Abstract

Title of Dissertation: Self-Replicating Structures in a Cellular Automata Space

Hui-Hsien Chou, Doctor of Philosophy, 1996

Dissertation directed by: Professor James A. Reggia
Department of Computer Science

Biological experience and intuition suggest that self-replication is an inherently complex phenomenon, and early cellular automata self-replication models developed by computer scientists and mathematicians supported that view. However, since von Neumann's original work in the 1950's, the study of cellular automata models of self-replicating systems has progressively led to smaller and simpler systems. This thesis demonstrates for the first time that it is possible to create automatically self-replicating structures in cellular automata models rather than, as has been done in the past, to design them manually. These emergent self-replicating structures employ a General Purpose Self-Replicating cellular automata rule set which can support the replication of structures of different sizes and their growth from smaller to larger ones. This thesis also demonstrates that, by letting self-replicating structures carry additional information besides replication instructions, they can be used to solve computationally hard problems such as the Satisfiability (SAT) problem. It is shown that self-replicating structures can be made to carry characteristic codes and selection forces can be implemented in cellular automata space. This study opens the door to further studies that could lead to general, solution-evolvable structures and truly self-programming systems.

Self-Replicating Structures in a Cellular Automata Space

by

Hui-Hsien Chou

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1996

Advisory Committee:

Professor James A. Reggia, Chairman/Advisor
Professor Lindley Darden
Associate Professor Williams I. Gasarch
Associate Professor David M. Mount
Associate Professor Joel Saltz

Dedication

To my parents, whose love and care have been with me throughout my life.

Acknowledgements

I want to express my deepest gratitude to my advisor, Dr. Reggia. I would not have gotten this wonderful research topic nor finished the work without his enthusiastic support and advice.

I also like to acknowledge my advisory committee for their time and energy on my work: Dr. Darden, for spending a considerable amount of time correcting my dissertation; Dr. Mount, for giving me valuable information on the cellular automata metric system; Dr. Gasarch, for sharing with me his wisdom on the NP-complete problem; and Dr. Saltz, for suggestions on the potential applications of my work.

In addition, I am grateful to Ms. Nancy Lindley for her cheerful and devoted help since my first day in the department, Ms. Gwen Kaye for lending me equipment, Ms. Jodie Gray for filling out assistantship paperwork for me each year, and all the other wonderful staff in our department.

This work was supported in part by NASA Award NAGW-2805. I want to thank NASA for providing the funding of this research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cellular automata are computational systems where individual automata are connected in a grid like manner and only local interactions are allowed among these automata [von Neumann, 1966; Codd, 1968; Toffoli & Margolus, 1987]. Cellular automata are inherently massively parallel systems. *Self-replicating systems* are systems that direct their own replication by mechanisms mostly embodied in themselves. Examples of self-replicating systems in nature are living systems. A *self-replicating structure* in a cellular automata space is a pattern of active (non-quiescent) automata cells which will replicate itself based on instruction codes embedded within itself.

John von Neumann first tried to capture the idea that the fundamental information processing principles and algorithms involved in self-replication, although an essential property of living systems, were independent of the physical system which realized them [Burks, 1970; von Neumann, 1966]. He designed a computer-theoretic self-replicating machine embedded in a cellular automata space to support this idea. This opened the door for computer scientists to study the phenomenon of self-replication and other life-like behaviors from an information processing point of view, a field that is referred to today as "artificial life" [Langton, 1989; Langton, 1991].

The study of self-replication using computational models is a different approach than the synthetic way chemists use in their laboratories. Computer models permit arbitrarily large numbers of simulations with precise control over the details and parameters of individual experiments. They are open to detailed and repeated internal inspection of why certain emerging properties appear in the simulations. More importantly, as von Neumann notes, they permit one to set aside the issue of what specific chemical substances are involved in self-replication and focus on the functional information processing properties that are present. The computational study of self-replication can help to develop a better theoretical understanding of the fundamental information processing mechanisms underlying self-replication. Such study may allow computer scientists to build systems that are much more autonomous in future systems than we can build today. The choice of cellular automata as the base model for studying self-replication is especially advantageous since cellular automata have some nice properties. They are local, simple, scalable and can be easily mapped to physical systems for hardware implementation.

Since von Neumann's pioneering work, the study of self-replicating systems by others has led to progressively smaller and simpler systems [Codd, 1968; Langton, 1984; Byl, 1989; Reggia *et al.*, 1993a]. The existence of these systems raises the question of whether contemporary techniques developed by organic chemists studying autocatalytic systems [Orgel, 1992] or the many innovative manufacturing techniques currently being developed in the field of nanotechnology [Schneiker, 1989; Drexler, 1989; Hopfield *et al.*, 1988] could be used to realize self-replicating molecular structures patterned after the information processing occurring with those systems in cellular automata space. As others have indicated, creation of such self-replicating devices may prove to be critical for atomic-scale manufacturing technology [Drexler, 1989]. In addition, the existence of such simple self-replicating systems may provide theoretical support for those theories of the origins of life that postulate a prebiotic stage involving simple, self-replicating molecules [Oró *et al.*, 1990; Ponnamperuma *et al.*, 1992].

1.1 Specific goals, hypothesis and motivation

Although using computers to study self-replicating systems has long been believed to be promising, and some remarkable discoveries of self-replicating structures have been made with computer models, the border between man-made machines and true artificial living systems is still wide. Even though artificial self-replicating structures can direct their own replication, and therefore in some sense achieve life-like behavior, they were designed and implemented by people. It is of interest to be able to **automatically** generate self-replicating structures in a cellular automata space.

In addition, although self-replication phenomena in cellular automata spaces have attracted great scientific interest and are worth studying by themselves, they have not actually made any real contribution in solving problems. Therefore, it is also highly desirable to be able to see how self-replicating structures may potentially solve problems at the same time they are doing their self-replication.

This research attempts to address the following issues:

- whether it is possible to provide a general rule set for self-replication of arbitrary cellular automata structures, in contrast to previous work where different self-replicating structures generally required different, incompatible, cellular automata rule sets;
- whether it is possible for self-replicating phenomena to be made to occur spontaneously in a cellular automata space, in contrast to previous work where the first self-replicating structure always has to be introduced by a person before self-replication will start; and
- whether it is possible to make use of the self-replicating behaviors of cellular automata structures for useful computational work, in contrast to previous work, where self-replicating structures only do self-replication, but nothing more.

In the short summary below and in the following chapters, it can be seen that all of these primary research issues have been addressed and answered in the affirmative.

1.2 Summary of accomplishments

A major contribution of this work is the discovery of a *general purpose* self-replicating rule set, which supports self-replicating structures with different sizes and shapes. These variable size structures replicate and grow in the cellular automata space **at the same time**. This is a significant improvement over previous self-replication incarnations, in that each different self-replicating structure needs its own supporting cellular automata rule set to function. Since previous rule sets are generally incompatible with each other it is unlikely they could support different self-replicating structures in a cellular automata space at the same time using any one of these rule sets. The new general purpose self-replicating rule set not only supports different self-replicating structures at the same time, it even allows structures to grow or evolve into others of increased size. This kind of universality provides the foundation upon which to base the next major accomplishment.

With this general purpose self-replicating rule set, it is demonstrated in this research that it is now possible to **automatically** create self-replicating structures rather than, as has been done exclusively in the past, to design them manually. It is shown that self-replication can be an emergent property arising from numerous concurrent but local interactions of basic components in the cellular automata space. Starting from a spontaneously generated minimal self-replicating structure, it is shown that larger and larger structures can grow out of the starting one, resulting in a fast expanding, self-replicating colony with lots of variation beyond what the original minimal structure might suggest.

Another main result in this research is the application of the self-replicating structures to do something besides just replicate. It is shown that computations can be done as well as self-replication by the self-replicating structures. Specifically, it is demonstrated that in addition to their own replication, self-replicating structures can be made to direct their effort at solving a computer-theoretical hard problem, the Satisfiability (SAT) problem. Self-replicating structures can be made to carry "characteristic codes." During the course of self-replication, their characteristic codes can evolve and selection forces are based upon the fitness of these characteristic codes so that those that survive at the end of the evolution process will be answers to the SAT problem.

All the discoveries above would not be possible without using a new general purpose cellular automata simulator and an associated high level cellular automata programming language, *Trend*. These software tools are also a major contribution of this work. The new simulator and language provide easy backtracking of a cellular automata simulation in a graphical environment, special language constructs to take advantage of the rotational symmetry of the cellular automata space, a large bit depth in each cell, arbitrary neighborhood templates and data field divisions within cells. These and lots of other features are not found in other past or currently available cellular automata simulation systems.

1.3 Overview of dissertation

The rest of this dissertation is organized as follows. A short summary of previous related work on cellular automata self-replicating systems is given in Chapter 2. This is followed by a description of some preliminary work relevant to this research in Chapter 3. A general purpose cellular automata simulation environment developed for this research is introduced in Chapter 4. After that, a high level cellular automata programming language, Trend, also developed for this research, is introduced in Chapter 5.

By using the new cellular automata simulation environment and the high level cellular automata programming language Trend, an emergent self-replicating cellular automata rule set, which allows self-replicating structures to spontaneously emerge and grow in cellular automata space, is presented in Chapter 6. Following that, in Chapter 7, another major cellular automata rule set which supports self-replicating structures capable of solving the Satisfiability (SAT) problem by letting them carry characteristic code and go through a selection process in the cellular automata space, is introduced. Finally, conclusions are drawn and some future research prospects are pointed out in Chapter 8.

Chapter 2

Previous Related Work

Brief summaries of previous related work are presented in this chapter, including the definition of cellular automata, the idea of self-replicating structures in cellular automata space, and some previously available software and hardware tools for cellular automata simulations. This brief review chapter provides a general orientation about what is being studied and gives links to the main goals of the research. It also reflects the contributions made in this work, which will become apparent in the following chapters.

2.1 Cellular automata

A cellular automata model is an array of identical processing units called *cells* that are arranged and interconnected throughout space in a regular manner. Each cell represents the same abstract finite automaton (computer). Cellular automata can be one, two or multiple dimensional arrays, but generally only one and two dimensional cellular automata are studied. One dimensional cellular automata have been extensively studied in the past and generated lots of interesting results [Wolfram, 1994; Wuensche & Lesser, 1992], but we are mainly concerned with two dimensional cellular automata in this thesis since they are the basis for self-replication studies.

In two dimensional cellular automata models, space is divided into cells, each of which can be in one of n possible states. These internal states are often represented by integers starting from 0 to $n-1$, where 0 is usually used to designate the "quiescent" or "inactive" state. Non-zero values are said to be "active" states. At any moment most cells are in the quiescent state while other cells are in active states. During each instance of simulated time, each cell or component uses a set of rules called the *transition function* to determine its next state as a function of its current state and the state of its immediate neighbor cells. Which cells are considered to be immediate neighbors varies from model to model. Two popular choices are the "von Neumann neighborhood" and the "Moore neighborhood", which are named after their inventors. In the von Neumann neighborhood a cell has four immediate neighbors: north, east, south and west. Generally, a cell is also part of its own neighborhood, so there are five neighbors in the von Neumann neighborhood. The Moore neighborhood includes all neighbors in the von Neumann neighborhood, plus the four diagonal neighbors: northeast, southeast, southwest and northwest, for a total of nine neighbors. The von Neumann neighborhood and Moore neighborhood are given in Figure 2.1.

In cellular automata the state transitions of each cell are governed by the same set of rules which collectively are called the transition function. For example, suppose each cell can be in one of the two states, 0 and 1. Also suppose that the Moore neighborhood is used. Then one transition

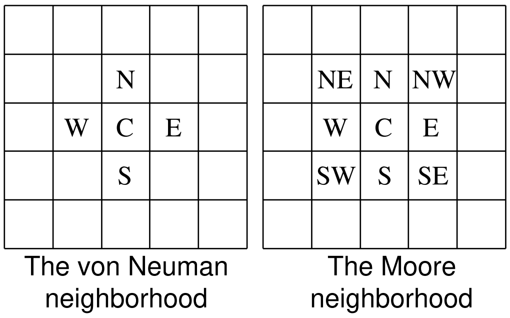


Figure 2.1: The von Neumann and Moore neighborhoods.

rule might be “If a cell is in the state 0, and of its eight neighbor cells exactly three are in state 1, then that cell should change to state 1 at the next instance of time, otherwise, it remains in state 0.” Each single rule is simple and based solely on locally-available information. However, experience has shown that the complete set of rules forming the transition function, through their application by all cells in the model simultaneously and repetitively over time, can produce very rich and at times striking behavior. For this reason, cellular automata are being increasingly used as models in physics, chemistry, biology, and other scientific fields [Gerhardt *et al.*, 1990; Preston & Duff, 1984; Toffoli & Margolus, 1987; Wolfram, 1986].

The rule used as an example in the previous paragraph is actually part of the famous Game of Life rule set developed by John Conway in the 70's [Gardner, 1970]. The second rule in this rule set is “If a cell is in the state 1, and of its eight neighbor cells more than three or less than two are in state 1, then that cell should change to state 0 at the next instance of time, otherwise, it remains in state 1.” The first rule is called the rule of birth, and the second rule is called the rule of death. Literally, a new active cell will be born if there are exactly three active neighbors around it, and an active cell will die due to overcrowding (over three active neighbors) or loneliness (less than two active neighbors). The application of the Game of Life rule set on a sample cellular automata space configuration is shown in Figure 2.2. Note that we follow the convention here and in the following figures of showing quiescent cells in state 0 as being empty or blank. The grid lines between cells will not be shown in the following figures to avoid cluttering. We can examine the content of each cell before and after each transition to verify that the rules have been correctly followed. This figure shows a staged diagonal transition of a pattern, called a “*glider*”, in the cellular automata space. This pattern is not prescribed anywhere in the rule set. Instead, it is a phenomenon arising from the collected behavior of each individual cell doing local interactions only. It is thus an *emergent property* of the cellular automata space under the Game of Life rule set.

2.2 Self-replicating models in cellular automata

John von Neumann first conceived of using cellular automata to study the logical organization of self-replicating structures or “machines” [von Neumann, 1966; Burks, 1970]. His and subsequent related work has tried to obtain a deeper understanding of the fundamental information processing principles and algorithms involved in self-replication, independent of how they might be physically realized. A self-replicating structure in cellular automata space is represented as a contiguous configuration of active cells, each of which represents a component of the “machine”. At each instance of simulated time, each cell or component uses the same rules to determine its next state

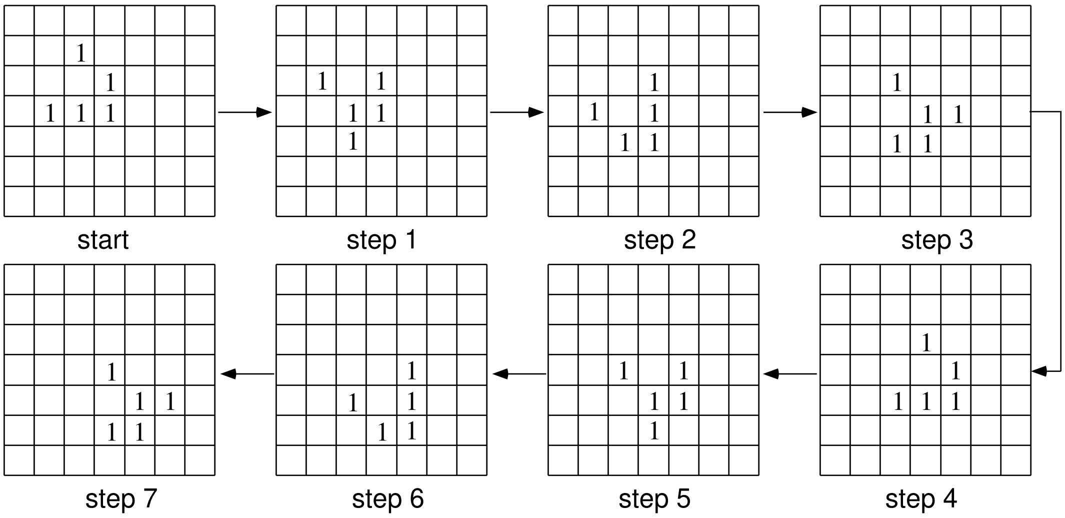


Figure 2.2: The Game of Life rule set. Successive application of the Game of Life rule set on an initial cellular automata pattern called the *glider*. This glider pattern will gradually move toward the lower right corner while each cell is plainly following the Game of Life rules.

as a function of its current state and the state of its immediate neighbor cells. Thus, any process of larger scale self-replication captured in a model like this must be an emerging behavior arising from the strictly local interactions that occur among cells. Based solely on these local interactions a self-replicating structure goes through a sequence of steps to construct a duplicate copy of itself.

John von Neumann's original self-replicating structure was a *universal constructor-computer* embedded in a two-dimensional cellular automata space that consisted of 29-state cells. It was literally a simulated digital computer that used a *construction arm* in a step-by-step fashion to construct a copy of itself from instructions on a *tape* (see Figure 2.3). Subsequently, Codd showed that if one assumes that the components or cell states meet certain symmetry requirements, von Neumann's configuration could be done in a simpler fashion using cells having only eight possible states [Codd, 1968]. Other variations of von Neumann's model have been studied to examine a number of issues, and continue to generate theoretical consideration today. However, while these structures self-replicate, they consist of at least tens of thousands of components or active cells, and have thus not actually been fully simulated computationally because of their tremendous size and complexity.

The complexity of these early cellular automata models seemed consistent with the remarkable complexity of biological self-replicating systems: they appeared to suggest that self-replication is, from an information processing perspective, an inherently complex phenomenon. However, more recently a much simpler self-replicating structure based on 8-state cells, the *sheathed loop*, was developed (see Figure 2.4a) [Langton, 1984]. The term "sheathed" here indicates that this structure is surrounded by a protective covering or sheath (X's in Figure 2.4). To create a sheathed loop the biologically-implausible requirement of universal computability used in earlier models was abandoned. To avoid certain trivial cases, sheathed loops are required to have a readily-identifiable stored *instruction sequence* that is used by the underlying transition function in two ways: as instructions that are interpreted to direct the construction of a replica, and as uninterpreted data that is copied onto the replica [Langton, 1984]. Thus, sheathed loops are truly "information repli-

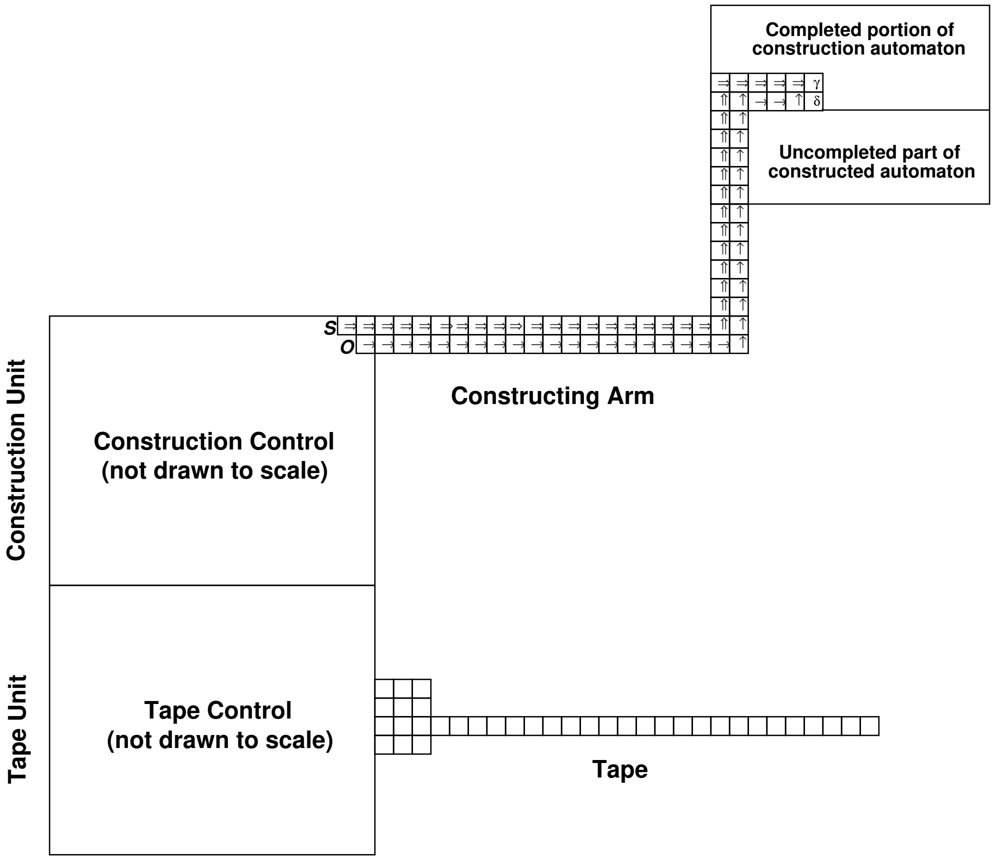


Figure 2.3: von Neumann's self-replicating machine. Schematic diagram of von Neumann's self-replicating cellular automata configuration (modified from an illustration in [Burks, 1970]). Note that this figure is not drawn to scale; the actual configuration contains tens of thousands of cells, and has not been fully simulated on a computer before.

cating systems" in the sense that this term is used by organic chemists [Orgel, 1992]. For example, in Langton's loop (Figure 2.4a) the instruction sequence "`+ + + + + + L L`" can be identified by reading off the loop content in a clockwise manner starting with the first '+' at the branching cell. This instruction sequence directs the loop to replicate another copy of itself in 151 iteration steps in the cellular automata space.

The original sheathed loop was a modified version of a device called a *periodic emitter* that was previously used as a storage element and timing device in a simplified version of von Neumann's original model [Codd, 1968]. It consists of 86 active cells as pictured in Figure 2.4a, and its transition function has 219 rules based on the von Neumann, or 5-cell neighborhood. Subsequently, two smaller self-replicating sheathed loops containing as few as 12 active cells in one case have been described (Figure 2.4b) [Byl, 1989].

These previous self-replicating structures have a drawback that they are all man-made: people designed them, in contrast to the fact that real self-replicating molecules are most likely naturally emergent. The replication of these self-replicating structures must be started with an initial *Adam* structure put in the cellular automata space by human experimenters, and the environment in which they replicate, the cellular automata space, must be free of impurities. Any tiny bit of

```
 XXXXXXXX
X0+ 0L 0LX
X XXXXXX X
X+X    X0X
X0X    X0X
X X    X0X
X+X    X0X
X0XXXXXX0XXXXX
X +0 +0 +00000X
 XXXXXXXXXXXXX
```

(a)

```
 XX
XLOX
XL+X
 X*
```

(b)

Figure 2.4: Langton and Byl's self-replicating loops. (a) Langton's self-replicating loops, SL86S8V. (b) Byl's self-replicating loops, SL12S6V.

unwanted configuration in the cellular automata space can dramatically ruin the whole replication process. Such a high vulnerability to noise is not observed in natural molecular replications.

