_{11}	11011100000101000101 11111100000000000101
$c = \overline{ab}$	91.4	i_{18}	i_{14}	o_5	10000000000100111101
$c = \overline{a + b}$	84.8	i_3	i_{19}	o_{19}	01101111100100000011 01110111101101000000 00110111111100000000 00110111111100000001

Table 5.10: Simple Boolean logic functions for 5×5 cellular automata using rule 48336.

The time delay of the solution is associated with the maximum Manhattan distances between the inputs cells and the output cell. The Manhattan distance is defined as the

distance between two points measured along axes at right angles. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), it is $|x_1 - x_2| + |y_1 - y_2|$.

A 2-bit counter program was used to test memory related global behaviors in the cellular automaton. The genetic algorithm required modifications to handle the memory behavior and the lack of inputs. The random signal generator was removed in the genetic algorithm's evaluating function. The target behavior simulator was configured as a 2-bit counter with a two-step period. The Table 5.11 shows the genetic algorithm best result.

Function	Outputs	Fitness	Genotype
2-bit counter	o_{18} o_{16}	90	00001111100110000100

Table 5.11: Simple memory function for 5 × 5 cellular automata using rule 48336.

The programming and execution sequence of the 2-bit counter solution is shown in Figure 5.5. The cellular automaton starts with all zeros initial configuration. The programming instruction is entered at time $t = 0$. The cellular automaton setup takes 6 time steps. The counting sequence starts at time $t = 6$. Each count takes 2 time steps. The sequence starts repeating itself at time $t = 15$, because cellular automaton's configuration is the same as in $t = 7$. Once a cellular automaton returns to a previous configuration state it will repeat the same sequence forever, unless the input stimulus changes.

Table 5.12 shows the output in time of the solution. A setup time of 6 steps is noticed in the second column. The third column shows the firsts execution cycles.

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$o_{18}[t]$	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	0
$o_{16}[t]$	0	0	0	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0

Table 5.12: 2-bit counter output in time.

The symbolic program verifier could not be used to verify the long-term correctness of the found solutions, because the expression simplifier wasn't able to simplify some of the expressions. Further development is needed.