A second drawback of these past self-replicating structures is that they have no useful computation ability. Although von Neumann's and Codd's self-replicating machines are capable of universal computation, they cannot be easily simulated on digital computers. Even if they could, their method of computation in the cellular automata space is still based on the sequential Turing Machine architecture, and is not efficient enough to do any practical computation. The self-replicating loop structures of Langton and others, although they can be simulated by computers, are incapable of doing computations. They basically just replicate themselves without doing any other useful work.

In this research, all the aforementioned drawbacks of self-replicating structures have been addressed. We will see, in Chapter 6, how self-replicating structures can be made to emerge in a randomly initialized cellular automata space, to replicate with some resistance to noise, and to gradually grow in size. We will also see, in Chapter 7, how self-replicating structures can be made to carry characteristic code, to compute solutions of a hard problem while they are replicating, and to do these computations in an efficient, parallel manner.

2.3 Software and hardware environments for cellular automata research

John von Neumann studied and solved his 29 state universal computer-constructor problem using an analytical approach, using only his own reasoning and testing a few cases by hand [Burks, 1970]. Usually, for the kind of research people have been doing with cellular automata, the following steps are taken. The investigator defines an experimental transition function for a cellular space. He or she then specifies an initial cellular automata configuration and then computes a finite fragment of the resultant cellular automata space in an attempt to produce one of the desired phenomena. This step is repeated until it succeeds or it appears not promising. In the latter case an alternate definition of the transition function may be tried. If the step succeeds, the definition is augmented further in an attempt to produce other desired phenomena, and this procedure repeats again. Without the help of a computer, these steps are tedious and almost impossible to be followed by a human.

Edgar Codd tried to reduce von Neumann's 29 state machine to 8 states, using an "interactive man-machine method" [Burks, 1970]. As von Neumann did, Codd chose as subgoals certain elementary phenomena such as signal passing and path extrusion, but Codd used a computer to assist him. There are great advantages to such simulations. The computer can make routine calculations rapidly as well as accurately, and it can assume responsibility for storing and arranging large amounts of data in a way that makes vital information immediately available to the user when he needs it. With a computer to test each possible transition function by simulation, Codd could aim directly at achieving the basic phenomena needed for his 8 state cellular automata machine. His success was due in part to the assistance he received from the computer [Burks, 1970]. Starting from Codd and other people in his time, the use of computers for cellular automata research has been extensive and natural.

Codd's computer was small in scale, having only 8K words of memory with 18 bits per word and a memory access time of 5 microseconds. It was used with only four I/O devices: a paper tape reader, paper tape puncher, online typewriter, and off-line printer. The cellular automata space was printed on paper for visualization, and saved on tapes. Commands to control the simulation were typed in using the typewriter [Codd, 1968]. Although his system was state-of-the-art in the 60's, it is not very useful compared to today's technology.

We need a cellular automata simulation system which can exploit the high speed computing power available today, and which also incorporates the mainstream graphical user interface strategy we all enjoy now. Although there were many public domain cellular automata specific simulation systems which had been developed when the work described in this dissertation began, they were usually designed for a specific model, such as the Game of Life rule set. There are very few general purpose cellular automata simulators in existence. Often a researcher has to write his or her own simulation program in order to carry out studies with a new cellular automata model. Two of the most capable general purpose cellular automata simulation systems are briefly introduced below. There are other cellular automata simulation programs which are documented online in the Internet[1].

The CAM-6 machine (abbreviated CAM in the following) is a cellular automata simulation system involving both hardware and software components [Toffoli & Margolus, 1987]. It consists of a module that plugs into a single slot of the IBM-PC or compatible models, and driving software operating under PC-DOS. The control software for CAM is written in Forth, and runs on the IBM-PC with 256K of memory. Forth is a semi-high level postfix programming language where the operators are always after their operands in the program source code, which is sometimes hard to read and understand. The Forth language adopted by CAM has been extended to contain a variety of words and constructs useful for defining cellular automata rules and for constructing, documenting, and running experiments. Source rule sets written in Forth are converted by the host computer to an internal rule table stored in the CAM hardware before the simulation starts. The simulation results can be visualized on a color monitor; each cell is represented by a colored dot. The colormap for different cell states can be specified by the user to suit the requirements of each experiment.

In CAM, up to four bit-planes are available for encoding the state of a cell; thus a cell can have up to 16 states. However, there are some restrictions on the collective use of the four bit-planes; the center cell can see only values of its own four bit planes at once. CAM has a limited set of preselected neighborhoods; there is no general mechanism to allow definition of arbitrary new neighborhoods. To avoid boundary problems, CAM space is wrapped at the edges to form a torus-

[1] `http://alife.santafe.edu/alife/topics/cas/ca-faq/soft/soft.html` as of 7/17/96.

like connected cellular automata space in all four directions. See the reference [Toffoli & Margolus, 1987] for more details about the CAM machine. The CAM machine is important because it is the first general purpose cellular automata simulation system that is widely available, it is the first low cost hardware accelerated simulator, it is the first simulator to introduce the concept of *data fields* within cells, and it is also the first popular cellular automata simulator using a high level language instead of a table to describe its rule sets. Although the capabilities of the CAM machine may seem comparably outdated, it has inspired several new cellular automata simulator designs including the one presented in this work, and is still being used by researchers around the world. However, most people do not have access to this hardware and thus its usefulness is limited.

Cellular is a system designed to model physical systems [Eckart, 1995]. It consists of the following separate program components: a programming language, *Cellang*, and associated compiler, `cellc`; a cellular automata simulator for execution, `avcam`; and a viewer for display output, `cellview`. Compiled *Cellang* programs can be run using `avcam` with input provided separately in a data file. The result of an execution can either be viewed on screen or output to a file. The output can later be viewed using the `cellview` utility.

Cellular uses an imperative programming language *Cellang* to implicitly specify deterministic cellular automata rules. Programs written in *Cellang* have two main components, a cell description and a set of statements. The cell description determines how many dimensions there are, what field(s) each cell contains, and the bit depth in each field. There are three kinds of statements in *Cellang*: **if**, **forall** and **assignment**. The *Cellular* space is also folded at the boundary cells to form a torus shape in the case of a 2-D space. This is a common practice among cellular automata simulators. A predefined variable `cell` has a special meaning: it refers to the current cell under consideration. Assignment to `cell` sets the next state value of the current cell. Neighbors can be arbitrarily referenced using a relative indexing format such as `[0,1]` for the east neighbor or `[-1,1]` for the northwest neighbor. There are no named neighborhood positions in the language. The *Cellang* language does not have language constructs to exploit the rotational symmetry of the cellular automata space, nor does it have mechanisms to prevent nonisotropic cellular automata rule set or rule conflicts. There are other interesting features of the *Cellular* system; additional information of the system can be obtained online[2].

Although the two simulation systems above are very capable in doing many general purpose cellular automata simulations, they are still inadequate to the kind of work required in this research. Specifically, we need a system which allows easy backtracking to a previous cellular automata space configuration, a high level cellular automata programming language which can exploit the rotational symmetry of the cellular automata space, a larger bit depth in each cell, etc. For these and other reasons, a new general purpose cellular automata simulation system has been developed, along with a new language for the definition of cellular automata transition functions. It is more powerful than most, if not all, of the previously available cellular automata simulation systems.

We will come back to make some comparisons of the new system with the two simulators mentioned above after the new system and its associated cellular automata programming language have been introduced in Chapters 4 and 5. We will see in those comparisons that the new system has some major advantages over the available ones, especially when doing simulations of the complex cellular automata models we will meet in Chapters 6 and 7.

---

[2] `http://www.cs.runet.edu/~dana/ca/cellular.html` as of 7/17/96.

Chapter 3

Preliminary Studies

This chapter provides a brief description of two preliminary studies relevant to my research. The first section describes work in which I participated that our group[1] did on creating and studying even simpler self-replicating loops [Reggia *et al.*, 1993a; Reggia *et al.*, 1993b]. The motivation for this research was that creating self-replicating structures of minimal complexity is an important prerequisite to studying their spontaneous emergence. I also did some random configuration experiments with these self-replicating loops. The second section describes a pilot study I did to extend the cellular automata framework to facilitate supporting movement and interaction between composite structures [Chou *et al.*, 1994]. Traditional cellular automata models are very inconvenient in modeling chemical reactions when reactants represented by multiple cells in the cellular automata space need to act and move together at once. The motivation for this research was to see if this modelling limitation could be remedied by allowing multiple cells to be considered as a whole in an extended cellular automata model.

3.1 Simplified self-replicating loops

In the work done here, it is essential for one interested in the spontaneous emergence of self-replication in cellular automata space to have self-replicating structures as small and simple as possible. The smaller and simpler the structure, the greater the chance of its self-assembly from components which are not self-replicating in the cellular automata space. That is the motivation underlying this first part of my preliminary studies.

After studying previous cellular automata models of self-replicating structures we hypothesized that adjustments to the rules controlling the interactions between components should allow elimination of the sheath (see Chapter 2), and this in turn would make simpler and smaller self-replicating structures possible. It was not obvious *a priori*, though, that complete removal of the sheath is possible. The sheath was introduced by Codd and retained in developing sheathed loops because it was believed to be essential for indicating growth direction and for discriminating right from left in a strongly rotation-symmetric space (see [Codd, 1968, p.40] and [Byl, 1989, p.296]). In fact, we discovered that having a sheath is not essential for these tasks. In the following it is demonstrated that removing the sheath leads to smaller self-replicating structures that also have simpler transition functions. For clarity, self-replicating structures are labeled in the following by their type (SL = sheathed loop, UL = unsheathed loop) followed by the number of components,

[1] In addition to myself, J. Reggia, S. Armentrout and Y. Peng worked on this problem.

```
-0+-0+-0L-0L
+          -
0          0
-          0
+          0
0          0
-          0
+          0
0          0
-          0
+          0
0-+0-+0-+0-+0000
```
(a)

```
0+-0L-0L
-      -
+      0
0      0
-      0
+      0
0      0
-+0-+0-+0000
```
(b)

```
000
0 0
L++00
```
(c)

```
00
L+00
```
(d)

```
00
L+0
```
(e)

Figure 3.1: Initial configurations of some unsheathed loops. (a) UL48S8V. (b) UL32S8V. (c) UL10S8V. (d) UL06S8V and UL06S6V (same initial configurations). (e) UL05S6V.

the rotational symmetry of the individual cell states (S=strong, W=weak), the number of possible states a cell may be in, and the neighborhood (V=von Neumann, M=Moore). For example, the sheathed loop in Figure 2.4a of Chapter 2 is labeled SL86S8V because it spans 86 active cells, has strongly-symmetric cell states with each cell assuming one of 8 possible states, and its transition function is based on the von Neumann neighborhood.

To understand how the sheath (surrounding covering of X's) can be discarded, consider the unsheathed version UL32S8V (shown in Figure 3.1b) of the original 86-component sheathed loop (shown in Figure 2.4a). The cell states and transition rules of this unsheathed loop obey the same symmetry requirements as those of the sheathed loop, and the signal sequence `+-+-+-+-+-+-L-L-` directing self-replication is also the same (read off of the loop clockwise starting at the lower right corner and omitting the "core" cells in state O). As illustrated in Figure 3.2, the instruction sequence circulates counterclockwise around the loop, with a copy passing onto the construction arm. As the elements of the instruction sequence reach the tip of the construction arm, they cause it to extend and turn periodically until a new loop is formed. A *growth cap* of X's at the tip of the construction arm enables directional growth and right-left discrimination at the growth site (seen in Figures 3.2b-d). It is this growth cap that makes elimination of the sheath possible. As shown in Figure 3.2e, after 150 units of time the original structure (on the left, its construction arm having moved to the top) has created a duplicate of itself (on the right).

The unsheathed loop UL32S8V in Figure 3.1b not only self-replicates but it also exhibits all of the other behaviors of the sheathed loop: it and its descendants continue to replicate, and when they run out of room for new replica, they retract their construction arm and erase their coded information. After several generations a single unsheathed loop has formed an expanding *colony* where actively replicating structures are found only around the periphery. Unsheathed loop UL32S8V has the same number of cell states, neighborhood relationship, instruction sequence length, rotational symmetry requirements, etc. as the original sheathed loop and it replicates in the same amount of time. However, it has only 177 rules compared to 207 for the sheathed loop, and is less than 40% of the size of the original sheathed loop (32 active cells vs. 86 active cells, respectively). The rules forming the transition function for UL32S8V are given in [Reggia *et al.*, 1992]. An example of rules for another loop UL06W8V can be seen later in Figure 3.4.

Successful removal of the sheath makes it possible to create a whole family of self-replicating unsheathed loops using 8-state cells. Examples are shown ordered in terms of progressively de-

```
OL-0L-00
-      0
+      0
0      0
-      +
+      -
0      0
-+0-+0-+0-+0
```

(a)

```
0L-00000
-      +
L      -
0      0
-      +
+      -
0      0   X
-+0-+0-+0-+0X
            X
```

(b)

```
                   X
-0+-0+-0         X0+-
+      L          X 0
0      -            +
-      0            -
+      L            0
0      -            +
-      0            -
+0-+00000-L0-L0-+0
```

(c)

```
0L-0L-00   -0+-0+-0
-      0   +      +
+      0  X0X     -
0      0   X      0
-      +          L
+      -          -
0      0          0
-+0-+0-+0-+00000-L
```

(d)

```
         0
         +
         -
         0
0000+-0+   0+-0L-0L
0      -   -      -
-      0   +      0
L      +   0      0
0      -   -      0
-      0   +      0
L      +   0      0
0-+0-+0-   -+0-+0-+0000
```

(e)

Figure 3.2: Successive states of unsheathed loop UL32S8V. (a) At t=3, the sequence of instructions has circulated 3 positions counterclockwise around the loop with a copy also entering the construction arm; (b) at t = 6, the arrival of the first + state at the end of the construction arm produces a growth cap of X's; (c) the configuration at t=80; (d) the configuration at t=115; and (e) at t=150, a duplicate of the initial loop has been formed and separated on the right; the original loop is already beginning another cycle of self-directed replication.

creasing size in Figure 3.1a-d and are summarized in the first four rows of Table 3.1. Each of these structures is implemented under exactly the same assumptions about number of cell states available (8), rotational symmetry of cell states, neighborhood, isotropic and homogeneous cellular space, etc., as sheathed loops. Given the initial states shown here, it is a straightforward but tedious and time-consuming task to create the transition rules needed for replication of each of these structures using software we developed for this purpose. The smallest unsheathed loop in this specific group using 8-state cells, UL06S8V in Figure 3.1d, is listed in row 4 of Table 3.1; it is more than an order of magnitude smaller than the original sheathed loop. Consisting of only six components and using the instruction sequence +L, it replicates in 14 units of time (column "Replication Time" in Table 3.1). Replication time is defined as the number of iterations it takes for both the replica to appear and for the original loop to revert to its initial state. This very small structure uses a total of 174 rules ("Total Rules" in Table 3.1) of which only 83 are needed to produce replication ("Replication Rules"); the remaining rules are used to detect and handle "collisions" between different growing loops in a colony, and to erase the construction arm and instruction sequence on loops during the formation of a colony.

The smallest previously described structure that persistently self-replicates, designated SL12S6V here, uses 6-state cells, has 12 components (Figure 2.4b of Chapter 2), and as indicated in Table 3.1, requires 60 state change replication rules [Byl, 1989]. We have created unsheathed loops, designated UL06S6V and UL05S6V, using 6-state cells with half as many components and requiring only 46 or 35 state change replication rules, respectively (last two rows of Table 3.1). The initial state of UL06S6V is shown in Figure 3.1d and that of UL05S6V is shown in Figure 3.1e; the complete tran-

Label	Replication Time	Total Rules	Replication Rules	State Change Rules	State Change Replication Rules	Reduced Total Rules	Reduced Replication Rules
-----	-----	-----	-----	----	-------	------	----------
UL48S8V	234	177	167	109	104	75	72
UL32S8V	150	177	166	109	104	74	71
UL10S8V	34	163	117	74	54	50	40
UL06S8V	14	174	83	91	49	66	32
UL48W8V	234	142	98	80	52	68	42
UL32W8V	151	134	98	77	52	66	42
UL10W8V	34	114	82	43	35	31	24
UL06W8V	10	101	58	44	31	33	20
SL86S8V	151	207	181	118	101	90	77
UX10W8V	44	173	103	70	36	57	25
SL12S6V	26	145	140	61	60	46	45
UL06S6V	18	115	83	64	46	30	30
UL05S6V	17	65	58	35	35	23	23

Table 3.1: Replication time and different measure on number of rules for each loop.

sition functions are given in [Reggia *et al.*, 1992]. To our knowledge, UL05S6V is the smallest and simplest self-replicating structure created under exactly the same assumptions as sheathed loops [Langton, 1984; Byl, 1989].

Cellular automata models of self-replicating structures have always assumed that the underlying two dimensional space is homogeneous (every cell is identical except for its state) and isotropic (the four directions NESW are indistinguishable). However, there has been disagreement about the desirable rotational symmetry requirements for individual cell states as represented in the transition function. The earliest cellular automata models had transition functions satisfying *weak rotational symmetry*: some cell states were directionally oriented [von Neumann, 1966]. These oriented cell states were such that they permuted among one another consistently under successive 90° rotations of the underlying two-dimensional coordinate system (rotational symmetry in these models is described mathematically in [Codd, 1968, pp.15–16]). For example, the cell state designated $\uparrow$ in von Neumann's work is oriented and thus permutes to **different** cell states $\rightarrow$, $\downarrow$, and $\leftarrow$ under successive 90° rotations; it represents **one** oriented component that can exist in four different states or orientations. However, the simplified version of von Neumann's self-replicating universal constructor-computer [Codd, 1968] and the dramatically simpler sheathed loops [Langton, 1984; Byl, 1989] are all based upon more stringent criteria called *strong rotational symmetry*. With strong rotational symmetry all cell states are viewed as being unoriented or rotationally symmetric. The transition functions for all unsheathed loops shown in Figure 3.1 also use this strong rotational symmetry requirement (indicated by S in their labels). Their eight cell states are designated

`. O # L - * X +`

where the period designates the quiescent state. All of these states are treated as being unoriented

```
00<00<LL0000
v          0
0          0
0          0
v          0
0          0
0          0
v          0
0          0
0          0
v          0
00>00>00>00^0000
```
(a)

```
00<LL000
v      0
0      0
0      0
v      0
0      0
0      0
>00>00>00000
```
(b)

```
000
0 0
L>>00
```
(c)

```
00
L>00
```
(d)

```
L0
>>0
```
(e)

Figure 3.3: Unsheathed loops based on weak rotational symmetry. (a) UL48W8V (b) UL32W8V (c) UL10W8V (d) UL06W8V (e) UL05W8V.

or rotationally symmetric by the transition function[2].

The fact that the simplest self-replicating structures developed so far have all been based on strong rotational symmetry raises the question of whether the use of unoriented cell states intrinsically leads to simpler algorithms for self-replication. Such a result would be surprising as the components of self-replicating molecules generally have distinct orientations. To examine this issue we developed a second family of self-replicating unsheathed loops, shown in Figure 3.3 whose initial state and instruction sequence are similar to those already described in Figure 3.1. However, for the structures in Figure 3.3 weak symmetry is assumed, and the last four of the eight possible cell states

$$. \ \mathtt{O} \ \# \ \mathtt{L} \ \wedge \ > \ \vee \ <$$

are treated as oriented according to the permutation (.)(O)(#)(L)($\wedge > \vee <$). In other words, the cell state $\wedge$ is considered to represent a single component that has an orientation and is thus permuted to $>$, $\vee$ and $<$ by successive 90° rotations of the coordinate system, while the remaining four cell states do not change. For example, in Figure 3.3b the states $>$, $\vee$, and $<$ appear on the lower, left and upper loop segments, respectively, to represent the instruction sequence $<<<<<<$LL. While cells in such a model have 8 possible states and are thus comparable in this sense with the above work on sheathed and unsheathed loops (Figures 2.4 and 3.1), they also can be viewed as simpler in that they have only five distinct possible components. As can be seen in Table 3.1 (rows 5–8) where the presence of oriented cell states or weak symmetry is indicated by W in the structure labels, relaxing the strong rotational symmetry requirement like this consistently led to transition functions requiring fewer rules than the corresponding strong symmetry version; this is true by any of the measures in Table 3.1. This decrease in complexity occurred in part because the directionality of the oriented cell states intrinsically permits directional growth and right-left discrimination, making even a growth cap unnecessary.

As noted earlier, the complete transition function includes a number of rules that are extraneous to the actual self-replication process (e.g., instruction sequence erasure) and many rules which

[2]Care should be taken not to confuse the rotational symmetry of a cell state as interpreted by the transition function with the rotational symmetry of the character used to represent that state. Here the character L is not rotationally symmetric, for example, but the cell state it represents is treated as such.

Rules for replication for UL06W8V:

```
.....  -> .   ....>  -> ^   ...^0  -> <   ...>.  -> 0   ...><  -> .
...v.  -> .   ...v<  -> .   ...0.  -> .   ...00  -> .   ...L<  -> .
...LL  -> .   ...#.  -> 0   ..>>0  -> .   ..<<.  -> #   ..0.0  -> .
..000  -> ^   ..0L.  -> .   ..L..  -> .   ..L>0  -> .   ..#v.  -> .
.>...  -> .   .<vv<  -> .   .<L.L  -> .   .0>..  -> .   ^...L  -> L
^.000  -> .   >...L  -> L   >.0.L  -> L   >00.L  -> L   v.0.L  -> .
<....  -> <   <...#  -> .   <..L0  -> L   0...>  -> >   0...0  -> 0
0..>0  -> <   0..L.  -> 0   0.^>0  -> <   0.<0.  -> v   0.<0^  -> v
0.0.>  -> >   0.0L.  -> 0   0.L.0  -> 0   0^0.>  -> v   0v0.v  -> >
00.0.  -> 0   00.0L  -> 0   000.>  -> >   00L.0  -> 0   0LL.0  -> 0
L..0v  -> 0   L.>.0  -> 0   L<>.0  -> 0   L<v.0  -> 0   L0^.0  -> 0
L0>..  -> 0   L#.0v  -> 0   #.<L.  -> 0
```

Rules after reduction for UL06W8V:

```
....>  -> ^   ...^0  -> <   ...>.  -> 0   ...#.  -> 0   ..<<_  -> #
..000  -> ^   ^.0__  -> .   >__.L  -> L   v.0__  -> .   <...#  -> .
<..L_  -> L   0.<0.  -> v   0.<0^  -> v   0._.>  -> >   0._>_  -> <
0^___  -> v   0v0._  -> >   000._  -> >   L____  -> 0   #____  -> 0
```

Figure 3.4: Cellular automata rule compression. Top: rules for replication of UL06W8V. Bottom: reduced rules for the same loop, UL06W8V.

simply specify that a cell state should not change. The state change rules alone are completely adequate to encode the replication process. For this reason, we believe that the number of state change rules used for one replication is the most meaningful measure of complexity of transition functions supporting self-replication. As shown in the sixth column of Table 3.1, this measure indicates that, from an information processing perspective, algorithms for self-directed replication can be relatively simple compared to what has been recognized in the past, especially when oriented components are present.

The simplicity of unsheathed loop transition functions when oriented components are used is even more striking if one permits the use of unrestricted placeholder positions in encoding their rules. I implemented a search program which takes as input a set of rules representing a transition function, such as those forming the top part of Figure 3.4, and produces as output a smaller set of *reduced rules* containing "don't care" or "*wildcard*" positions (bottom part of Figure 3.4). This program systematically combines the original rules, replacing multiple rules when possible with a single rule containing positions where any cell state is permissible (designated by the underline character '_'). Introduction of such wildcard positions is done carefully so that the new reduced rules do not contradict any of the original rules, including those that do not change a cell's state. The size of the reduced rule sets that result from applying this program to the complete original set of rules and to only the replication rules of each of the cellular automata models described above is shown in the rightmost two columns of Table 3.1. For example, with UL06W8V the single new rule

$$\texttt{L\_\_\_\_} \rightarrow \texttt{0}$$

that means "state L always changes to state O" replaces seven original replication rules, while the single rule

`>_.L → L`

indicating that L follows > around a loop replaces three original replication rules. With UL06W8V this procedure reduces the set of rules needed for one replication from 58 to 20. Thus, by capturing regularities in rules through wildcard positions, it is possible to encode the replication process for unsheathed loop UL06W8V in only 20 rules (Figure 3.4, bottom part).

This work done on simplifying loops is very useful with regard to my research. It not only gave me hints on how the basic building blocks can be constructed, but also on where to start searching for emerging self-replicating structures. Also, the experience of designing software which facilitates cellular automata modeling was quite helpful.

The self-replicating loops introduced in this section were constructed by giving both a dedicated cellular automata rule set and a well organized initial configuration. Each dedicated cellular automata rule set for a loop is incompatible with one another. Intuitively, the cellular automata rule set should be fixed and should not be changed during the course of self-replication. On the other hand, if the end goal is to discover **emergent** self-replicating structures, then the initial configuration should be allowed to vary and be randomly determined. This means that the cellular automata rule set to support emergent self-replication should be much more universal than those described above in order to encompass all possible situations in the cellular automata space. It should specify a correct next state value for any cell in the cellular space and with any possible current neighborhood configuration. In addition, the cellular automata rule set should be shared among different loops which are growing at the same time in the same cellular automata space.

If one closely examines the self-replicating loops described above it becomes evident that there are difficulties in using them directly for emergent self-replication research. The self-replicating loops we have now are all very fragile: if the initial configuration is wrong by even one cell, or if a single unwanted disturbance in the form of a nonquiescent cell adjacent to a loop occurs during the replication process, then the result can be unpredictable and can end in total destruction of the replicating process.

To examine this issue, some preliminary experiments were done where various disturbances were applied to self-replicating loops. These disturbances can basically be classified into two types. In one type a self-replicating loop was made incomplete (e.g., some part of it removed) either initially or during its replication. In the second type the self-replicating loop was not changed but some extra active cells were put into its periphery.

Figure 3.5 shows two examples of the first type of disturbance. In Figure 3.5a the tip L of UL10W8V as indicated is removed; the result is a configuration as shown which will cycle through some simple patterns but can never grow beyond its current size. In Figure 3.5b an O of UL32W8V is removed as indicated; the result is a fixed pattern of O's as shown.

Figure 3.6 shows two examples of the second type of disturbance. In Figure 3.6a # state is added in the periphery of UL32W8V. The result is a growing static area of O's which will finally take up the whole cellular automata space. In Figure 3.6b, O is added to the periphery of UL10W8V. Interestingly enough, this will not disturb the loop; instead, it will inhibit the growth of the loop. The loop will constantly send out signals through its arm but these signals will all be killed by the added O so the loop is no longer self-replicating.

In most cases examined in this fashion the self-replicating loop structure starts to deteriorate once a disturbance is introduced, and it does not take long for the whole cellular space to run

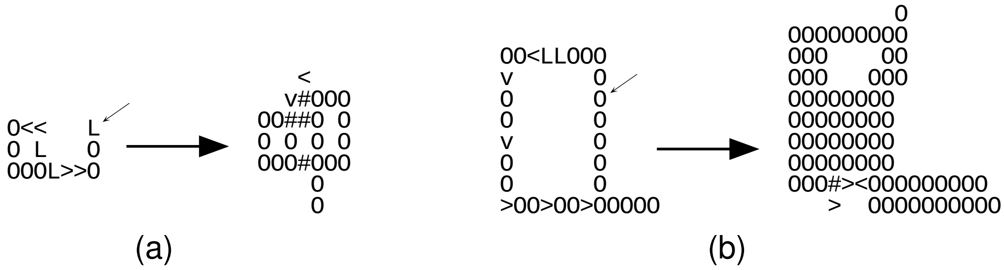


Figure 3.5: Two examples of removing an active cell from a self-replicating loop and the results. The removed active cells are indicated by arrows. (a) UL10W8V with its tip L removed during self-replication. (b) UL32W8V with one of the O's removed initially.

into an unpromising status such that no self-replication can proceed. Since random, unexpected configurations are assumed in my research, this shows that to design a robust *general purpose* rule set that allows a structure to survive at random environment configurations is hard. We will see how such a rule set is constructed in the following chapters.

3.2 An extended cellular automata model with binding and movement supports

The second direction I took during my preliminary studies was the introduction of movement and binding into basic cellular automata models. Most past work on computational models of self-replicating structures has been done with cellular automata. While such models have produced interesting results, they have been limited in terms of their biological plausibility. For example, most previous cellular automata models of self-replicating structures do not allow movement of these structures. As a first step to address this issue, I designed and implemented a software environment which permits composite structures spanning multiple cells to randomly translate and rotate. This system is described in detail elsewhere [Chou *et al.*, 1994]. The ideas involved were examined in the context of simulating a specific chemical reaction, a self-replicating deoxyhexanucleotide, C-C-G-C-G-G, which is referred to here as molecule T. This specific reaction was selected as a case study because: (1) it is the first report of autocatalytic replication of an oligonucleotide; (2) it has

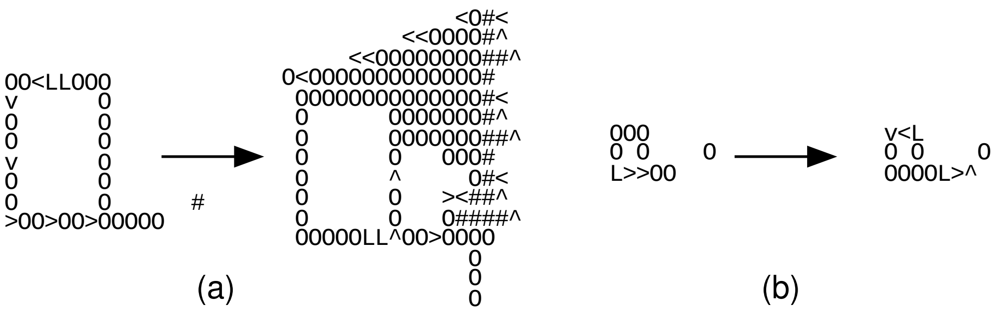


Figure 3.6: Two examples of adding active cells to the periphery of a self-replicating loop and the results. (a) a # is added close to UL32W8V; (b) an O is added close to UL10W8V.

Compound*	Symbol
d(MeO-*C*-*C*-G_p)	A
d(HO-*C*-*G*-G_p-Ph-Cl)	B
d(MeO-*C*-*C*-*G*-*C*-*G*-G_p-Ph-Cl)	T
d(MeO-*C*-*C*-G_p-C(=N-R_1)-NH-R_2)	A*
R_1-N=C=N-R_2	CDI

*Me is methyl, Ph is phenyl, *C* is cytosine, *G* is guanine, R_1 is C_2H_5 and R_2 is C_3H_6-$N(CH_3)_{2O}$

Table 3.2: Initial chemical species modeled for deoxyhexanucleotide reactions.

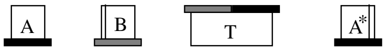
been more thoroughly studied from a kinetic point of view than other related systems; and (3) it is significant to prebiotic chemistry [von Kiedrowski, 1986].

The chemical species involved in the autocatalytic reaction that I used to demonstrate my approach are listed in Table 3.2. The right hand column specifies a symbol representing the two trideoxynucleotides (A, B) that can react to form the hexadeoxynucleotide (T). Graphically, these reactions can be represented as in Table 3.3 and Table 3.4. The labels associated with the arrows represent the probability of the reaction taking place as explained later in this section.

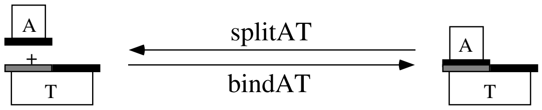
There are four *simple molecules* in the simulation: A, B, T, and A* (top row of Figure 3.7). They are treated as basic, or indivisible units in the simulation. *Composite molecules* are formed by bonding between these simple molecules. The bonding of simple molecules to form composite molecules is indicated by the reverse color of the icons. For example, the reverse color of composite molecules in the middle row of Figure 3.7 indicates that they are a single, bound structure rather than separate molecules that happen to be adjacent to each other, as shown in the last row. Since some molecules occupy multiple cells, we assign an *anchor point*, where all references of positions are made. The anchor point for each single cell molecule is simply its cell. The anchor points for multi-cell molecules are drawn in Figure 3.7. The anchor points for multi-cell molecules are always at their lower-left corner when they are sitting upright as shown in the figure.

The actual reaction occurs in a three dimensional space, but for simplicity a two dimensional space is used as has generally been done with cellular automata in the past. This two dimensional space is divided into a grid of cells. Each molecule is located at some particular cell position(s), with some molecules (e.g., T, A*T) occupying multiple cells. In displaying what is occurring during a simulation, simple small icons similar to those shown in Table 3.3 and 3.4 are used to represent molecules. Each icon in a cell indicates that a molecule of that type is in the indicated spatial location with the indicated orientation; for simplicity it is assumed that only one molecule can occupy a cell at a time. Each movement of a molecule must be discrete in a cell-to-cell manner. The simulation can be viewed on the screen, as depicted in Figure 3.8. The simulated world is much like that of a cellular automata model except for the following differences:

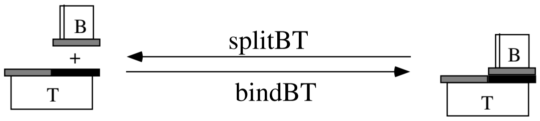
- Each cell in the cellular automata is a state machine in itself, but the cell used here is just a spatial position that a molecule can occupy.
- It's possible to have a molecule which occupies more than one cell and acts as a whole. But

1. Initial chemical species: 

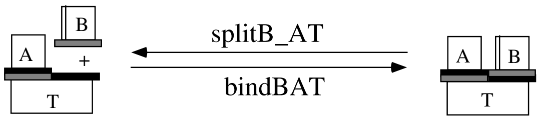
2. Monomer A. One of its sides (■) can interact by hydrogen bonding with the corresponding portion (■) of the template (T) leading to a reversible association of the complex TA:



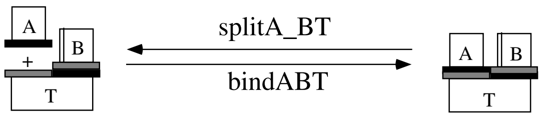
3. Monomer B. Its active side (■) can interact by hydrogen bonding with the corresponding portion (■) of the templae (T) leading to a reversible association of the complex TB.



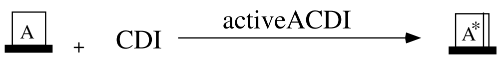
4. Monomer B can also associate reversibly with the complex TA.



5. Complex TAB can also be formed by the reaction:



6. Monomer A reacts irreversibly with 1-(3-dimethylaminopropyl)-3-ethylcarbodiimide (CDI) leading to A's activation:



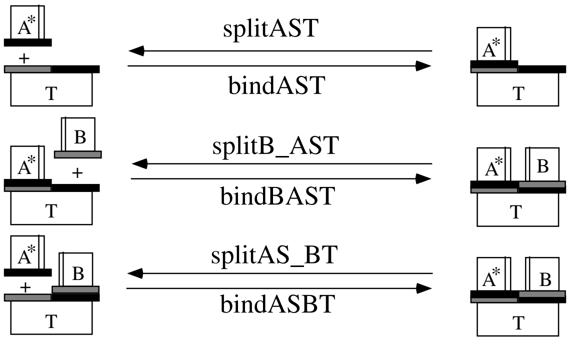
7. Complex TA can also react irreversibly with 1-(3-dimethylaminopropyl)-3-ethylcarbodiimide (CDI) leading to its activation:

Table 3.3: Deoxyhexanucleotide reactions modeled, part one.

8. The hydrolysis of CDI (irreversible reaction) decreases the rate of formation of the hexadeoxynucleotide. Its consumption from the reaction mixture is the limiting factor that causes the autocatalytic replication to end.

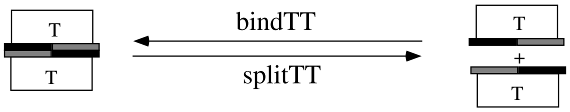
$$C_2H_5\text{-}N{=}C{=}N\text{-}C_3H_6\text{-}N(CH_3)_2 + H_2O \xrightarrow{\text{hydrolysisCDI}} C_2H_5\text{-}N{=}C(\text{-}OH)\text{-}NH\text{-}C_3H_6\text{-}N(CH_3)_2$$

9. A* behaves the same as A in terms of association with the template:

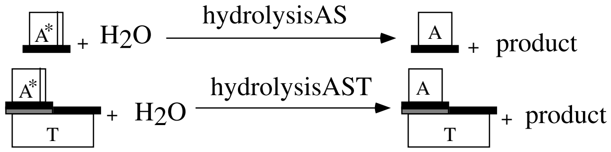


10. Complex TA*B reacts irreversibly leading to the formation of a new complex, TT:

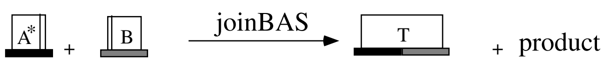
11. Complex TT dissociates reversibly forming two single- stranded templates:



12. Monomer A* and complex TA* undergo deactivation by hydrolysis:



13. Another route to the synthesis to the template is the non-directed template reaction of A* with B:



14. A side reaction producing a pyrophosphate is:

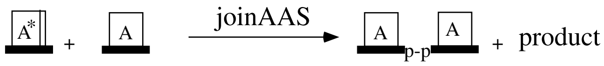


Table 3.4: Deoxyhexanucleotide reactions modeled, part two.

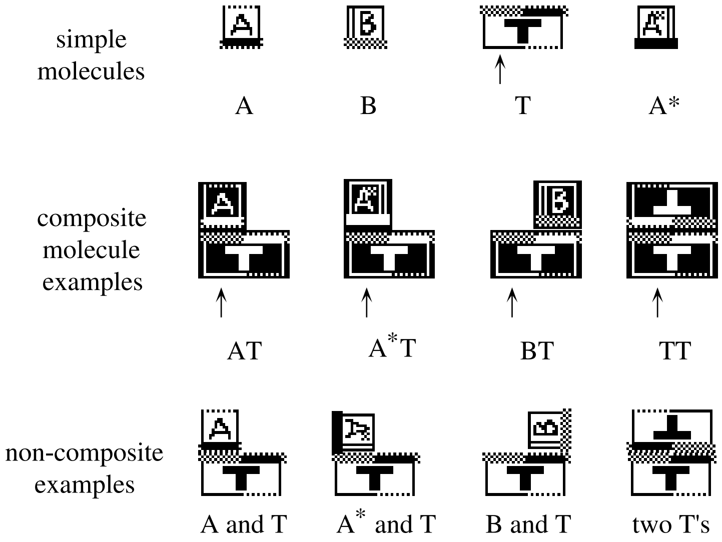


Figure 3.7: Sample deoxyhexanucleotide reaction molecules. Top row: simple molecules. Middle row: composite molecules treated as single structures as indicated by reverse coloring. Bottom row: adjacent molecules that are not bonded together and treated as independent entities. Anchor points for multi-cell molecules are indicated by arrow marks.

in cellular automata, there is no concept of "multiple cells as one".

One drawback of traditional cellular automata models is their lack of an aggregate operator: each cell acts individually according to local rules and in general a composite pattern/structure spanning multiple cells cannot act as a whole[3]. There is also no concept of bond formation between structures occupying adjacent cells. These limitations pose an immediate problem when repre-

[3]It **is** possible to simulate movement of composite structures in cellular automata models such as the *gliders* in the Game of Life [Gardner, 1970], but this movement often involves cyclic structural transformations and is not applicable to simulating chemical reactions.

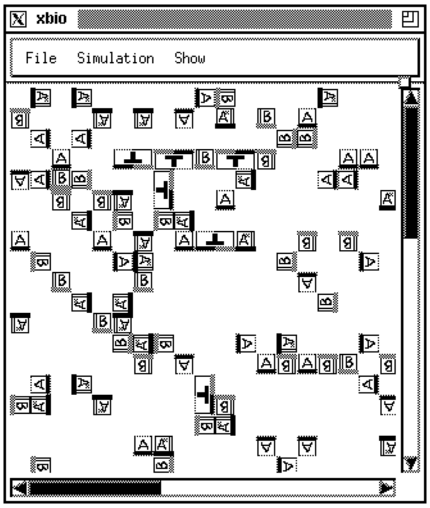


Figure 3.8: Sample simulated deoxyhexanucleotide world configuration. Each icon denotes a molecule of some kind. The figure also shows the control menu and scroll bars of the program.

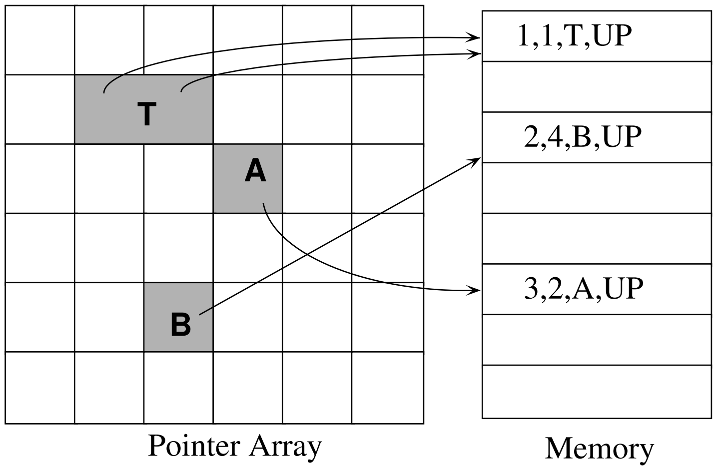


Figure 3.9: Data structures used by the deoxyhexanucleotide simulator. The computational space is represented by a two-dimensional pointer array of cells. If there is a molecule occupying a cell position, the cell will contain a pointer to the appropriate molecular data structure allocated somewhere in memory.

senting molecular structures with cellular automata. Although it is easy and straightforward to represent each simple molecule in an individual cell, it is very hard to represent composite molecules that span multiple cells and still move as a whole. For example, suppose that a nucleotide is represented as a sequence of 10 units spanning 10 cells. If one wants to represent rotation or translation of such a multi-cell structure as a unit, how is one end of the molecule to know the direction that the other end is moving given that only local operations between adjacent cells can occur? While one can imagine possible solutions to this problem within the basic cellular automata framework, they are not realistic in the chemical sense. An alternative approach that I adopted was to try to implement phenomena that involve operations on *a multicellular structure considered as a unit* (movement, binding, etc.). This was done by making changes and extensions to the cellular automata framework itself while preserving the local nature of the computations involved.

To represent the simulated space computationally, I declared a two dimensional array of pointers to *molecular data structures*. Each molecular data structure has the following information for the molecule occupying a cell: x, y coordinates, type of molecule, and orientation (Figure 3.9). The benefits of this data structure are that memory storage for molecular information must be allocated only for molecules which exist, and molecules can be moved very easily by changing the pointer values in each cell without explicitly moving the data structure in memory.

Before a simulation is started, molecules must be put into the array representing the simulated space. The initialization procedure will put as many molecules of each kind as defined by the user into the space, each with a random position and orientation. It does this by first picking a random position and orientation for each molecule. If that position has already been occupied by other molecules, it will then search from there sequentially until it finds an empty space for that molecule. The goal of the initialization procedure is to distribute the initial molecules fairly evenly.

The simulation algorithm (Figure 3.10) works by examining at each iteration all cells in the space trying to find if there are any molecules there. The order of examining cells is determined randomly for each iteration (called *epoch*), so there will not be any bias toward any direction of the simulated space. For each molecule found, the procedure in Figure 3.10 does the following:

1. See if the molecule there will change its identity. Possible changes are dissociation (split into two molecules), hydrolysis (deactivation of the molecule), and condensation (dehydration reaction of A*BT to form TT). If one of these does occur, the molecule has changed its identity; the procedure would then proceed to examine the next cell.

2. If no identity change occurs, the procedure then determines a random new orientation for the molecule. If the cells for the new orientation have not been occupied by other molecules, the rotation is made successfully[4]. But if the cells in the new location are already occupied by other molecules, a check for potential chemical reactions is made. If a reaction does occur, the molecule changes its identity, and the procedure proceeds to examine the next cell. Otherwise, the rotation cannot be made and the molecule must stay with its old orientation.

3. If no reaction has occurred, the procedure now determines a random new position for the molecule. If the cells for the new position have not been occupied by other molecules, the translation is made successfully. Otherwise, the procedure determines whether a chemical reaction occurs. If a reaction does occur, the molecule changes its identity, and the procedure proceeds to examine the next cell. If no reaction occur, the translation cannot be made and the molecule must stay in its old position.

Molecules rotate and translate constantly. In my implementation, a random number generator determines the new orientation and position of molecules. In those cases where rotation and translation do not result in collisions between molecules, the new molecular orientation and position are simply updated as described later. If the new position has been occupied by other molecules, a check for reactions is made. If two molecules collide but do not react, they will remain in position with the old orientation. As noted earlier, simply being in cells adjacent to each other with the appropriate orientation does not represent a collision and does not result in a reaction. Collisions occur only if one molecule tries to move into the other during random rotations and translations.

While it is relatively easy to describe the actions the program takes, the actual coding is quite complex since there are so many different molecules, each with potentially different sizes and geometric shapes. Every new cell that a molecule tries to occupy must be checked to determine if it can be safely used, and all of this must be done for four different orientations for each molecule.