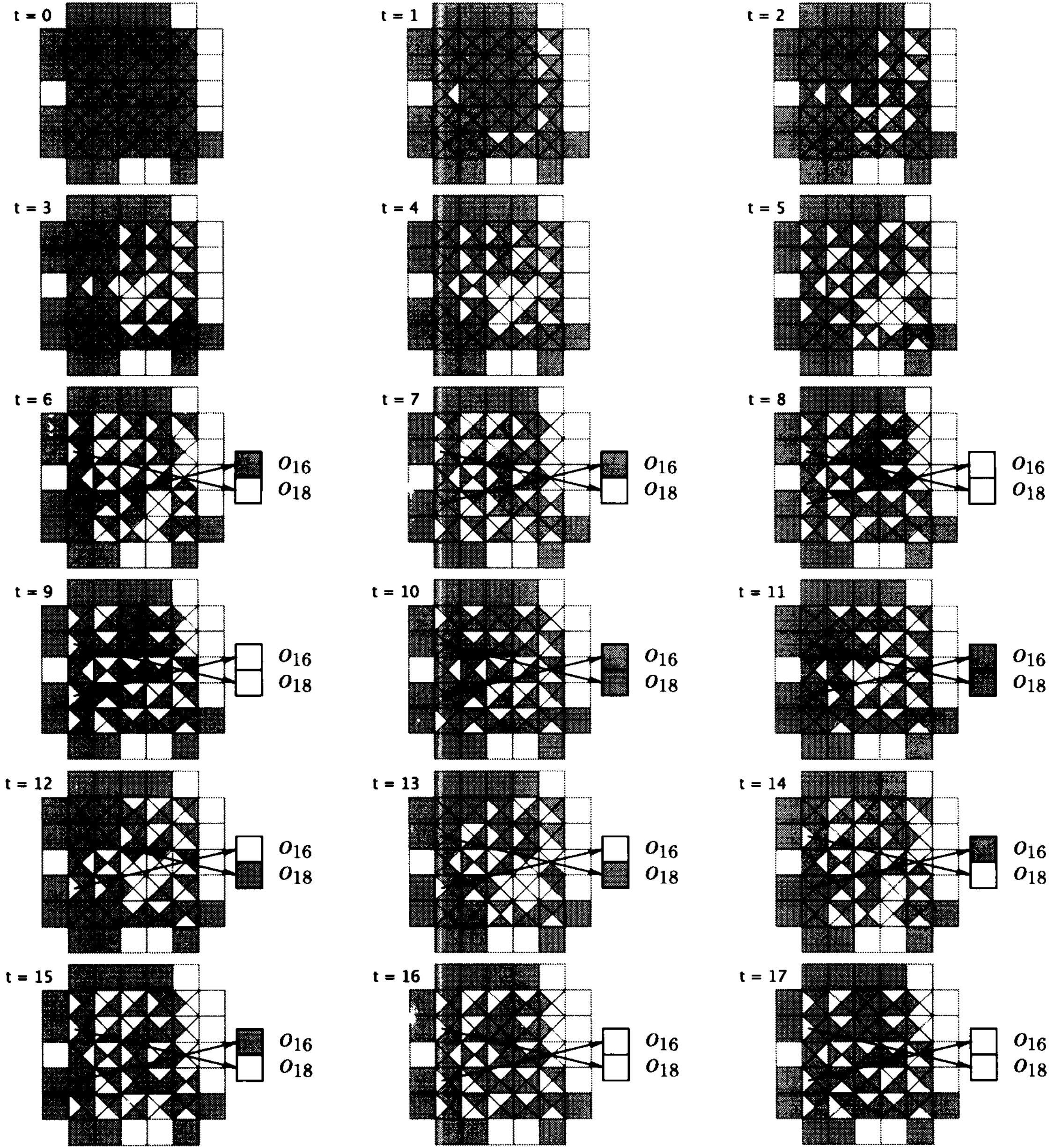


Figure 5.5: 2-bit counter programming and execution.

5.6 Results Validity

The results obtained are not the ultimate possible results. They are only a sample of what could be obtained following the proposed procedure in the time frame of the present research. Other good solutions may be there, and time should be spent searching for them. The rule test selection criteria can be changed to consider a broader rule search space. The genetic algorithms can be tuned to increase the population diversity of the solution programs.

5.7 Applications

The cellular automaton programs present in this chapter are not useful for any real life application. In the current stage of development the proposed cellular automata architecture could only be used to generate high quality random signal and test patterns. In the near future the proposed architecture could be used to simulate arbitrary cellular automata, leading to the first massively parallel Cellular Automata Machine. The application would be able to simulate, in real-time, physical models such as turbulence and optics.

Chapter 6

Conclusions and Future Work

The main contribution of this document is the new approach of how to exploit parallelism at a physical level using the mature technologies already available. The simulations results of the prototype presented in this work support the feasibility of the new proposed approach. The development scheme proposed in section 1.7 can be the common framework and guideline to do further research. Specific contributions are:

- Simulations demonstrated that global behaviors like program retention, data mobility and data independence were present in some of the rules explored under the proposed cellular automata topology.
- The value conservation local behavior plays an important role in the cellular automata ability to exhibit program retention, data mobility, and data independence global behaviors. These behaviors are essential for the proposed programming and execution methodology.
- The research results also proved that a compiler based on genetic algorithms could make cellular automata behave as desired by applying stimulus on the cellular automata boundary.
- A unique rule was able to evolve simple Boolean functions and simple memory behaviors.
- It was concluded that the cellular automata input/output data boundary locations should not be forced to specific locations. There may be no solution to some input/output placement configurations.

Future research work should focus on developing better programming methodologies and tools. Genetic algorithm should be improved in order to obtain better solutions in less time as trying to reuse previously related solutions. The genetic algorithm should be oriented to find correct solutions in a strict logical sense and solutions with optimal performance and resource allocation. A possibility is to merge the symbolic verification and the genetic algorithm evaluating function.

Many of the medium and high-level components of the computer shall be adapted to use the cellular automata computing resources in an efficient way. The operating system

will be of fundamental importance to achieve general-purpose computing architecture. The OS would have to perform many tasks generally associated to hardware. The proposed architecture is considered completely defined only with a completely developed operating system. The operating system tasks shall include:

- Data structures handle.
- Resources allocation management.
- Dataflow management.
- I/O management.

The development of the operating system should take place in parallel with the refinement of the development tools. The maturity of these tools is essential for the proposed architecture, and is the main area for further research. Research efforts should be mainly focus on this area.

Bibliography

[1] U. R. W.B. Ligon III, "Exploration of reconfigurable architectures: An empirical approach," tech. rep., Software Engineering Research Center, College of Computing Gergia Institute of Technology, 1990.

[2] M. P. V. Joao M.P. Cardoso, "Architectures and compilers to support reconfigurable computing," *ACM Crossroads*, 1999.

[3] B. N. Paul Graham, "Reconfigurable processors for high-perfomance, embedded digital signal processing," *Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1998.

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley Longman, Inc., 1989.

[5] R. D. Melanie Mitchell, James P. Crunchfield, "Evolving cellular automata with genetic algorithms: A review of recent work," *Proceedings of the First International Conference on Evolutionary Computation and Its Applications*, 1996.

[6] S. Wolfram, *A New Kind of Science.* Wolfram Media, Inc., 2002.

[7] M. Mitchell, *An Introduction to Genetic Algorithms.* MIT Press, 1998.

[8] P. Mazumder, *Genetic Algorithms for VLSI Design Layout and Test Automation.* Prentice-Hall, Inc., 1999.

[9] K. N. Hideyuki Ito, Kiyoshi Oguri, "The plastic cell architecture for dynamic reconfigurable computing," *NTT Optical Network Systems Laboratories*, 1998.

[10] J. Z. Jean-Yves Perrier, Moshe Sipper, "Toward a viable, self-reproducing universal computer," *Swiss Federal Institute of Technology*, 1996.

[11] N. Margolus, "Universal cellular automata based on the collision of soft spheres," *New Constructions in Cellular Automata*, 2002.

[12] T. Toffoli, *Cellular Automata Machines, A new environment for modeling.* MIT Press Series in Scientific Computation, 1985.