Molecules in each cell can take on one of four possible orientations: UP, RIGHT, DOWN, LEFT. It is necessary to maintain these orientations since chemical reactions between molecules will occur only in some specific mutual positions and orientations. The other important data is the *anchor point* for each molecule. This is especially important for multi-cell molecules, where rotations and translations are related to their anchor point. The orientation and anchor point data are stored in the *molecular data structure* related to each molecule (recall Figure 3.9). Each cell in the pointer array occupied by the same molecule will contain a pointer to the same molecular data structure in memory. But only the anchor position and orientation of that molecule are stored in the molecular data structure. When dealing with the translation or rotation of multi-cell molecules, the correct pointers from cells to molecular data structures must be maintained at all times. The anchor point and orientation of each molecule are heavily referenced to determine the **correct** pointer updates. When a molecule undergoes a translation, the following things must be done:

[4]Note that single cell molecules can always be successfully rotated.

```
for each cell in the spatial array do
   if there is a molecule there then
      if it changes its identity then
         continue with the next cell
      endif
      determine a random new orientation for it
      if this new orientation has been occupied then
         if this collision causes a reaction then
            do the reaction and continue with the next cell
         else
            stay with old orientation
         endif
      else
         rotate the molecule accordingly
      endif
      determine a random new position for it
      if this new position has been occupied then
         if this collision causes a reaction then
            do the reaction and continue with the next cell
         else
            stay in old position
         endif
      else
         translate the molecule accordingly
      endif
   endif
enddo
```

Figure 3.10: An algorithmic description of the deoxyhexanucleotide simulation procedure.

- store the new anchor point coordinates in the data structure of the molecule;
- erase all pointers in cells corresponding to the old position of the molecule; and
- store the new pointers to the molecular data structure in cells corresponding to the new position of the molecule.

When a molecule rotates, the following things need to be done. Note that rotation is done using the anchor point as the center; the anchor point cannot change during rotation.

- store the new orientation into the molecular data structure;
- for multi-cell molecules, erase pointers in cells corresponding to the old orientation; and
- for multi-cell molecule, store pointers in cells corresponding to the new orientation.

The rotation and translation of molecules can only occur one step at a time. A step is a change to an adjacent orientation or position. For example, rotating from orientation RIGHT to DOWN or UP, and moving from coordinates $(22, 22)$ to $(23, 23)$ or $(21, 22)$, etc., are considered to be one step. Rotating from orientation UP to DOWN or moving from coordinates $(22, 22)$ to $(24, 22)$ are two steps.

joinASBT	joinAA	bindABT	activeACDI
bindASBT	bindAT	activeATCDI	joinBAS
bindBAST	bindBAT	bindBT	bindTAS
bindTT	hydrolysisAS	hydrolysisAST	hydrolysisCDI
splitAST	splitAS_BT	splitAT	splitA_BT
splitBT	splitB_AST	splitB_AT	splitTT

Table 3.5: Deoxyhexanucleotide reaction parameters.

Almost all actions during the simulation are determined by random numbers: new orientations, new positions, changes of identity, reactions, etc. There are various parameters which govern the decision making about the change and reaction of molecules (see Table 3.3 and 3.4). These parameters are loosely related to chemical reaction kinetic constants. To make the decision making process as efficient as possible, an integer is used to represent the probability about a particular reaction. Each time a decision needs to be made, a random number generator is called to produce an integer. If this random integer is greater than the stored parameter integer, the corresponding decision is "no"; otherwise, the decision is "yes". For example, to determine if two molecules will react, first the program checks if they collide with each other and have the prerequisite positions and mutual orientations. If they do, the program simply gets a random number to determine if the reaction occurs. If the reaction does occur, the old molecules are replaced by their reaction products in the space.

Listed in Table 3.5 are those parameters used in the program; they are the labels on the arrows in Table 3.3 and 3.4. For example, `joinASBT`[5] determines if the two molecules A* and B in the configuration A*BT will bond to form TT, `bindASBT` determines if two molecules A* and BT will react to form A*BT, and `hydrolysisAS` determines if A* will be hydrolyzed to form A.

Recall from Table 3.3 and 3.4 that molecules can change in four ways: association, dissociation, hydrolysis and condensation. Except for association, which has been described above, the outcome of the other three changes for each molecule is determined according to its type. For example, to determine whether the molecule AT will split into A and T, first a random decision is made according to the reaction parameter `SplitAT`. If dissociation does occur, two new molecular data structures are acquired (one from the old one, the other is newly allocated), and their new values are established according to the original orientation of the molecule AT.

The simulation program I developed was subsequently used by others to conduct a battery of simulations [Navarro-González *et al.*, 1994]. As shown in Figure 3.11, it produced data reminiscent of an actual chemical experiment reported in the literature [von Kiedrowski, 1986]. In Figure 3.11a, the time-variation of molecule T's is shown. Comparing that with the actual chemical reaction curves in Figure 3.11b, the similarity is evident. It is the first successful use of a modified cellular automata environment to simulate a self-replicating oligonucleotide. With this technique the oligonucleotide molecules are represented as active cells embedded in a two-dimensional array of

[5] We use "`S`" in place of "*" in the program code since "*" is inconvenient in the programming language C.

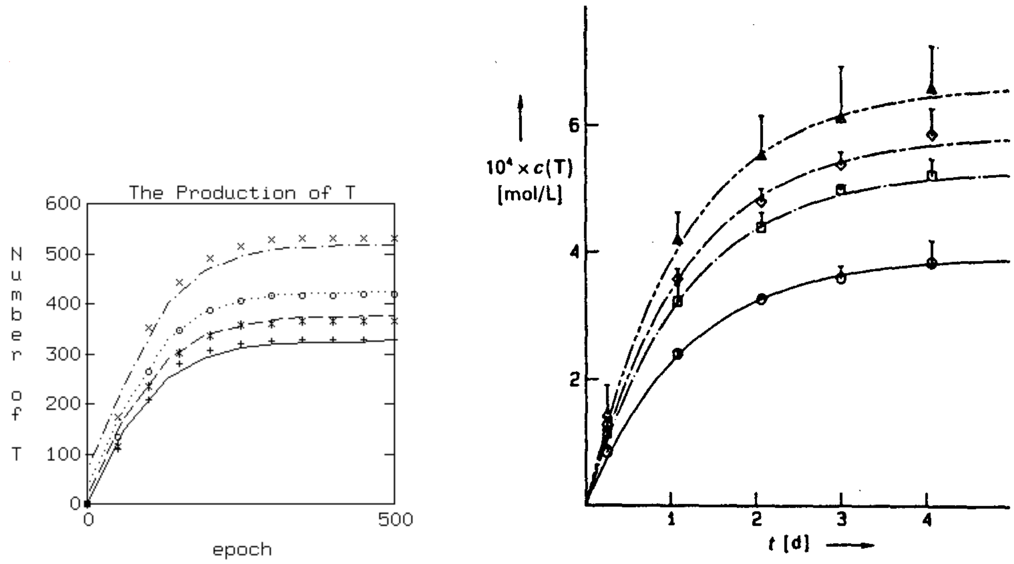


Figure 3.11: The time-variations of molecules T in deoxyhexanucleotide reactions. Left, simulation results. Right, figure from actual chemical reactions [von Kiedrowski, 1986]. Dr. Rafael Navarro-González and Miss Jayoung Wu provided the simulation data for this illustration.

inactive cells. Random movements and probability-governed chemical reactions occurring in a cellular space can effectively simulate the experimental behavior observed in self-directed replication of oligonucleotides.

In this model the multi-cell translation and rotation problems have been solved by making some changes to the basic cellular automata framework. This experience provided helpful guidance that led to efficient implementation of a general purpose cellular automata simulator as described in the next chapter.

Chapter 4

A General Purpose Cellular Automata Simulator

Introduced in this chapter is a general purpose cellular automata simulation program that was created to support the research described in subsequent chapters. It was used for the development, experiment and simulation of all cellular automata rule sets presented in this work.

The simulator is created because there is no other cellular automata simulation software available which can provide the requirements for the research conducted in this work. Specifically, we need a cellular automata simulator which provides a high level language for rule set definitions, a large number of allowable cell states (2^{64}), a mechanism for easy backtracking of simulation steps, support for data fields within cells, and an integrated, easy to use graphical user interface. When this research was starting, none of the available cellular automata simulation software, both in the public domain and through commercial channels, can provide an adequate set of features as described above to support the research. Therefore, although development of a powerful cellular automata simulator was not intended at the beginning, it became a very important part of this work.

Because nothing available came close enough to the specification of the new simulator which could have been used as a starting point, this simulator was built completely from scratch. A basic framework of the simulator was developed first which allowed at least the beginning of simulations for some simple cellular automata rule sets. Then, a large effort went into the development of the high level cellular automata programming language and its compiler. Parallel to the compiler development was the development of an evaluation module for the virtual machine code generated by the compiler. This was used for the actual execution of the cellular automata rules during simulations. The language was gradually enhanced to contain a complete set of powerful high level language constructs specifically designed for cellular automata. Later, more features were added into the visualization and control modules of the simulation, which further extended the usefulness of the simulator and made it even easier to use.

Although the new simulator described in this chapter is developed for the study of self-replicating structures in a cellular automata space, it is by no means limited to just that application. It is useful for a wide variety of simulation needs in the cellular automata research domain. Because of its high level, general purpose programming language, and its flexibility in the neighborhood definition and data field allocation, the simulator can be used for virtually all one or two dimensional cellular automata experiments. Even better, it could also be used in other similar domains such as neural network research, where autonomous entities are connected together in a fixed pattern to do cooperative computations.

The simulator has a built-in compiler for the high level cellular automata programming language, *Trend*, which can describe cellular automata rules in a highly readable and logical format. In addition, the simulator uses a state-of-the-art graphical user interface to provide easy operations of its functions. It also includes a variety of display options providing clear and instant feedback about the simulation status, a built-in text editor for entering and correcting Trend rules on-the-fly, and a sophisticated backtracking mechanism which actually encourages users to experiment with different cellular automata rules. The backtracking mechanism guarantees safe return to a previous cellular automata configuration at any time if needed.

The Trend cellular automata programming language, its compiler and the low level virtual machine code generated by the compiler will be the main topic of the next chapter. In this chapter, we will see how the simulator works. First, an example is given of how to use the simulator to experiment with cellular automata models. Next, a brief description of the various simulator functions and components is given. Finally, the data structure of the cellular automata space the simulator is based on, the organization of the simulation program, its internal program modules and their interactions, are described.

4.1 Using the simulator

Although all the simulator controls will be described in the following sections, it may be clearer to the reader if a typical session using the simulator is first demonstrated to conduct the cellular automata simulation described earlier. The capabilities, and generally how users can use this simulator to implement a cellular automata model, are also summarized in this section.

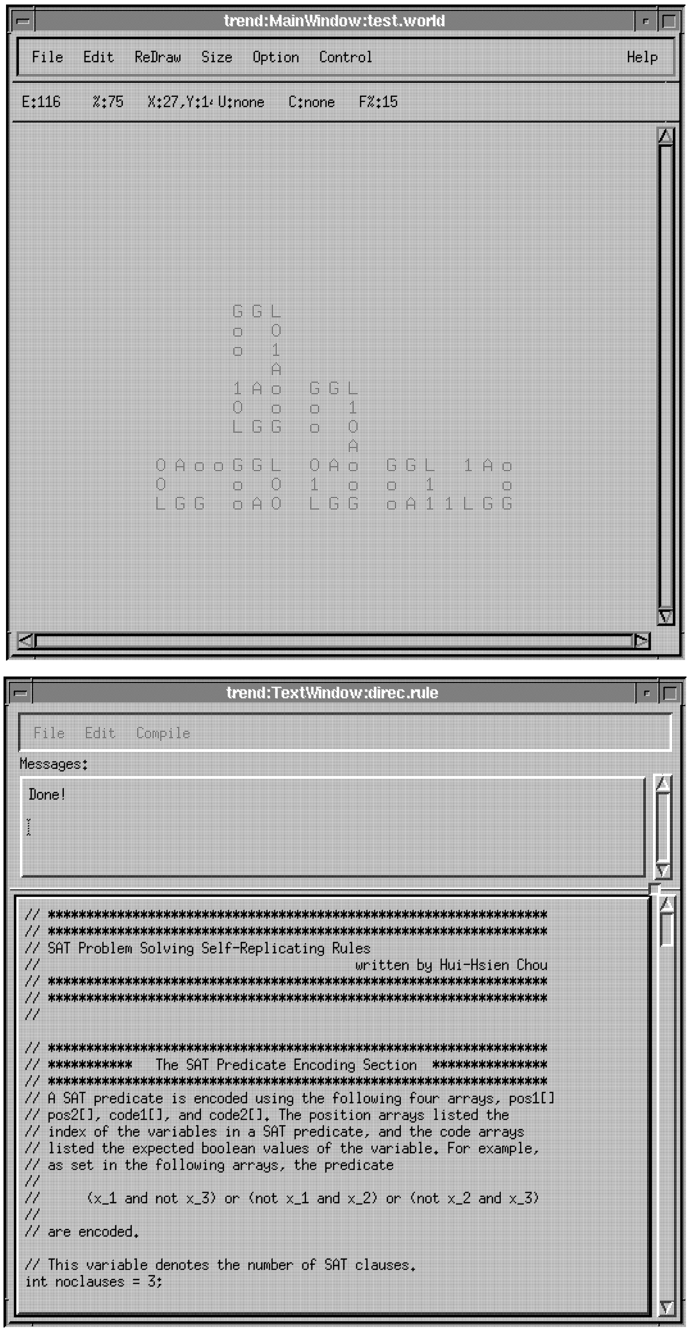
4.1.1 A typical session using the simulator

Usually the user starts up the simulator, then tries to provide cellular automata template information either by loading a previously designed one from a template file, or by designing a new one using the template design facility of the simulator. The template information includes the neighborhood positions, data fields and their sizes, symbols used to denote states, etc. When the simulator gets the template information it needs, it will present the user with two empty windows, similar to the main window and the text window shown in Figure 4.1, but without active cellular automata cells and loaded rules which is visible in the windows of Figure 4.1.

The user can load a previously designed rule program into the text window of the simulator, or the user can start inputting new rules into the text window. The user also needs to provide an initial cellular automata configuration in the main window. This can be done by either hand-inputting values into individual cells of the main window or by loading a previously design configuration from a file. The user can also load in a configuration file and then modify it for use. If the default cellular automata space size is not adequate, the user can also use the "Size" menu item to modify it.

Once the initial cellular automata configuration and the rule set for simulation have been prepared, the user can start the simulation by using either the right arrow key or the up arrow key on the keyboard. If this is an initial development of new rules such that the rule set can have some problems, the user can turn on the tracing mechanism by selecting "Trace simulation steps" in the "Option" menu.

If everything goes well the simulator will compute the next configuration from the current cellular automata configuration according to the rules in the text window. If there is any errors the



(a) The main window

(b) The text window

Figure 4.1: The main window and the text window of the simulator.

simulator will stop and report the problem. The user will have to find out what is wrong and correct the problem before he or she can continue the simulation. The user can use the backtracking buttons to go back in time to see if the problem is caused by a previous mistake. Going back and forth in simulation is very common when designing new rules, and is a major advantage this simulator provides to the user. Typical main window and text window contents when using the simulator are shown in Figure 4.1.

During the course of a simulation the user can do lots of things. He or she can stop the simulation for examination at any time. The user can choose different display speeds from the "ReDraw" menu item. The cellular automata space can be displayed using pixels for the user to see the big picture, or the viewing area can be scrolled around the simulated cellular automata space for the user to examine different parts of the space in detail. Some of the possibilities are presented in Figure 4.2.

The user can continue the simulation for as long as he or she sees fit. At the end of the simulation the user can either save the final cellular automata configuration to a file for further analysis, or he can export the configuration to an Encapsulated Postscript (EPSF) file for printing purposes.

Those above are just some typical operation steps of the simulator. The simulator is fully interactive and all its commands can be used in any order. Therefore, there are plenty of different approaches the simulator can be used, which are limited only by the imagination of the user.

4.1.2 Capabilities of the simulator

One of the major benefits of the simulator is that it allows an arbitrary number of neighbors to be defined in arbitrary positions within an eleven by eleven cellular automata region centered around the target cell. This is in direct contrast to most previous cellular automata simulation systems, which generally provided only a limited number of preset neighborhood templates for the user to choose from.

The simulator imposes no *a priori* limit on the size of the cellular automata space which can actually be simulated. This is determined by the actual computer memory size and CPU power. Up to 64 bits can be allocated for each cell of the cellular automata space, which can then be arbitrarily subdivided into different data fields for various applications.

Another major benefit of the simulator is that any neighbor or field name defined for a particular cellular automata model will automatically become one of the reserve words in the Trend cellular automata programming language, and can then be used in many of the powerful cellular automata specific language constructs provided in the language.

The simulator is very flexible in rule set definition and is designed to facilitate development of new cellular automata models. A user can start with only a very limited number of cellular automata rules, and work toward augmenting and perfecting the rule set while the simulation is going. When more rules are needed to lead the present cellular automata configuration to the next configuration desired by the user, they can simply be added to the current rule set. The backtracking mechanism of the simulator allows multiple levels of undoing and redoing, which gives 100% control to the user on how the cellular automata model or rule set should be modified and/or corrected.

There is no limit on how big the cellular automata rule set can be, either. The one-pass compiler makes virtually no difference in compiling speeds between rule sets of different sizes. Evaluation of the compiler generated code is also very efficient. Because of the way the cellular automata rules

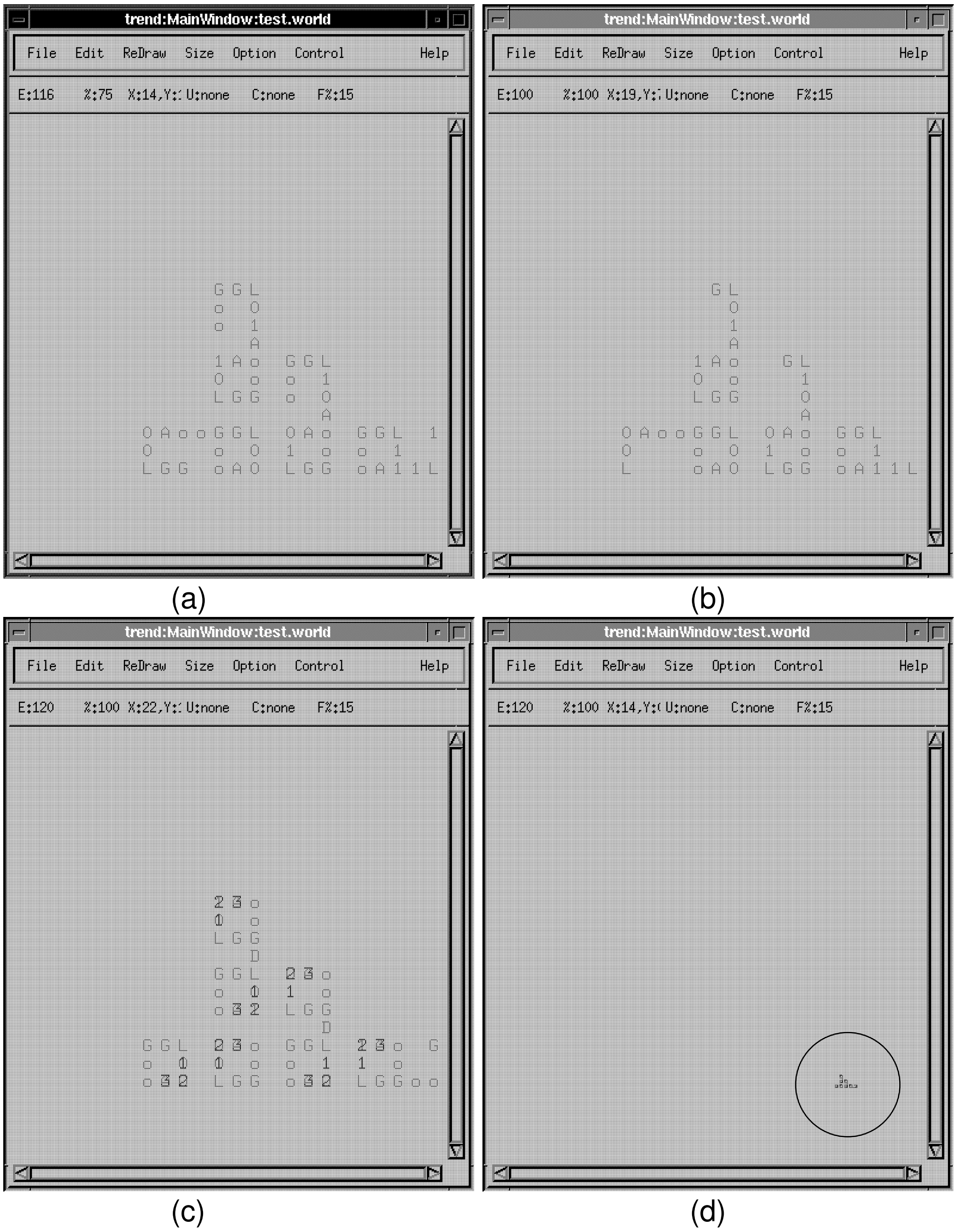


Figure 4.2: Operation examples of the simulator window. The main window of Figure 4.1 can be resized, as shown in part (a). If the backtracking mechanism has been turned on, a simulation can also go backward as shown in part (b). The "E:100" label shows the current epoch number, which is smaller than in part (a). The simulation can go forward again and some data fields can also be turned on for displaying, as shown in part (c). In addition, the whole cellular automata space can be displayed using pixels instead of characters, which allows the user to see a bigger picture of the cellular automata space, as shown within the circled region of part (d).

are usually composed, the evaluation speed of the compiled code does not necessarily relate directly to the size of the cellular automata source rules.

4.2 Simulator functions and components

A concise but otherwise thorough description of the various features of the simulator is provided in this section. First we will see how the simulator is initialized with some suitably defined template information for the cellular automata model being used. Controls and features of the two major windows of the simulator, the main window and the text window, will be described in details next.

4.2.1 Initialization

Upon starting up, the simulator needs to know which kind of neighborhood template the user is using for a particular cellular automata model. Since the simulator uses a high level programming language to describe the cellular automata rules, it also needs to have names associated with each neighbor position in the neighborhood template. In addition, since a cell can now hold different fields, it is also necessary to assign names to each field. The user may assign different symbols and colors to some fields for displaying them on the screen. The user must also designate if the cellular automata neighborhood template is symmetric and if some field states have weak rotational symmetry (see Chapter 3). All information like this for a particular cellular automata model is called its *template information* and is stored in a *template file*.

When the simulator is starting up it asks the user to provide this template information, either from a previously saved template file or by designing a new one. The user can also choose to load in an old template file and modify it for use instead of always having to start from scratch. All three choices are presented in the "Template Query" dialog window as shown in Figure 4.3. In the following two subsections we will see how different approaches work.

Loading a predefined template

If the user chooses to read in a predefined template file, a "File Selection" dialog window will pop up, as shown in Figure 4.4. The user can simply select a template file by clicking on one of the filenames displayed in the dialog, which are all files with the designated filename extension `*.tmpl` for template files.

When a template file has been selected the simulator will load it in, and display the two major windows, the main window and the text window, which will be described shortly.

Designing a new template

If the user chooses to design a new template, or if he or she chooses to modify an existing template design, the "Template Design" dialog window will appear, as shown in Figure 4.5. In the case of modifying an existing template, the "File Selection" dialog mentioned above will appear first to allow the user to choose a template file to work on.

The "Template Design" dialog window consists of two major parts, dealing with neighbor information and field information separately.

In the left part of the "Template Design" window is a set of choices regarding the neighborhood template of the cellular automata model being designed. First the user will have to determine if

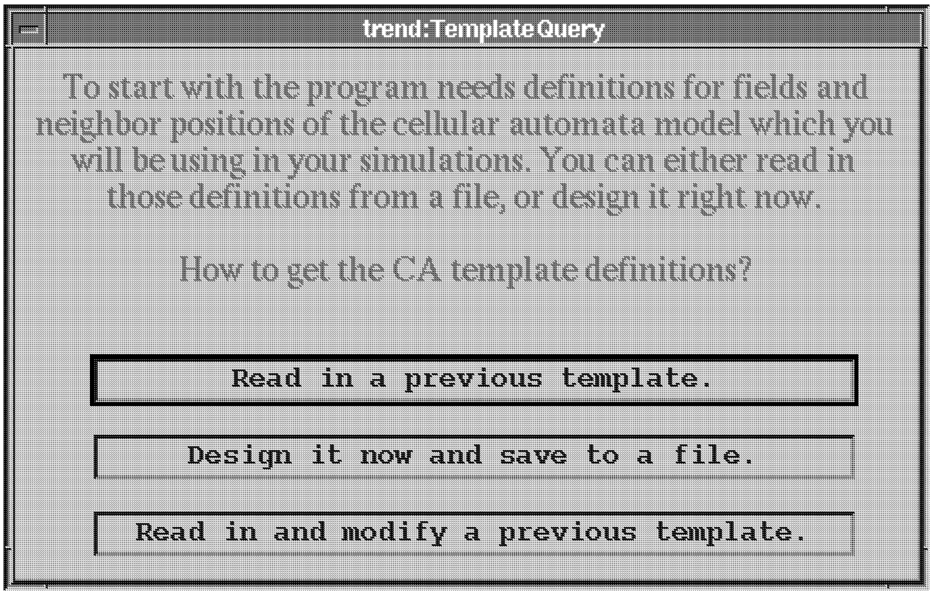


Figure 4.3: The template query window. The simulator asks users to provide a template information definition by giving three choices. Users can choose to read in a predefined template file by clicking the first (topmost) button, design a new template on the fly by clicking the second (middle) button, or read in an old file and modify it by clicking the third (bottommost) button.

the neighborhood template is rotationally symmetric, i.e., if cellular automata rules defined for this model can be applied after rotation or not. For example, if the neighborhood template is rotationally symmetric, the following rule can set a cell to 1 if any of its north, east, south or west neighbor cells is 1.

rot if `(north:cell==1) cell=1;`

This rule tests the north neighbor of a cell to determine if the assignment should be made. If the neighborhood template is rotationally symmetric and the *rot* prefix is given, this rule will be automatically rotated by the compiler to test against the other three neighbors east, south and west, which are all rotationally symmetric positions of the north in the template. If the neighborhood template is not symmetric this rule cannot be rotationally applied and only a value of 1 in the north neighbor cell can trigger the assignment statement. Actually, if the neighborhood template is not rotationally symmetric the reserve word *rot* cannot be used in the cellular automata rule at all. The compiler will check the symmetry status of the neighborhood template and deny usage of *rot* if it is not rotationally symmetric. See Section 5.10 at page 72 about the *rotated if* statement for details.

Following the symmetry choice in the "Template Design" window is a matrix of cells for neighborhood position definition. The center cell, marked by the letter 'C', is always selected since it is required in the cellular automata fundamental definition to include the center cell into the neighborhood template. It will be the "current cell" or cell of focus for any neighborhood template. The user can define additional neighbors by clicking on the other cells. If a neighbor is defined it will have an 'X' mark on it and its name is shown in the "Neighbor Name" area. If the user deletes the name of a neighbor, in this area, that neighbor will be deleted from the template and its 'X' mark

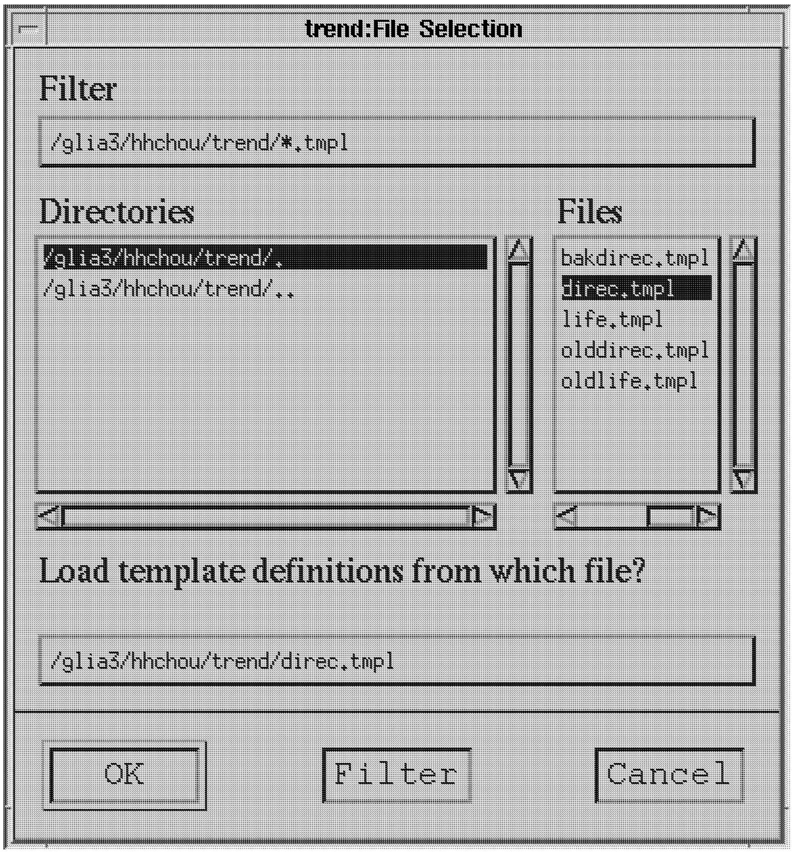


Figure 4.4: The file selection dialog window. The left box shows the directories accessible from the current directory. The right box shows files which conform to the filename extension pattern as given in the filter area. Below the two boxes is the filename area, which displays the filename which is being selected. Down below there are three buttons OK, Filter and Cancel, which can direct the simulator to either load the file, re-scan the current directory, or cancel file loading.

will be removed. The user cannot remove the center cell from the template due to the requirement stated above.

All neighbor names defined in a template will become *reserve words* in the cellular automata language *Trend*, i.e., they will be associated with neighbor positions but nothing else when they are used in the rules.

If rotational symmetry is chosen, neighbor positions must also be symmetric, or cellular automata rules will be undefined when rotated. In the previous rule example, the east, south and west neighbors must also be defined in the template for the rule to be rotatable. The simulator will check this property and will report errors if rotational symmetry is chosen but the neighborhood template itself is not symmetric.

In the right part of the "Template Design" window is a set of choices regarding field definition of the cellular automata model. In the top is a set of four "Bit Depth Reference" choices. "Bit Depth Reference" gives visual advice on how many bits has been allocated to fields and how many free bits are still available within the designated bit depth. The user can use this to budget bit allocations to different fields. The "Bit Depth Reference" selection has no real effect on the actual

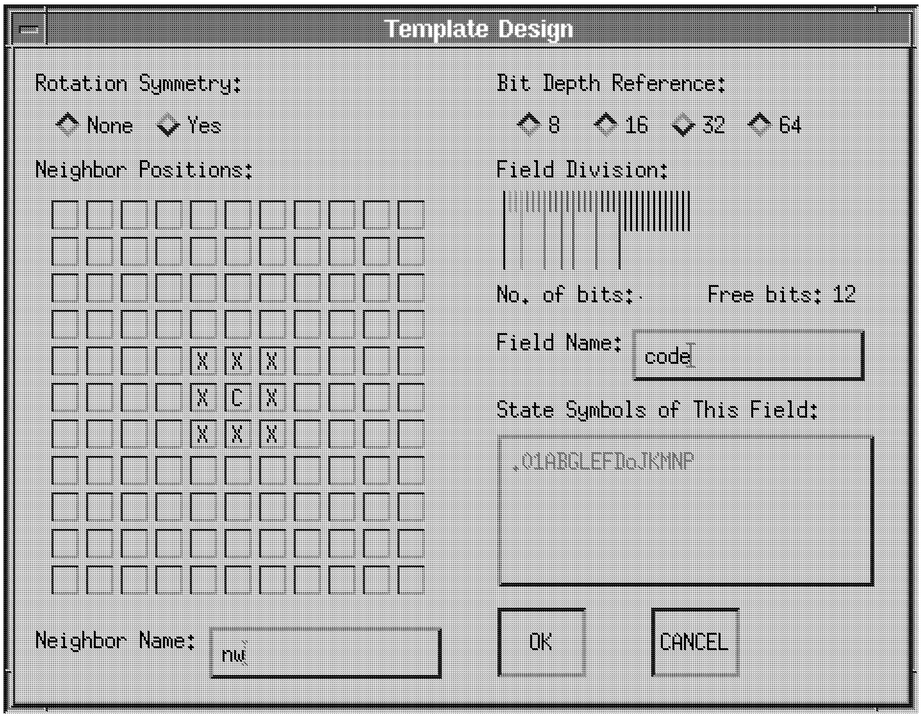


Figure 4.5: The template design window. The left portion of the window is used for neighbor information definitions. The right portion is for field information definitions in a cell. Fields are named bit groups which can be used to store data. See the text for details. A Moore neighborhood is illustrated here.

number of bits used for a cell in a cellular automata model. The simulator uses one byte as the basic unit for cell storage allocation and allocates only as many bytes as needed to represent all fields in a cell. For example, if the user defines four fields, each using five bits, the actual allocated bits for a cell will be 24, or three bytes. The smaller the bit depth, the less storage space is required, and the faster the simulation speed will be. Therefore, a user should never allocate more bits than necessary to a field to prevent wasting storage space and slowing down the simulation.

Below the bit depth reference choices is the "Field Division" area, where the user clicks in to define new data fields. The two indicators "No. of bits" and "Free bits" give the user an idea of the size of data fields he or she allocates. Once a new field is defined its name will be displayed in the "Field Name" area. Deleting the name for a field will effectively remove that field from the template. If color is enabled in a simulation session, the user can pick a color for a defined field for displaying on the screen. Color is enabled when the user is using a color capable computer workstation and that at least 36 free colors are available to the simulator. If color is not enabled all fields will be displayed using a default color which varies from system to system.

Once a field has been defined, if its bit depth is smaller than eight bits, state symbols can be chosen for that field, which are displayed in the "State Symbols of This Field" area. The user can define symbols of his choice to represent different states of this field. The user can also determine the strong or weak rotational symmetry of a state. If a state is weak rotational symmetry the next three states following it will be reserved as its weak rotational counterpart values. Recall from the definition that a state has weak rotational symmetry if it rotates to three other different state values. These four weak rotational states will be displayed on screen using the same symbol but at different orientations. The last three states of a field cannot be weakly rotational symmetry since there is not enough space after them to make up their weak rotational symmetry accompanying values.

If a field has more than seven bits the user cannot define individual state symbols for its states. A default symbol 'o' will be used for all its states when displaying on screen. This is because a large number of states makes individual state differentiation on the screen difficult and no longer meaningful. In addition, there is only a limited number of symbols in any computer font set we use. The user is encouraged to split up a large data field into smaller fields if feasible.

Do not confuse the rotational symmetry property of states of a field and the rotation symmetry property of the neighborhood template. The user can always choose weak rotational symmetry for states even when the neighborhood template is not symmetric. In this case the user simply picks the four orientations of a particular symbol to display the four states in a weak rotational symmetry group, without having any real relationship between each of them. Of course, if the neighborhood template is also symmetric and *rotated if* rules are used, these four states will again start to rotate into one another.

After the template is complete, the simulator will check the integrity of the template design as stated. If the template information is found to be consistent, the user can save the current template design to a file of his choice.

4.2.2 The main window

After the simulator has received all the template information it needs, it will display two major windows on the screen. One is the "Main Window" where most simulation activities are conducted and displayed. The other is the "Text Window" where cellular automata rules can be edited and compiled. We will look at functions of the main window first in this subsection. The Text window will be discussed in the next subsection.

Main window components

A sample of the main window is shown in Figure 4.6. The name of the most recently saved or loaded cellular automata world file is shown at the very top of the window. A cellular automata world file is used to store the cellular automata space configuration, usually with the `.world` extension in its filename. Below the names is the menu bar, from which the user can select a variety of menu commands to execute. Under the menu bar is a status bar, where various simulation information is displayed. The majority of the main window belongs to the working area, where the cellular automata space configuration is displayed and edited.

In the right side of the main window is a scroll bar which the user can use to move upward or downward the current viewing area of the cellular automata space. In the bottom of the main window is another scroll bar the user can use to make left or right adjustment of the viewing area. A simulated cellular automata space can be much larger than the screen can display at once, therefore we need these scroll bars to adjust the portion of the cellular automata space which is visible through the working area.

In the working area the user can see the currently simulated cellular automata space configuration. Field states are represented by their predefined symbols in the template information prescribed by the user. A set of weak rotational symmetric states are represented by the same

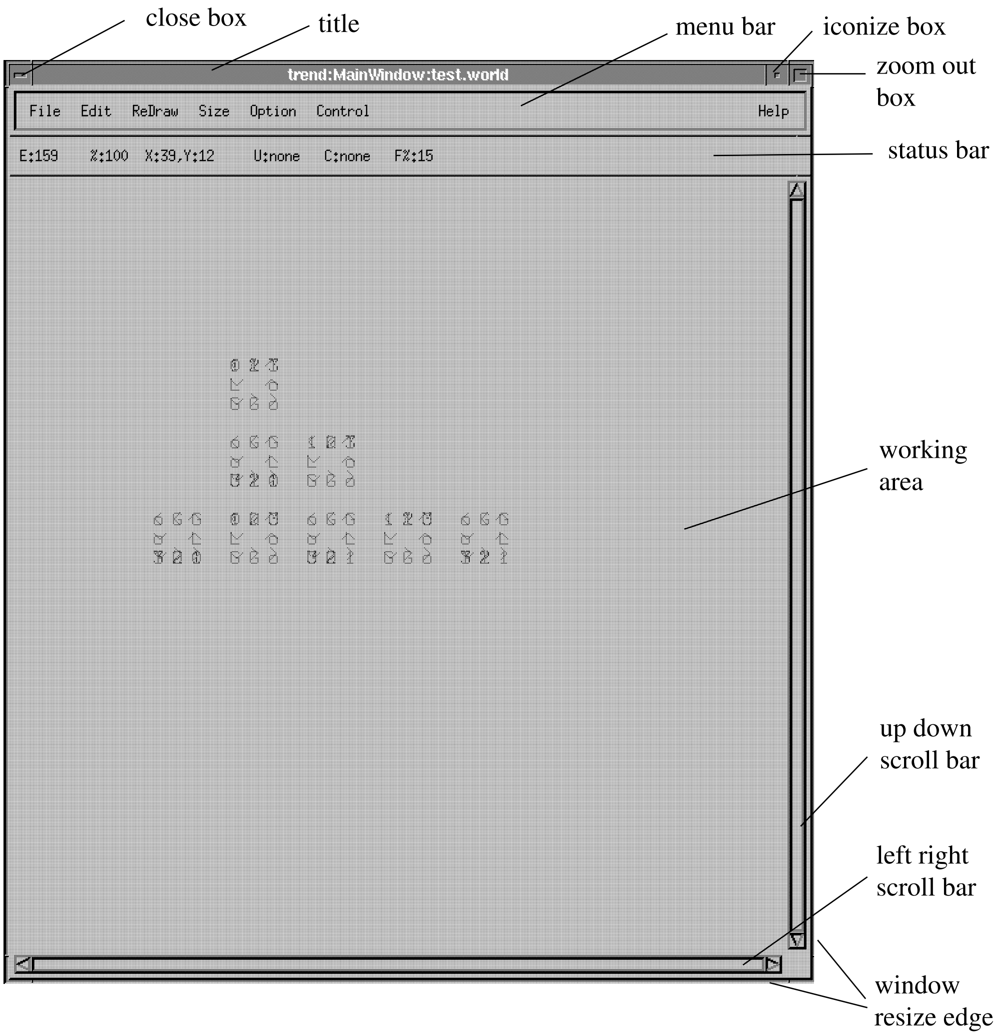


Figure 4.6: The main window of the simulator. In the top is information about the program name, window name and filename, followed by the menu bar. Under the menu bar is a status bar where simulation information is displayed. The working area makes up the largest portion of the main window where the cellular automata space is displayed. One vertical scroll bar and one horizontal scroll bar can be seen in the right and lower region of the window. Various window manager decorations surround the main window as marked. Their functions depend on the particular window manager the user uses. Characters representing the state of different data fields in a cell can overlay each other; they are easier to read off on computer screen than presented here because originally they are in color. These characters can also be rotated to represent weak rotational symmetric states.

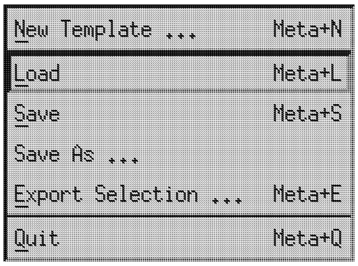


Figure 4.7: File related menu commands. "New Template" allows the user to change the current template information. "Load" allows the user to load in a cellular automata configuration from a `.world` file. "Save" and "Save As" allows the user to save the current cellular automata space configuration into a `.world` file. "Export Selection" lets the user export the currently selected region in the working area to an Encapsulated Postscript File for printing or inclusion into documents. The Encapsulated Postscript format, or EPSF, is a resolution independent file format suitable for high quality printing. "Quit" will terminate the simulator.

symbol, properly rotated. Different data fields can be chosen to be displayed or to be hidden from the scene using the floating "Control Window", which will be described later. The more data fields displayed, the harder it is to view each individual symbol. Therefore, it is up to the user to determine the suitable data fields to be displayed. The working area always reflects the current cellular automata space configuration. When a simulation is going on or when backtracking is triggered, the cellular automata space displayed in the working area will be updated accordingly.

File related commands

The first menu item of the menu bar is "File", which hosts a sub-menu with many commands as shown in Figure 4.7. The "New Template" command allows the user to change the current template information used by the simulator. The "Load" command allows the user to load in a cellular automata space configuration from a world file.

The "Save" and "Save As" command both allow the user to save the current cellular automata space configuration into a `.world` file, but the "Save" uses a default filename which is given in the title area as seen in Figure 4.6. The "Export Selection" command lets the user save the current selection in the working area into an Encapsulated Postscript file, which can later be used in printing or for inclusion into the other documents. All cellular automata space samples in the following chapters are made by this command. Finally, the "Quit" command will terminate the execution of the simulator.

The floating window for display control

There is a floating "Control Window" of the simulator, which is associated with the main window. It is used to control the display and editing of cellular automata fields in the working area. A typical "Control Window" is shown in Figure 4.8. Listed names in the right column are fields defined in the current template. Clicking on any of the field names will select that field as the current *focus field*. A focus field has two properties. First, the focus field content will always be displayed. Second, the field editing popup menu, which will be discussed next, will always be associated with the current focus field. There is only one focus field at any time; a new focus field

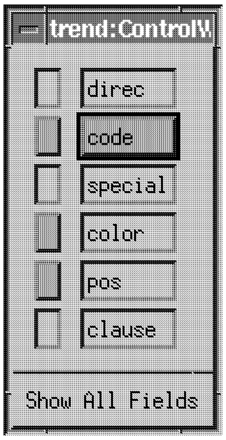


Figure 4.8: The floating control window. Field names are listed in the right column which the user can choose to set the focus field. In the left there is a set of check boxes the user can use to control displaying of the fields. The “Show All Fields” button will force all fields to be displayed, disregarding the status of their check boxes.

selection will replace the previous focus field.

To the left of the name list is a set of check boxes. The user can decide which data fields are necessary to be displayed at any time by setting these check boxes. It does not matter if a field is the current focus field or not.

Editing cellular automata space

Editing in the working area is simple, and is conducted by the combination of mouse actions and a few menu commands. Under the “Edit” menu item there is a set of commands for editing the cellular automata space configuration, which is shown in Figure 4.9.

The first command is “Undo”. No matter what change the user has just made in the cellular automata space, choosing the “Undo” command will undo that change.

A set of five commands “Cut”, “Paste”, “Copy”, “Clear” and “Drop” are the second group in the edit sub-menu. Their ranges of application are determined by the status of the floating control window. All fields currently being displayed will be affected. For example, the “Cut” command will remove everything within the currently selected region from every field which is currently being displayed into the “Clipboard”, which is a temporary storage area set aside by the simulator. The original content of all affected fields will be set to zero, which is usually the quiescent state for each field.

The region which the user “Cuts” is stored in the “Clipboard”, which the user can paste back into the cellular automata space by using the “Paste” command. The pasted content includes all fields that are previously stored in the “Clipboard”. The pasted clipboard content will form a new selected region which can replace the current region. A newly pasted region has **not** been actually combined with the cellular automata region under it. It is floating atop the cellular automata space. The user can adjust its position using the mouse. When the user has decided to settle the floating selection with the cellular automata space, he or she can choose the “Drop” command to drop the floating selection into the cellular automata space. The original cellular automata region under the floating selection will be replaced by the selection. Before the user drops a floating selection, the actual cellular automata space content has not been changed.

The “Copy” command behaves like the “Cut” command but does not remove the current cellular automata space content. The “Clear” command removes the current cellular automata space content but does not put it into the clipboard. As stated before, these five editing commands

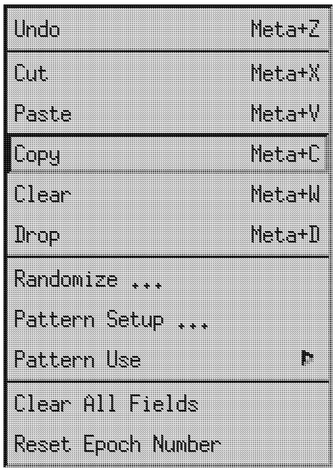


Figure 4.9: The editing commands. "Undo" will undo any recent change in the cellular automata space. "Cut" will copy the selected region into the clipboard and erase the region in the space. "Paste" will put the clipboard content into the cellular automata space. "Copy" will copy the selected region into the clipboard without removing it. "Clear" will erase the selected region without moving it into the clipboard. "Drop" will settle a floating selection into place. "Randomize" can generate some random states in the current focus field by user specifications. "Pattern Setup" can define the content of the focus field within the currently selected region as a pattern, which can later be used by the "Pattern Use" selection. "Clear All Fields" will set all fields in all cellular automata cells to the quiescent state (i.e., all zeros). Finally, "Reset Epoch Number" will reset the current epoch counter of the simulation to zero.

in the second menu group will influence any field which is currently being displayed, not just the focus field only. But the following three editing commands in the third menu group will influence only the focus field content.

The "Randomize" command displays a query window, as shown in Figure 4.10, which the user can use to randomize the current focus field. After the user has set appropriate conditions in this window, the focus field can be randomized with state values from the range specified by the two bounds and to the percentage set by the user. The "Randomize" command is a useful tool to randomly initialize a cellular automata space.

The "Pattern Setup" command defines the content of the current focus field within the selected region to be a *pattern*, which the user can use later to fill out a cellular automata field using the "Pattern Use" submenu, as shown in Figure 4.11. If the user chooses any pattern in this sub-menu, that pattern will fill the current focus field by being repetitively copied into the field.

In the bottom group of the edit sub-menu, there are two more commands. The "Clear All Fields" command will literally do what it says: clear all data fields in the cellular automata space. The final "Reset Epoch Number" command will set the current epoch counter to zero, which does not actually influence the cellular automata space but can be quite handy when the user wants to start counting a new simulation cycle.

One last thing which has not been mentioned is how the initial cell value can be entered into the cellular automata space in addition to just being randomly generated. The user can use the mouse button to pop up a state menu of the current focus field within the working area. The user can choose a state value to be placed in the cell under the mouse position from this popup menu.

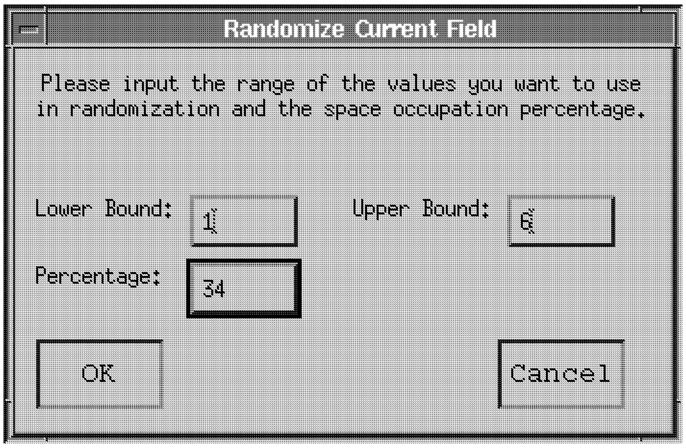


Figure 4.10: The randomize query window. The "Upper Bound" and "Lower Bound" are used to set a value range. The "Percentage" is used to set the filling density. The current focus field will be randomized accordingly.

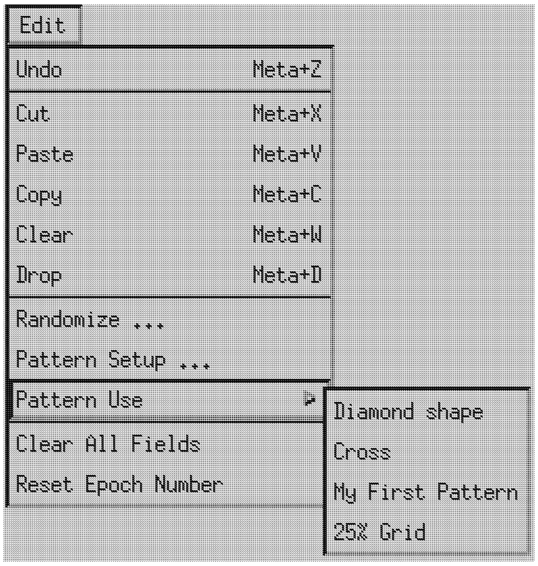


Figure 4.11: The pattern use sub-menu. The user can choose a pattern using this sub-menu to fill the current focus field.

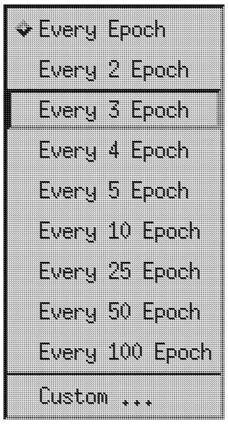


Figure 4.12: The Redraw speed selection sub-menu.

Changing the focus field will change the popup menu too, so the user can edit different fields.

Scrolling, re-drawing and resizing the viewing area

As said before, the simulated cellular automata space can be very large, larger than the screen can display at once. Therefore, the working area of the main window will be viewing just a portion of the simulated cellular automata space. There are two scroll bars in the right and bottom area of the main window which the user can use to *scroll* another portion of the cellular automata space into the viewing area.

The user can choose the screen update frequency by using the "ReDraw" menu item, as seen in Figure 4.12. This is useful when the user is not interested in individual simulation steps and wants to speed up the simulation. The size of the simulated cellular automata space can also be changed on-the-fly by using the "Size" menu item, as seen in Figure 4.13.

When the user chooses to enlarge the simulated space, the current smaller space configuration will be left in the center of the larger new one. Those cells originally connected in the boundary of the smaller configuration will now become disconnected. If the user chooses to shrink the cellular automata space, those cells beyond the smaller new boundary will be discarded.

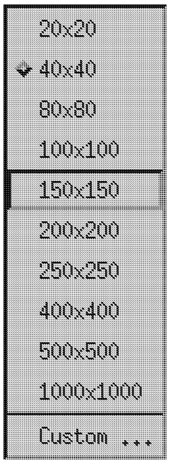


Figure 4.13: The Size selection sub-menu.

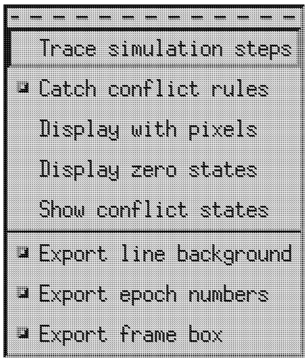


Figure 4.14: Options of the simulator. See the main text for descriptions of each option.

Options

There are some simulator options the user can choose, which are under the "Option" menu item as displayed in Figure 4.14. The dashed line in the top of the sub-menu denotes that this is a *tear-off capable* sub-menu panel, which means the user can select the dashed line to cut the sub-menu off the menu bar. This is very handy when the user wants to constantly change the options without going through the menu each time.

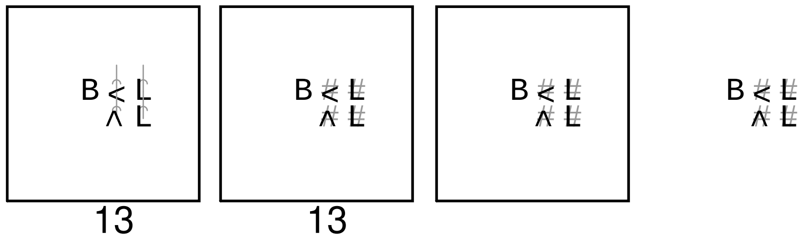
The first option is "Trace simulation steps". If set, it will enable the tracing mechanism of the simulator. Cellular automata space changes during the simulation will be saved to the file system if this option is on. The user can go back to a previous configuration using the backtracking commands only if this option is set. Tracing will use up available file system space temporarily (the trace file will be deleted when the simulator quits), but it allows very convenient trial-and-error development of the cellular automata rules.

The second option is "Catch conflict rules". Normally this option is set so that the simulator will try to catch cellular automata rules which are conflicting to each other during runtime, such as assigning different values to the same field of a cell. The conflict error is described further in Section 5.11 in the following chapter about the cellular automata programming language. This error catching mechanism is very useful when debugging the cellular automata rule set, but may incur some speed penalty to the simulator. The user should disable this option only when the cellular automata rule set has been found to be correct.

The third option is "Display with pixels". When the simulated cellular automata space is very large, sometimes it is hard to see the whole space even by constantly scrolling within the working area. The user can choose this command to display cells in the cellular automata space by pixels, instead of by symbols which the simulator normally uses. This can give the user a big picture of the cellular automata space but with lesser description power than using symbols. Each field is now displayed using only the same color pixels, so there is no telling which state value is at each cell. If multiple fields are chosen to be displayed, colors of different fields will be mixed, which makes reading even more difficult.

The fourth option is "Display zero states". Normally quiescent states in each field are represented by the zero state value. Zero states are not displayed by the simulator on screen since it is believed that these states are not interesting to the user and can potentially prevent a clearer view of the cellular automata space. The user can check this option to override that simulator behavior so any state value will be displayed by their associated symbols.

The fifth option is "Show conflict states". Normally a cell where a rule conflict is found is



(a) (b) (c) (d)

Figure 4.15: The effects of the EPSF export options. (a) all options are on. (b) "Export line background" is disabled. (c) "Export epoch number" is also disabled. (d) "Export frame box" is also disabled.

reported in the message area of the "Text Window", which will be described in the next subsection. If the user finds that it is hard to locate the cell by the coordinates reported in the message area, he or she can choose to show conflict states by reversed video too, so that a cell with conflicts can be easily identified. The trouble with this option is that undefined cells, where no applicable rules can be found to calculate the next state values, are also displayed with reversed video by the simulator. Therefore, there will be a problem knowing if a cell in reversed video is having a conflict or is undefined. The user has to check the message area coordinates to determine if it is an undefined or conflict cell.

The last three options deal with exporting the cellular automata space selection to Encapsulated Postscript files. Each of them affects the EPSF output in one of the following ways:

- "Export line background". This option will export background field state values between 1 and 4 using directed lines. The focus field or any state value not falling in that value range will be exported normally using symbols. This is very useful when the user uses the state values 1 through 4 to represent directions in the cellular automata space. Otherwise, it is seldom used.
- "Export epoch number". The current epoch number will be included in the EPSF file if this option is selected.
- "Export frame box". A rectangle box will be drawn around the selected region in the exported EPSF file.

Examples of how these three export options affects the look of the EPSF file are given in Figure 4.15.

Simulation and backtracking controls

The simulation progress is controlled by the sub-menu under the "Control" menu item as seen in Figure 4.16. The dashed line is again showing that this sub-menu can be torn off to form an independent floating window, just like the "Option" sub-menu mentioned in the previous subsection.

The "–>>" command will push the simulation forward continuously. The "–>" command will move the simulation forward one step at a time. The "stop" command will stop the current continuously forward or backtracking operation. The "<–" command will backtrack the simulation one step backward, and the "<<–" command will start backtracking continuously until reaching the first saved cellular automata configuration or until the user presses the "stop" button. Note

Figure 4.16: The control sub-menu. Arrow keys direct the progress of the simulation in the associated directions. Double arrow keys make it run continuously. “Stop” will stop any continuously running action. The dashed line allows the sub-menu to be teared off to form a floating window.

that backtracking is available only when the “Trace simulation steps” option is on and when at least one cellular automata configuration has been recorded.

The tracing mechanism is very handy since it allows the user to pinpoint the cellular automata evolution back and forth with great flexibility. But the tracing mechanism eats up disk storage, especially for large cellular automata space and long simulations. The simulator will attempt to keep an eye on the current file system usage ratio and will stop the tracing mechanism automatically with an error message to the user when the file system is about 90% full.

Status bar

The last undiscussed component of the main window is the “Status bar”. The Status bar is shown in Figure 4.17. Basically, some simulation and editing information is displayed in the status bar, which is outlined below:

- The first flag “E:” denotes the current epoch number. If the simulator is just starting up, the current epoch number is zero. When the simulation moves forward, this number will increase. When backtracking, this number will decrease.
- The “%:” flag shows how efficiently the caching mechanism of the simulator is functioning. The caching mechanism of the simulator tries to record recent cellular automata rule evaluation results for the oncoming evaluations so that if a result can be found in the cache lookup table, the rule evaluation can be skipped. If this efficiency value is low, it means most of the time the simulator has to resort to rule evaluations.
- The “X:” and “Y:” flags show the current coordinates of the mouse pointer in the cellular automata space covered by the working area.
- The “U:” flag shows the number of cells with *undefined* errors, i.e., cells whose next state value cannot be determined by the cellular automata rule set.
- The “C:” flag shows the number of cells with *conflict* errors, i.e., cells with more than one next state value when evaluated by the cellular automata rule set.
- The final “F%:” flag shows the current file system usage status. If this value is high, a file system full crash may occur with a higher probability.

Figure 4.17: The status bar. Various simulation and editing information is displayed in the status bar. See the text for explanations.

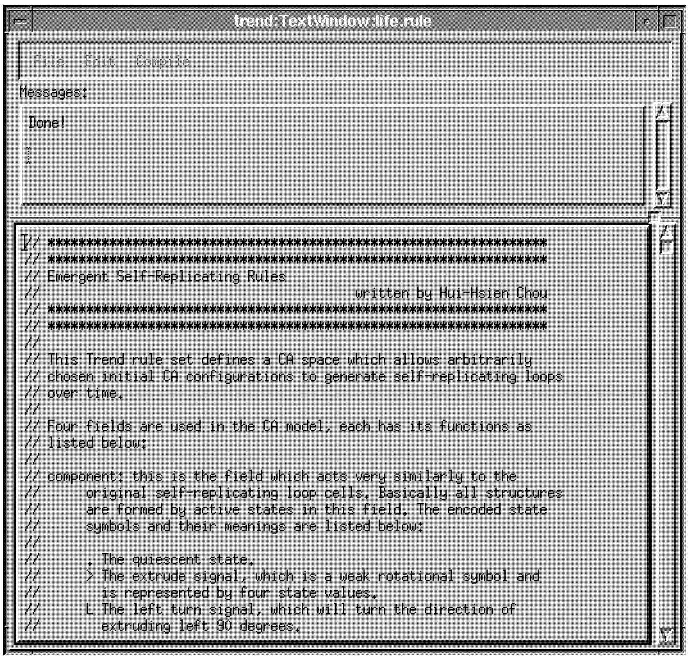


Figure 4.18: The text window of the simulator. The title shows the current program name, window name and file name of the simulator. A menu bar follows the title which has three submenus: "File", "Edit" and "Compile". Under the menu bar is a message area where various errors are reported, such as the compiling syntax error or the runtime conflict error. The text area occupies most of the text window, where cellular automata rules can be edited just like a text editor. The message area and the text area both have scroll bars to adjust the viewing position if the text to be displayed is larger than the size of the window.

4.2.3 The text window

Besides the main window, there is another "Text Window" that the user can use to load the cellular automata rule set into the simulator, edit and experiment with the rules, and save those rules back to a `.rule` file. A sample of the text window is shown in Figure 4.18.

Standard basic Gnus Emacs editing commands is supported in the text area. The rules within the text area can be saved to or loaded from a rule file by commands under the "File" menu item, which is shown in Figure 4.19.

Some text editing commands are shown in Figure 4.20. Two interesting commands "Jump" and "Goto" are listed in the third group. They do not actually change the text area in any way. The "Jump" command moves the cursor in the text area to a character position shown in the message area. Usually when a compiler error or runtime error is found, the simulator reports error spots with their character positions in the rule text. The user just needs to highlight (select) the position value, then choose the "Jump" command to jump to the error spot in the rule text. This function is very convenient when the user is trying to find the error spots within the rule text. The "Goto" command behaves similarly but asks the user to input the character position rather than getting it directly from the the message text.

The last group of editing commands deal with searching and replacement of strings in the rule text. The "Find" command asks the user a string to search for and then tries to locate that string

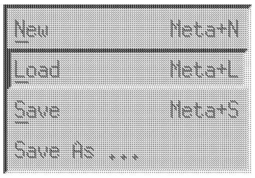


Figure 4.19: File commands for rule set manipulation. "New" will erase the content of the text area to prepare for a new rule set. "Load" will load a `.rule` file into the text area, which can later be modified and compiled. The "Save" and "Save As" command both try to save the current text area content into a `.rule` file, the difference being that the "Save" command will use the default rule file name shown in the title of the text window if there is one.

in the rule text if there is any. The "Find Next" command continues the search of the same string done used by a previous "Find" command. The "Find&Replace" command asks for both a search string and a replacement string so that if the search string is found in the rule text it will be replaced by the replacement string.

The final menu item is "Compile" which is used to invoke the Trend compiler. If compilation is successful the newly generated evaluation codes will replace the current codes directly, even when the simulation is still running. If there any error is found during compilation, it will be shown in the message area.

4.3 Under the hood

In this section we will see how the simulator is implemented. First, we will discuss the idea of *data fields* within cellular automata cells, as well as the neighborhood and field definition capability

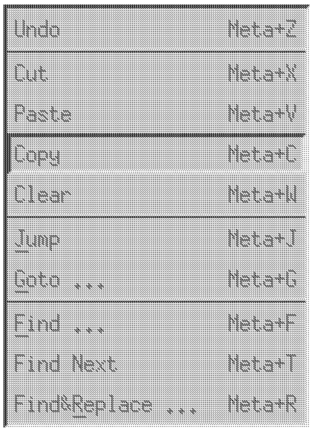


Figure 4.20: Editing commands for rule text. "Undo" will undo any recent change in the rule text. "Cut" copies the selected text into the clipboard and erases it. "Paste" puts the clipboard content into the text at the cursor position. "Copy" copies the selected text into the clipboard without removing it. "Clear" erases the selected text without copying it into the clipboard. "Jump" moves the cursor to a text position highlighted in the message area. "Goto" does a similar job but allows the user to specify the position. "Find" locates a string in text specified by the user, "Find Next" continues the search done by the previous "Find" command. "Find&Replace" replaces the search string with a replacement string specified by the user.

provided by the simulator. Next, we will see an overview picture of what the various modules of the simulator program are and how they are connected together. After that, we will discuss the major data structures used by the simulator, the transition function evaluation module, the tracing mechanism, the compiler, the memory management of the simulator, the template design module, file formats used by the simulator, the user interface construction, and finally the resource file of the simulator program.

4.3.1 The cellular automata space organization

In the early days of cellular automata, two dimensional cellular automata models had only two commonly used neighborhood templates: the von Neumann template and the Moore template[1]. Each cell of a cellular automata model was treated as a state variable, i.e., it could hold different values, but was considered to represent the same *property*. Actually, most of the time the state variable was just a binary bit showing 0 or 1.

With the advent of more complex modern cellular automata models, new neighborhood templates were introduced for different applications, but the two classic neighborhood templates still retain their popularity because they are simple and regular. However, a single variable cell design became very limited for any but the simplest models. To solve this limitation, the concept of cell division into *fields* (sometimes called *layers* or *planes*) was introduced [Toffoli & Margolus, 1987]. Fields are functional divisions of the state variable of a cell into different bit groups, each encoding a different property of the cellular automata model being used. We can think of each field as a slice of a cell state variable with its own name and value. Alternatively, we can just treat each field as a different state variable in a cell, as if each cell can now hold many state variables.

For example, the emergent self-replicating rule set (which will be discussed in Chapter 6) uses four data fields, `component`, `special`, `growth` and `bound`, as shown in Figure 4.21. Each field is assigned a name and can be referenced with that name in the cellular automata programming language. The automata transition function used to compute a new value for a field can be based on current values of many fields in many neighbors of the neighborhood template. With the help of fields, a very complex cellular automata model and its rule set can be designed in a concise and easy to understand manner.

Up to 64 bits can be allocated for a cell. The simulator described here allows an arbitrary number of data fields to divide this 64 bits. The neighborhood template is no longer predefined and limited to a number of well-known templates. With the new template design facility mentioned above, any cell within an 11 by 11 cell region around the center cell can be part of a neighborhood template. The possibility of new template designs becomes enormous.

4.3.2 The construction of the simulator

A functional view of the simulator software components and their mutual relationships are depicted in Figure 4.22. In this figure, data structures are represented by rectangle boxes and program modules are represented by rounded rectangle boxes. The relationships between different program components are indicated by arrows. Modules in the gray area are called by all the other modules, so arrows are omitted from them.

---

[1]See Figure 2.1 at page 5 for the definition of the von Neumann and Moore neighborhoods.

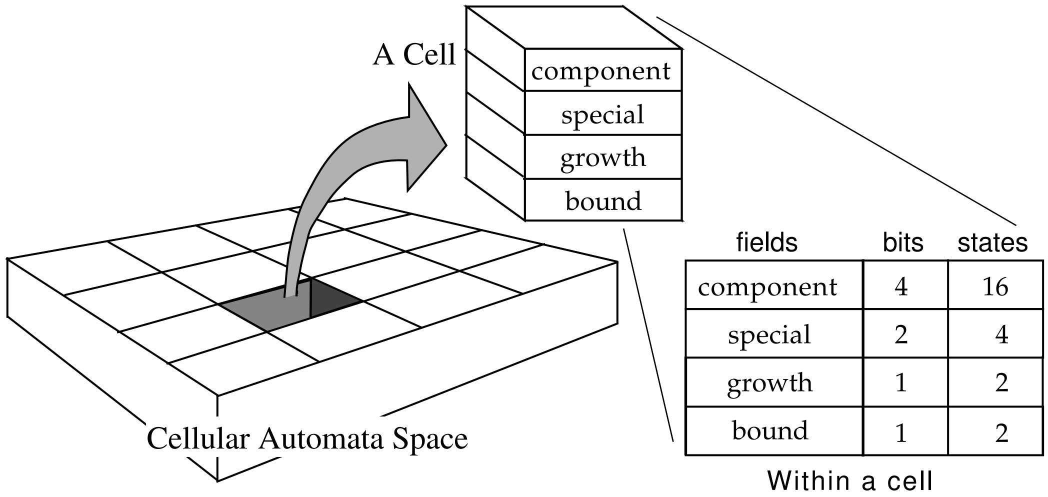


Figure 4.21: The cellular automata fields. The cellular automata state variable in each cell is horizontally sliced into different bit groups called *fields*. Each field represents a specific piece of information in the cellular automata model, and is supported by rules which compute its new value during each iteration. Different bit-depths are assigned to different fields as indicated. The total number of bits used is the size of the state variable in each cell; the simulation program keeps track of this information.

The simulation control module calls either the tracing mechanism module or the evaluation module to modify the simulated cellular automata space. The change in the cellular automata space configuration is reflected to the screen by the display module. The tracing mechanism uses its own files to store the tracing data to the file system, which makes returning to a previous epoch possible.

The evaluation module can be viewed as the core of the whole simulation system. It uses the compiled rule codes generated by the compiler and references the template information maintained by the template design module to determine the next state of the cellular automata space configuration. It also maintains a cache table for itself to speed up evaluation. The compiler takes the rule text source and also references the template information to generate the compiled rule codes which is used by the evaluation module.

There are two editing modules, one for editing the rule text, the other for editing the cellular automata space configuration. The cellular space editing module also uses an auxiliary cellular automata space of its own to maintain the "Undo" information.

The four supporting modules (memory management, input/output, user interface and error recovery) are called by all the other modules in the simulator. Their involvement with the other modules is deep and more than what is shown in Figure 4.22.

In the following several subsections each component will be discussed separately.

4.3.3 Major data structures

The most obvious data structure the simulator uses is the cellular automata space data. It is amazingly simple, consisting of only pointers to two character arrays. They are declared in the following C language constructs:

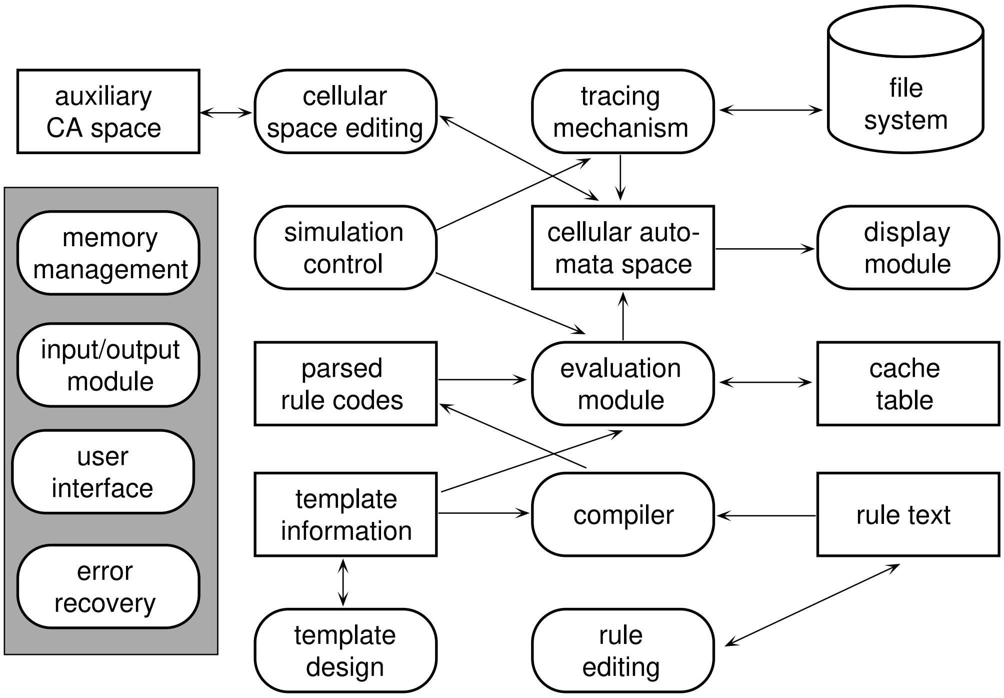


Figure 4.22: The structure of the simulator program. Data structures are represented by rectangles. Program modules are represented by rounded rectangles. Reference or data flow directions are represented by arrow lines. If a data structure is referenced by a program module, there is an arrow line from the data structure to the module. If a program module writes a data structure, there is an arrow line from the module toward the data structure. If a module both reads and writes a data structure, a two-way arrow links them. The simulation control module calls both the tracing mechanism module and the evaluation module, so there are arrow lines from the control module toward both the tracing and evaluation module. In the gray area to the left of the figure there are some supporting modules which are called by all the other modules. To simplify matters, lines are not drawn for them.

```
typedef char Cell;
Cell *OldWorld, *NewWorld;
```

The simulator needs to maintain two separate arrays `OldWorld` and `NewWorld` in order to compute the next state value from the current state. To obtain maximum flexibility for the size of the cellular automata space, a one dimensional array is favored over the standard two dimensional C language array to represent the space. The simulator keeps track of the shape of the two dimensional cellular automata space with the help of a couple C macros and the `WorldSize` variable. All two dimensional references to the cellular automata space are converted by the simulator into the one dimensional array.

Compiler generated virtual machine code for the cellular automata rule set is kept in the `ParseNode` records, with one record per one code. The declaration of the record is shown below. Each `ParseNode` record is like a machine code with one instruction field and four branching address fields left, right, true and false which are maintained by the four pointer fields `left`, `right`, `true` and `false`. The `operator` field keeps the instruction code and the `value` field keeps the immediate value, if any, of the machine instruction. Valid instruction codes in this virtual machine language are listed in Table 4.1 at page 55, which will be discussed in detail in the following chapter about the compiler.

```
typedef struct ParseNode {
  int operator, value;
  struct ParseNode *left, *right, *true, *false;
} ParseNode;
```

The cellular automata template information is kept in two separate arrays of `NeighborStruct` and `FieldStruct` records. Each neighbor or field information is kept in one record in those arrays. The two record declarations are listed below, together with the explanation of their data fields.

```
typedef struct NeighborStruct {
  int id; /* The integer id of the neighbor */
  /* Pointer to its check box in the ``Template Design'' window */
  Widget w;
  char *name; /* Referenced name of the neighbor in rules */
  /* Positions of the neighbor in the neighborhood template.  */
  struct {
    int x, y;
  } offset[4];
} Neighbor;
```

The `id` field records the integer id of the neighbor. It starts from zero to one minus the number of neighbors in a neighborhood template. The `w` field contains a pointer to a check box in the "Template Design" window (Figure 4.5 at page 36), which is used to associate the neighbor with the check box. The `name` field records the name of the neighbor when referenced in the cellular automata rule set. This name actually is part of the reserve word set of the Trend language. The final four `x, y` pairs in the `offset` field record the actual coordinate positions of the neighbor, when properly rotated. If the neighborhood template is not rotationally symmetric, then only the first `x, y` pair is used.

```
typedef struct FieldStruct {
  int bits; /* The number of bits in the field */
  int states; /* The number of states in the field */
  int show; /* If user chooses to display this field on screen */
  char *name; /* Referenced name of the field in rules */
  unsigned mask, mark; /* Data used by the evaluation module */
  int shift; /* Data used by the evaluation module */
  int type; /* Data used by the evaluation module */
  Widget popup; /* Pointer to the popup menu for the field */
  int pixel; /* The color of the field for displaying */
  GC normal, reverse; /* Data used by the display module */
  char *symbols; /* The defined state symbols of the field */
  char (*rotation)[4]; /* The state rotational values array */
  Pixmap *positive, *negative; /* The normal and reverse video */
  int select; /* See if this field is selected for output.  */
} Field;
```

The bits field records how many bits are allocated to this field. The states field records how many states are expressible in this field, which is always $2^{\texttt{bits}}$. The show field determines if this field should be displayed on screen or not, which is set by the user using the "Control Window" (Figure 4.8 at page 40). The name field store the name of the data field which is referenced in cellular automata rules. This name becomes part of the reserve words of the Trend language when the template information containing this field is loaded. The mask, mark, shift and type fields are lower level auxiliary data used by the evaluation module for cellular automata transition function computation. The popup field contains a pointer to the popup menu of the field which is used to manually input state values into this field by the user. The pixel field stores the color for displaying this data field on screen. The normal and reverse fields are used by the display module for displaying normal cells and cells with undefined or conflict errors. Same is the positive and negative field.

The symbol field stores all symbols defined by the user to represent states of this data field. The rotation table records the mutual weak rotational symmetric states for states. For example, if states 1 to 4 are within a weak rotational symmetric group, then rotation[1] will contain 1, 2, 3, 4, rotation[2] will contain 2, 3, 4, 1, rotation[3] will contain 3, 4, 1, 2, etc. Finally, the select field determines if a field is chosen by the user when outputting the current simulated cellular automata space configuration to a file.

The cache table used by the evaluation module is built from the following basic record unit:

```
typedef struct Cache {
  char *lhs, *rhs;
  int replace, visit;
  struct Cache *search;
} Cache;
```

The cache table stores the recently computed cellular automata transition function values. It is both a hash table for fast value lookups and a priority tree for value replacements. The lhs and rhs fields keep the priority tree structure, and the field keeps the hash table structure. Collision of a hash table entry is resolved by using the search field to form a linked list in that entry. The visit counter keeps the number of references to a particular record and the replace counter keeps the order in the priority tree. When a record is referenced again, it is moved down in the priority tree. When a new record is needed to store a new transition function value, the top node of the priority tree is used.

The last major data structure used in the simulation software is the tracing array used by the tracing mechanism module which is composed of the following data record:

```
typedef struct TraceRecordStruct {
  int num, pos, epoch;
} TraceRecord;
```

The pos field keeps track of the file position in the trace file on the file system. The epoch field keeps the epoch number represented by this record and the num field stores the number of the trace record.

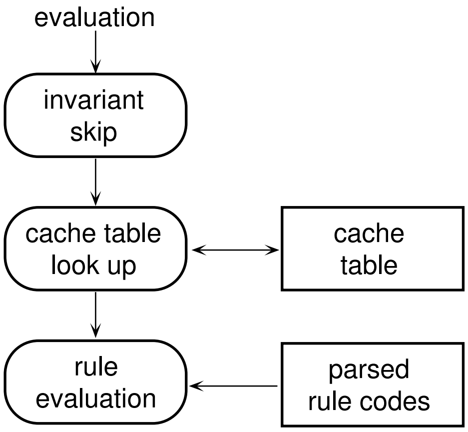


Figure 4.23: The evaluation steps. The evaluation module first skips invariant cells. It then uses the cache table to quickly find any applicable previous evaluation result which it can use. If both methods fail the evaluation module will have to resort to the compiled rule codes to do the evaluation.

4.3.4 Evaluation

The evaluation module calculates the next cellular automata space state from the current one, using the rule code compiled by the compiler from the source rule set expressed in the Trend language. The evaluation module uses a three step strategy to conduct the next state evaluation process. This is depicted in Figure 4.23.

The first step is invariant checking. Most cellular automata cells, especially during earlier simulation steps, are at quiescent states. According to the definition of the cellular automata model, a quiescent cell in the cellular automata space will remain quiescent indefinitely, until its neighbors become active. The evaluation module takes advantage of this fact. Actually, it takes it even further. If a cell and its neighbors have not changed their values during the past epochs, that cell can be skipped in the evaluation process since its value will not change in the current epoch. That is exactly what the evaluation module does to avoid evaluating invariant cells.

If the cell has different neighbor values than the previous evaluation, or if its value has changed in the previous evaluation, the evaluation module has to calculate its next state value. The evaluation module first looks at the cache table to see if it can find any recent evaluation which has exactly the same neighbor configuration, or transition function inputs, as the cell in question. If such a case can be found its evaluation result will be taken to be the next state value for the cell, so actual evaluation is not necessary. The `visit` counter of the found case in the cache table is increased by one and the entry is also moved down in the priority tree, so that its chance of getting replaced is lower.

If none of the above procedures works, the evaluation module will have to calculate the next state by using the compiled rule codes. First, states in a cell and its neighbors are split into individual fields which can be referenced directly and efficiently by the compiled rule codes. Then, the rule codes are executed. After the termination of the rule codes, the evaluation module will check to see if there is any undefined field value for the cell. If found, the cell has *undefined* errors. If the evaluation module finds that multiple assignments have been made to the same field, the cell has *conflict* errors. In either case, the evaluation module will report errors and mark the problem cell.

If rule execution is successful for a cell, the evaluation module will pack new fields back into the cell. It will also put the new evaluation result into the cache table for future references. If no empty entries can be found in the cache table, the less frequently referenced entry will be replaced

PFIELD1 1	PAND 11	PPLUS1 21	PLEQ 31	PMOD 41
PFIELD2 2	POR 12	PPLUS2 22	PLT 32	
PFIELD3 3	PXOR 13	PFSET 23	PRETURN 33	
PFIELD4 4	PVALUE 14	PFPLUS 24	PROTS 34	
PADDR 5	PCALL 15	PDEF 25	PRVALUE 35	
PARRAY 6	PROT 16	PNEQ 26	PFMINUS 36	
PADD 7	PROTE 17	PNZE 27	PMINUS1 37	
PSUB 8	PBREAK 18	PGT 28	PMINUS2 38	
PMUTL 9	PSET1 19	PGEQ 29	PRFIELD1 39	
PDIV 10	PSET2 20	PEQ 30	PRFIELD2 40	

Table 4.1: Virtual machine codes used by the simulator.

by the new evaluation result.

If an evaluation error is found during the evaluation process, the simulation control module will stop the simulation. The display module then displays the new cellular automata space configuration on screen. Marked cells will be displayed in reversed video. If no error is found during the evaluation and the user chooses to continuously simulating, the simulation control module will exchange the `OldWorld` and the `NewWorld` pointer and continue the simulation.

4.3.5 Tracing mechanism

If the user chooses to trace the cellular automata simulation, the tracing mechanism of the simulator will save the **differences** between each iteration to a temporary trace file on disk. The location and size of each iteration difference information within the trace file is recorded in memory using the trace record array. The tracing mechanism uses dynamic addressing into the trace file to recall previous configuration differences from the current configuration, and applies them to the current cellular automata space to get to previous epochs.

4.3.6 Compiling

The compiler for the Trend cellular automata programming language uses a standard LR(1) grammar to describe the rule syntax, and was built by using the `yacc` compiler construction tool in Unix. The compiler currently is strictly one-pass with no forward references allowed in the rules. That means all variables and functions must be declared before being used. Because of this, recursive procedure call is impossible with the Trend language. Actually, for efficiency reasons, all variables are allocated statically in a heap rather than dynamically on a stack. Therefore, multiple entrances to the same procedure may produce unpredictable results. It is determined that for cellular automata programming none of these unsupported features are needed.

The compiler generates compiled rule codes represented in a virtual machine instruction set, which is listed in Table 4.1. The benefit of compiling into a virtual machine code rather than to the host computer machine code is that it is machine independent. Porting the simulator to the other workstation platforms does not require rewriting of the code generation modules in the compiler. The drawback of this is that a simulation runs about three times slower than if it is using real machine instructions. But with the help of invariant skipping and cache table lookup in the evaluation module, the slowdown is more than justified by the portability and stability.

The compiler optimizes output machine code by removing constant expressions (expressions which do not involve variables) and by coercing multiple indirect jumps to one direct jump. Since the compiler is one pass only and is efficiently implemented, it can compile thousands of Trend source lines in less than a second, essentially making the compiling phase not noticeable by the user. The compiler can be called even during active simulation runs. The compiled rule code it generates will immediately replace any previous code and be used by the evaluation module.

We will discuss the virtual machine instruction set and the compiler in more detail after we have introduced the Trend language in the following chapter.

4.3.7 Memory management

The simulator attempts to do memory management itself rather than by using the standard C language memory management functions such as `malloc()` and `free()` directly. Since only a limited number of fixed-size data types are used by the simulator, the simulator tries to form a pool of free data objects for each type internally. Whenever a data object is freed, it is returned to the object pool rather than to the heap maintained by the standard C routines. When a new object of the same type is needed, objects in its associated pool will be provided first. In case there is no more free objects in a particular pool, the memory management module of the simulator will then call the standard memory allocation routine again to get a bunch of the new objects at once. Surplus objects will be returned to the pool.

Managing data objects by the simulator itself can greatly improve the execution speed and prevent heap fragmentation caused by using the standard C library memory management routines.

4.3.8 Template design

The simulator makes use of the "Template Design" window shown in Figure 4.5 on page 36 to provide template design controls to the user. The template design module will check the consistency of the user design, as stated before. The template information generated by the template design module is stored in the neighbor and field information arrays and is used by both the compiler and the evaluation module.

Currently the template design module allows neighbors within an eleven by eleven square, rooted on the center cell, to be put into the neighborhood template. It also allows a maximum of 64 bits in each cell to be used for data field allocations. These limitations are arbitrary rather than mandatory, and can be extended by resetting some of the program parameters, but it is found that these limitations are seldom reached by any practical application of the simulator, i.e., they are more than enough for normal uses.

4.3.9 File formats

The simulator makes use of three different file formats for recording the template information, the cellular automata space configuration, and the rule set source text. The rule set source text is just like any text file in the computer system, and can be edited by any popular text editor.

A sample of the template information file is given below. We can see that it records most of the data fields in the `NeighborStruc` and the `FieldStruc` records. Although the template file is designed to be readable, it is not meant to be edited by the user in anyway. If the user wants to modify a template file, he or she should load that template file using the simulator and use the "Template Design" window of the simulator to do the job.

```
File: life.tmpl
Created Mon Mar 4 21:01:16 1996
BitDepth=8, ByteDepth=1, LhsLength=9, UseColor=1
Symmetry=1, NeighborNumber=9, FieldNumber=4, SolvedMask=17
Neighbor Descriptions:
  ce: (0,0), (0,0), (0,0), (0,0)
  no: (0,-1), (1,0), (0,1), (-1,0)
  ne: (1,-1), (1,1), (-1,1), (-1,-1)
  ea: (1,0), (0,1), (-1,0), (0,-1)
  se: (1,1), (-1,1), (-1,-1), (1,-1)
  so: (0,1), (-1,0), (0,-1), (1,0)
  sw: (-1,1), (-1,-1), (1,-1), (1,1)
  we: (-1,0), (0,-1), (1,0), (0,1)
  nw: (-1,-1), (1,-1), (1,1), (-1,1)
Field Descriptions:
  component: bits=4, states=16, mask=17, mark=1, shift=28, type=1, color=26
      .>___LOEFBCDoooo
  special: bits=2, states=4, mask=3, mark=2, shift=26, type=1, color=32
      .-*#
  growth: bits=1, states=2, mask=1, mark=4, shift=25, type=1, color=18
      .+
  bound: bits=1, states=2, mask=1, mark=8, shift=24, type=1, color=9
      .!
```

The above denotes that the template information is saved to the file `life.tmpl`. The template uses 8 bits in a cell for field allocations (`BitDepth=8`), which is exactly one byte (`ByteDepth=1`) long. The neighborhood template has nine neighbors (`NeighborNumber=9`), and each cell is one byte long, so the total input length to the cellular automata transition function is 9 bytes (`LhsLength=9`). This template has color information encoded for each field (`UseColor=1`). Its neighborhood template is rotational symmetry (`Symmetry=1`). It has four data fields (`FieldNumber=4`), and the bit pattern for checking for four data fields are 1111, which is 17 if expressed in octal number format (`SolvedMask=17`).

After the template parameter definitions, the actual neighbor and field descriptions follow. Each neighbor description entry contains the name of the neighbor `ce`, and the four rotational symmetric positions of the neighbor, expressed in coordinate pairs in the neighborhood template. The center cell of a neighborhood template always has the coordinates (0,0). Each data field description contains the name of the field `component`, the number of bits allocated to the field `bits=4`, the number of states expressible in the field `states=16`, the mask, mark, shift and type data used by the evaluation module, and finally, the color for displaying this data field on screen `color=26`. In the second line of a field description, it lists the symbols defined to represent states of this field `.>___LOEFBCDoooo`. The three underline symbols "_" and the symbol > before them means that they are weak rotational symmetric states and are displayed on screen by the same symbol > but rotated differently.

The cellular automata space configuration is recorded in a `.world` file. A world file consists of a readable preamble part which details the saved fields in the world file, together with a binary unreadable part which actually records the configuration. A sample of the readable world file preamble is shown below.

```
File: sample.world
Created Tue Mar  5 19:23:31 1996
WorldSize=40, SavedFields=4
Field name and depths:
        component: 4 bit(s)
        special: 2 bit(s)
        growth: 1 bit(s)
        bound: 1 bit(s)
```

It says that the cellular automata space configuration is saved to the file `sample.world`, the cellular automata space has 40×40 cells (`WorldSize=40`, and there are four data fields saved in this filed (`SavedFields=4`). After that, the names and bits for each saved data field are listed.

4.3.10 User interface

The simulation program provides a graphical user interface to present great ease of use of its functions to users. The graphical user interface allows the user to interactively pursue rule set development and cellular automata simulations. The power of the simulator is greatly increased by the ease of use of the interface.

Currently, the simulator is built around the X Window system under Unix. The simulator uses the Motif widget library to provide the user interface components, such as the menus and scroll bars, but all other interface features like cellular automata space drawings on screen and template design window operations, etc., are still implemented in the simulator itself.

Since the core simulation routines and compiler are not dependent on the user interface design, porting the simulator to other platforms involves rewriting the user interface components only.

4.3.11 The resource file

The current simulation program under Unix makes heavy uses of the X Window *resource file* mechanism. Whenever possible, any feature and look-and-feel setup is defined in the resource file, rather than hard-coded into the program source code. Working with the resource file has the benefit of flexibly changing the appearance of the simulation program without the need to recompile the source code. All colors and text labels that the program uses can be modified. Even changing to a foreign language for multi-language ports is easy using the resource file. A portion of the resource file which defines the "File" sub-menu content is shown in the following example. It can be seen that all the hot keys, names and colors can be changed here.

```
*MenuBar*foreground: yellow
*MenuBar*NewTemplate.labelString: New Template ...
*MenuBar*NewTemplate.mnemonic: N
*MenuBar*NewTemplate.accelerator: Meta<Key>N
*MenuBar*NewTemplate.acceleratorText: Meta+N
*MenuBar*Load.mnemonic: L
*MenuBar*Load.accelerator: Meta<Key>L
```

```
*MenuBar*Load.acceleratorText: Meta+L
*MenuBar*Save.mnemonic: S
*MenuBar*Save.accelerator: Meta<Key>S
*MenuBar*Save.acceleratorText: Meta+S
*MenuBar*SaveAs.labelString: Save As ...
*MenuBar*Export.labelString: Export Selection ...
*MenuBar*Export.mnemonic: E
*MenuBar*Export.accelerator: Meta<Key>E
*MenuBar*Export.acceleratorText: Meta+E
*MenuBar*Quit.mnemonic: Q
*MenuBar*Quit.accelerator: Meta<Key>Q
*MenuBar*Quit.acceleratorText: Meta+Q
```

Modification of the resource file should be done as carefully as changing the program source code, otherwise unexpected errors may appear. The resource file for the simulator should be installed in the X Window resource file directories for all users to share. Some user changeable properties, such as the font used to display cells or the color of the selection dashed lines, can be changed by an individual user by putting those values of his preference into his own `.Xdefault` file.

4.4 Comparison to previous simulators

The CAM machine and *Cellular* system described in Section 2.3 are more or less designed for simulating physical systems, notably reaction-diffusion systems [Keener & Tyson, 1986; Toffoli & Margolus, 1987]. Their design objective is very different from the simulator presented in this chapter. This new simulator is designed for the sole purpose of complex cellular automata rule set development, which supports feature-rich and somewhat more complex cellular automata structures. In the development of cellular automata rule sets which support a desired behavior such as self-replication, we need to observe constantly the simulation results of our current rule set under development, modify the rule set, go back in time, and run the simulation on the same starting cellular automata space configuration again. This kind of design approach is not easily supported in the two aforementioned systems. In those systems, the cellular automata space configuration is not saved automatically unless the user gives specific commands to save it. Loading and saving a cellular automata configuration is not as easy as the one click backtracking offered in our new simulator.

The CAM machine has at most four bit planes, which support only 16 states in a cell; it is therefore unable to support the cellular automata model studied in this research. The new simulator supports up to 64 bits in a single cell, which can be further divided into separate named data fields for different purposes. But it is worth noting that since the CAM machine is a hardware accelerated cellular automata simulator, for any particular cellular automata model this machine is capable of simulating, it can usually simulate it much faster than software-based simulators. The *Cellular* system also allows definition of arbitrary data fields in a cell, but it does not allow multiple data fields to be displayed on screen at the same time, nor can it use symbols to represent states of each field. The *Cellular* system can only display cells on screen using colors and, in some platforms, the numerical values of a field. Our new simulator allows the definition of colors and symbols to represent data fields on screen for any field having less than 128 states. Those symbols which represent weak rotational symmetric states can even be rotationally displayed on screen if necessary.

Neighborhood configurations are predefined in the CAM machine and cannot be easily modified or extended. The *Cellular* system allows arbitrary neighborhoods to be used, but it does not allow symbolic naming of the neighbors in a neighborhood template. References to neighbors therefore have to be done using indexing conventions, which sometimes require some imagination to be used properly.

All of the other main features of the new simulator, like direct entering of cellular automata states using popup menus, direct exporting of cellular automata space content to Encapsulated Postscript files, the easy cut, copy and paste editing operations of the cellular automata configurations, etc., are not seen in the other systems. The on screen design of new cellular automata models using the "Template Design" dialog window is especially convenient. It provides at a glance all the essential structures of a new cellular automata model being specified. The direct linking of neighbor and field names to compiler reserved words is also very handy; we will see their usefulness in the following chapter.

4.5 Discussion

In this chapter a general purpose cellular automata simulator is presented. Currently this simulator is available on Unix platforms, but it will soon be ported to the other two popular computer systems, the Apple Macintosh and Microsoft Windows 95/NT. All the cellular automata world files, template files and rule files are machine independent, so everything developed on Unix can be used on the other platforms without any modification.

Currently the simulator is reasonably fast for world sizes up to 500 by 500 cells, say, at about one iteration every 20 seconds on a Sun Sparcstate 20. Beyond that, the speed of the simulation can be slow, which hurts the original interactive design of the simulator. Since an intrinsic characteristic of cellular automata is that they are scalable, it should be easy to port the simulator, or at least the evaluation module portion of it, to a multi-processor parallel computer. This can greatly improve the speed of the simulation. Actually, it is possible to build the simulator so that when running on a parallel computer, it can dynamically adjust workload to available processors installed, so no rebuilding of the simulator program is necessary when the power of the parallel computer improves. The Trend cellular automata programming language (to be discussed in the next chapter) or the user interface does not have to be modified to port it to the parallel computer.

The simulator is for two dimensional cellular automata simulations. It can be used for one dimensional cellular automata simulations, too. It may not be easily used with three or higher dimensional cellular automata simulations, although the large bit depth of 64 in each cell can be used for simulations of some specific three dimensional layered models, such as the neural network simulations. Neural network simulations usually use a limited number of two dimensional layers connected vertically with one another, which is perfect for this simulator.

It is possible to modify the evaluation engine of the simulator to support higher dimensional cellular automata simulations, but actually, for such higher dimensional simulations, the technical difficulty is not how to simulate them, but how to visualize the results. A three dimensional visualization library such as the PEXLib for the X Window system or the QuickDraw 3D system extension for the Macintosh can be used for that purpose.

Chapter 5

Trend: A High Level Cellular Automata Programming Language

Trend is a high level language for cellular automata programming[1]. It is used in the cellular automata simulator described in the previous chapter. Traditionally, cellular automata transition functions are depicted in a tabular format, which lists all mappings from the neighborhood configuration domain into the next state value range. This kind of representation has at least two problems:

- When the the number of states in each cell or the number of neighbors in the neighborhood template gets bigger, the table size grows exponentially, making it hard, if not impossible, to represent the table explicitly in the limited computer memory.
- A tabular representation of a cellular automata transition function is not easy to understand since it does not explicitly convey the idea of the rule set to the reader. It is hard for readers to understand a table full of plain numbers. In addition, it is inconvenient for a cellular automata rule set designer to convert his ideas into the tabular format, something he must do first before he can start testing the ideas.

Because of these problems, a high level structured programming language approach is taken. In this approach the cellular automata transition function is **implicitly** defined by the algorithmic operations expressed in the language. These operations define how a next state value can be calculated based on the various conditions in the cellular automata space. Because the Trend language contains most modern programming language constructs, it allows algorithms expressed using it to be very complex, yet still quite readable. This greatly extends the power of cellular automata programming when compared to tabular rule sets.

Previously there has been some similar work to improve cellular automata programming. The programming language used in the CAM-6 machine was a semi-high level language based on the stack-operated language Forth [Toffoli & Margolus, 1987]. This language used the postfix statement format rather than the infix format commonly used in modern programming languages, which could impose some difficulties in learning it. The idea to slice neighbor values into data fields was introduced in this system, too. Trend itself was modeled after the popular programming language C, with cellular automata specific constructs added to it. These constructs include statements to scan

[1] It is named "Trend" because we hope that it will be useful in programming the *trend* of cellular automata evolution.

all neighbors in a neighborhood template, special data types to access data fields in the neighbors, special notations for rotatable[2] literal value representations, and a special rotated *if* statement to exploit the rotational symmetry characteristic of cellular automata. A previous knowledge of the C language could make learning the Trend language very straightforward, but it is not required.

Conceptually, a high level language for cellular automata programming should provide abstract, named data items to denote the data fields within neighbors in a specific neighborhood template used in a particular cellular automata model. Based on those data items, new values are computed and assigned to names representing fields in the center cell. These values are taken as the next state values of the cellular automata. The user's job of finding and listing the transition function of a cellular automata model in the past now becomes the job of defining how the next state values can be computed from data items representing the current cellular automata neighbor values. This is familiar and straightforward using high level algorithmic language constructs for anyone with some basic modern programming language experience. Therefore, adapting to cellular automata programming becomes nothing more than learning a new programming language. The tedious work of the past, sometimes involving writing a complex program or making a complex rule table just to start up a simple simulation, is no longer needed once the user has mastered the Trend language and its associated simulator.

The Trend compiler is a full-fledged compiler bundled with the cellular automata simulator program. Although the language is modeled after C, the compiler itself is not an extended C compiler. Instead, it is a new compiler implemented from scratch, because cellular automata rules work in parallel among cells, which are intrinsically different from sequential C programs. A user loads the Trend language source code into a text window and then invokes the compiler to parse the code. If no error is found, the compiler will generate a virtual machine code which the simulator uses for efficient runtime evaluation of the cellular automata rules. If the runtime behavior of the cellular automata is not what the user wants, the user can modify the source code right in the text window, recompile and run the simulation again. This highly interactive design and testing environment is the major benefit of the Trend language and its simulator.

5.1 A preliminary example

Before we consider the details of the Trend language, let us look at a simple example of the Trend language first. The following is the famous "game of life" rule expressed in the Trend language[3]. Here life is a field (and the only field) of a particular cellular automata model which is defined by the template information the simulator loads during starting up.

```
default life=life;          // default is no change for cell values
int count;                  //declare an integer variable 'count'
nbr y;                      //declare a neighbor variable 'y'
count=0;                    // initialize counter to zero
```

[2]That is, their value changes when rotated with the rules.

[3]The game of life rule states that an active cell will be born if it has exactly three active neighbors, that an active cell will keep active if it has two or three active neighbors, and that an active cell will die with less than two or more than three active neighbors.

```
over each other y:          // count the number of active neighbors
    if (y:life) count++;
if (count<2 || count>3)     // the death rule
    life=0;
if (count==3)               // the birth rule
    life=1;
```

The statements of the Trend language are executed in order from the viewpoint of **a single cell**, just like in C. The fact that different cells can follow different rules of the same Trend rule set makes up the parallelism of cellular automata programming. In the beginning *default* denotes that the statement after it will be a *default* statement, such that if no rule is applicable for the current cell, the default statement will be used to determine the next state value for that cell. Here it just states that everything stays unchanged if no applicable rule is found. Normally the default rule is used to catch all "left over" conditions of the rule definition.

After that, *int* and *nbr* are used to declare two variables, `count` and `y`. One is used to store integer values and the other to store a neighbor position index. The rule starts with an initialization operation to set the counter to zero, and then accumulates the number of active neighbors. Finally two rules are used to determine the two value change situations: birth and death. If none of the birth or death rules is applicable (say, if `count` equals to 2), the default rule will be used.

Comments on the rule set can be marked by either the delimiter `//` or the `/*` and `*/` pair. Anything after `//` until the end of line will be ignored by the compiler as a comment. Similarly, anything enclosed between `/*` and `*/` will be ignored by the compiler too, which can include several lines. Note that nested comment pairs are not allowed in Trend.

The Trend language utilizes a strictly one-pass compiler to speed up the compilation process. Since it is built within a highly interactive cellular automata simulator, the user should not need to wait for compilation, and a one-pass compiler facilitates this. Because of the strongly one-pass compiler, all variables and functions must be declared before their use. Normally the program starts with variable and function declarations, followed by the main rules. The default rules can be put anywhere within the program, as long as they obey the same "declare before use" restriction for variables and functions. Usually default rules are put at either the beginning or the end of the program.

5.2 Reserve Words, Names, and Variables

Just like ordinary programming languages, the Trend language has its own set of reserve words for language constructions. These reserve words cannot be used for any other purpose within the language. Reserve words will be displayed using a special *slanted font face* in this article in order to distinguish them from the other language elements. Reserve words in the Trend language are:

if, int, nbr, fld, rot, default, over, void, each, else, while, other, break, return.

Unlike other programming languages, Trend has a special set of semi-reserve words called *names* which are defined not in the compiler itself, but in the simulator template information loaded during each invocation of the simulation program. They can also be defined on-the-fly by the user using the cellular automata template design window provided by the simulator and saved for future uses.

These semi-reserve words are the field and neighbor names for the corresponding cellular automata template. These names will be displayed in this article using a sans-serif font face in order to distinguish them from the other language elements. Some possible names are

North, East, South, West, ne, se, sw, nw, fieldA, component, life.

Finally, the user can define temporary storage space in the Trend language as variables. These variables can store temporary computational values, neighbor or field indices, etc. Variables are displayed in this article using a `typewriter-like font face` in order to distinguish them from the other language elements. Some example of variables are

`x, y, z, count, a, b, c, from, to, pos, layer.`

5.3 Data types

There are three data types in the Trend language. One is an integer type which is common in other programming languages, but the other two are special types used only in the Trend language. Types are not interchangeable among each other. However, a user can explicitly write code to map values in one data type to values in another data type by a sequence of if-else statements. See Section 5.5 for an example.

- *int* is a positive integer type which can be stored in cellular automata cells to represent cell states. In fact, cellular automata cells can accept only data of this type. During every epoch each cellular automata cell is expected to get a new value for each of its fields. Otherwise a runtime error will be reported. See Section 5.11 for details about runtime errors. Symbols, such as 'O', 'L', '>', etc., can be defined in the simulator model template for values of the integer data type which can later be used in the language to represent those integer values. These symbolic literal values are converted to integers by the compiler during compiling time. See the following section about data objects for details about literal values.

- *nbr* is a special type which is used to denote neighbor positions, like north, south, east, west, etc. When combined with the *fld* data type they can uniquely specify a particular field within a particular neighbor cell.

- *fld* is a special type which is used to denote *fields* within each cellular automata cell. The concept of *fields* came from the book by [Toffoli & Margolus, 1987]. Basically, the original bit depth of a cellular automata cell (say, 8 bits) is functionally divided into different *fields* (say, 2, 2 and 4 bits each) such that each field encodes different meanings and functions (to the human rule writer). The utilization of field division greatly simplifies cellular automata rule programming, and makes the resulting code much more readable.

Data of type *int* can be manipulated and compared just like normal integer values in other programming language. Data of type *nbr* and *fld* are more restricted; they cannot be used with mathematical operators since it is meaningless to add two neighbor indices together, for example. They can be compared by equal '==' and unequal '! =' operators only.

5.4 Data objects

Like all major programming languages, the Trend language provides different data objects for programming convenience. Data objects are the basic constituents of expressions, and can be literals, variables, neighbor references, array elements, or function calls.

5.4.1 Literals

Literals are defined by their face values. Their values are constant in the program, and cannot be put into the left hand side of an assignment statement. There are two basic formats of literals: the numeric format and the symbolic format. Symbolic format is defined in the simulator template for the cellular automata model being used. For example

```
0, 1, 2, 99, 'O', '>', '>,2', '>,1:s', North:, :fieldA.
```

are all literals[4]. Here `'O'` represents the *int* type value of the symbol "O" defined in the *current field*[5], which is defined by the current program context. It could take the numeric value 5 if "O" is the sixth symbol defined for the current field (starting from zero) in the model template. The literal '>,2' denotes the second rotation value of the weak rotational base symbol ">". Thus, if the numeric value of ">" is 6, then '>,2' will assume the value 8 (rotation is counted clockwise). '>,1:s' denotes the first rotation value of the weak rotational base symbol '>' for the field s. Note that symbolic format and numeric format are interchangeable: we may use 5 and 8 in replace for 'O' and '>,2', and vice versa. However, the symbolic format generally provides more descriptive information to the reader of the cellular automata program. In addition, only the symbolic format will be rotated in a rotated *if* statement. See Section 5.10 for details. But symbols must be defined in the cellular automata model template before they can be used to represent integer values.

North: and :fieldA are literal values for *nbr* and *fld* data types. North: is a neighbor constant of the *nbr* data type and :fieldA is a field constant of the *fld* data type, provided they both are defined in the loaded cellular automata template in the simulator. Note that there is no numerical format for *nbr* and *fld* data type literals. Note also the mandatory symbol ":" **after** a *nbr* literal name and **before** a *fld* literal name to distinguish them from the other data types. For example, fieldA denotes the value of the field named "fieldA" in the center cell, which is a reference to the value of "fieldA" (thus not a literal) and has the type *int*, but :fieldA denotes the field index of the field "fieldA", which is a literal and has the *fld* data type. Thus, the value of :fieldA can be assigned to a *fld* type variable, say `slice_ptr`, by the following assignment statement:

```
:slice_ptr = :fieldA;
```

The actual value of fieldA can then be indirectly accessed by using the *fld* variable `slice_ptr`, which contains a pointer to the actual value stored in "fieldA". On the other hand, if the assignment statement is written in the following way (assuming that `slice_value` is an *int* type variable), a **copy** of the value of "fieldA" is made to *int* variable `slice_value`, instead of a pointer to it. This new copy is therefore independent of the value stored at "fieldA".

[4] Assuming, of course, that the neighbor "North", fields "fieldA" and "s", the strong rotational symbol "O" and the weak rotational symbol ">" are all defined in the current cellular automata template.

[5] See the following paragraphs for the explanation of *current field*.

```
slice_value = fieldA;
```

The syntax of the four different formats of literals is the following:

```
:field_name                    // fld type literal
neighbor_name:                 // nbr type literal
's[,n][:field_name]'           // symbolic int type literal
n                              // numerical int type literal
```

where `s` stands for a symbol defined for the current field in the cellular automata model template and `n` stands for an integer number. Note that symbols '[' and ']' denote optional parts of the literal and are not part of the literal. Therefore, both the rotation count `[,n]` and the field designation part `[:field_name]` of the symbolic *int* literal can be omitted. The rotation count, if used, must be either 1, 2, or 3.

The field designation part (e.g., the "`:s`" in `'>,1:s'`) of a symbolic literal is needed only when an effective *current field* cannot be resolved by the compiler. Normally if the literal is used together with a field name in an assignment or boolean expression the compiler can extract the current field information from the program context, and thus an explicit field designation part is not needed. For example, in the following code segment examples an explicit field designation is not needed within the literal since they have been specified implicitly somewhere before the literal object:

```
fieldA='O';   //assign the symbolic literal 'O' to field fieldA
s='>,2';      //assign the symbolic literal '>,2' to field s
component='L';  //assign the symbolic literal 'L' to component

//if fieldA value in the north neighbor is equal to 'O', set
//the variable 'count' to 0.
if (North:fieldA=='O') count=0;

//if the southwest component field equals '>,1', set current
//component value to '>,1' too.
if (sw:component=='>,1') component='>,1';
```

A specification like "`fieldA='O:fieldA'`" is acceptable although it is redundant to specify the field designation tag fieldA twice, both implicitly in the assignment target and explicitly within the literal quotations.

However, if a literal is used together with a variable or array element name where the intended field is not obvious from the context so the compiler has no way to figure out the current field, the designation part will be needed within the literal quotation marks, as shown below (assuming `x, y, z` are all variables). Compare this to the examples above.

```
x='O:fieldA';
y='>,2:s';
z='L:component';
if (x=='O:fieldA') count=0;
if (z=='>,1:component') z='>,1:s';
```

The compiler will report errors when a current field is not available from the context and the field designation part is not given in a symbolic literal either, such as in the statement "`x='O';`".

5.4.2 Variables

Variables are named storage places to hold different data values. Their values can be changed by an assignment statement. They are usually used to hold test results, flags, or temporary calculation results. Their values are undefined if used before being initialized by either an assignment statement or an initialization operator in declarations.

5.4.3 Neighbor references

Neighbor references are given in the format

nbr : *fld*

where *nbr* can be any valid *nbr* data type object like a literal, variable, array element or function call. Similarly, the *fld* can be any valid representation of the *fld* data type. This whole structure denotes one particular field within one particular neighbor cell and the value there is referenced with the *int* data type. The "*nbr* :" part can be omitted all together. In that case, the default neighbor is the center cell.

5.4.4 Arrays

A chunk of storage space can be allocated at once and referenced using the array index within brackets "[" and "]". Only one dimensional arrays are supported in Trend. The array index must be of type *int*, but the array itself can be any of the *int*, *nbr* and *fld* types. For an N element array the array index runs from 0 to N−1. No runtime checking of array index bound is offered by the simulator. The value of an out of bound array reference is undefined, and an assignment to an out of bound array can crash the simulator at present.

5.4.5 Function calls

Function calls are formed by giving a declared function name together with a list of actual arguments to the function, separated by commas. Those actual arguments can be expressions or data objects (e.g., other function calls). A function returns a value in its declared data type. Calls to a function which returns a value must be done within an expression; a function cannot be used as a procedure unless it is declared as a procedure with the *void* data type, which means that no value is returned. A procedure which does not return a value cannot be used in expressions. Instead, a procedure call is a stand alone statement itself.

A function returns values by giving an expression to the *return* statement within the function body. The *return* statement will immediately terminate the execution of the current function and return the value of its argument to the caller routine to be used in an expression.

5.5 Declaration and initialization statements

Variables, arrays and functions must be declared before used. Variable and array declarations consist of a type name, followed by a list of variable or array names separated by commas, and ending with a semicolon. The difference between a variable and an array lies in the fact that an array has an index declaration marked by the "[" and "]" symbols. An optional initialization part can follow each variable or array name which assigns initial values to the variable or array.

For example, all of the following declarations are valid:

```
int i, j, k; // int type variables i, j and k
nbr from, to, where; // nbr type variables from, to and where
fld a, b, c; // fld type variables a, b and c

// declare an int array buf[] with ten elements, an int array
// x[] with two elements, and an int variable yy.
// Initialize them too.
int buf[10]={ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
    x[2]={ 'O:layerA', '>,2:layerB'}, yy=99;

// declare a nbr array permute[] with five elements,
// initialize it to the five neighbor values.
nbr permute[5]={ Center:, East:, South:, West:, North: };

// declare fld variables tag and trace and initialize them
fld tag = :layerA, trace = :layerB;
```

Note that a field designation is usually needed when initialization with a symbolic literal value like 'O:layerA' since the compiler has no way to know which field the symbol "O" belongs to. There can be many "O" symbols in different fields, each having different values. But within each field all symbols used must be unique.

Function declarations consist of a type name, followed by a function name, a left parenthesis, formal arguments (if any) separated by commas, a right parenthesis, and finally the function body. The function body can be just a single statement or a block statement. See Section 5.8.2 for details on block statements.

The following is an example of a typical function declaration. This particular function maps a neighbor position to an integer value which is treated as a pointer in the "d" data field, as mentioned in Section 5.3. The symbol `'V'` is used to **visually** denote how neighbors should point to the center cell. Therefore, the north neighbor should point to south with `'V'`, the east neighbor should point to west with `'V,1'`, etc. Because there is no diagonal arrow symbols in the usual character set, the symbol `'Q'` is used for similar purposes for the diagonal neighbors northeast, southeast, southwest and northwest. The extra "leg" of the letter Q is used as a directional arrow. Therefore, the northwest neighbor should point to southeast as denoted by `'Q'`, and the southeast neighbor should point to northwest as denoted by `'Q,2'`.

```
int ntod(nbr x)
    if (x:==no:)  return 'V:d'; // north points south
    else if (x:==ne:)  return 'Q,1:d'; // northeast points southwest
    else if (x:==ea:)  return 'V,1:d'; // east points west
    else if (x:==se:)  return 'Q,2:d'; // southeast points northwest
    else if (x:==so:)  return 'V,2:d'; // south points north
    else if (x:==sw:)  return 'Q,3:d'; // southwest points northeast
```

```
else if (x:==we:)  return 'V,3:d'; // west points east
else if (x:==nw:)  return 'Q:d'; // northwest points southeast
else return 0;
```

5.6 Mathematical expressions

Simple mathematical expressions are formed by grouping together *int* data objects using mathematical operators. When evaluated, they will generate *int* values which can be assigned to fields as next state values, or be stored in variables for further computational purposes. Mathematical expressions can be used in comparison expressions to form boolean values for use in control flow statements, or they can be arguments to a function or procedure call. A mathematical expression will never form a statement by itself; it must be part of a complete statement. Note that only *int* type data objects can be used in mathematical expressions; *nbr* and *fld* types cannot be combined with mathematical operators.

All normal mathematical operators are provided in the Trend language. Listed below are these operators in order of ascending precedence. Operators with the same precedence level will be evaluated from left to right.

```
+ -
* / % & | ^
( )
```

Parentheses have the highest precedence and can be used to change computation order. Multiplication ($*$), division (/), modulus (%), bitwise conjunction (&), bitwise disjunction (|) and bitwise exclusion (ˆ) operators are at the second priority level. Finally, the addition and subtraction operators ($+$, $-$) have the lowest priority among mathematical operators.

Note that mathematical operators always take precedence over comparison operators (e.g., $>$, $<$, $==$, etc.), which in turn take precedence over boolean operators (&&) and (||). Assignment statement operators ($=$, $++$, $--$) always have the lowest precedence among all operators.

5.7 Assignment statements

Assignment statements assign a value computed from a mathematical expression (for *int* type only) or a value from a data object (for *int*, *nbr* or *fld* types) to a variable, array element, or a field with the same data type. If the value is assigned to a field name, that value is taken as the next state value of that field and must be of type *int* .

5.7.1 Normal assignment statements

A normal assignment statement is composed of a left-hand-side, an equal sign, a right-hand-side, and a semicolon as the terminator. The left-hand-side can be in any of the three data types *int*, *nbr* and *fld*, although in the latter two cases the corresponding right-hand-side can only be simple data objects with the same data type; they cannot be expressions since *fld* and *nbr* data objects cannot form expressions. The left hand side can be field names, variables or array elements but not literals or function names.

5.7.2 ++ and −− statements

Two special assignment statements are formed by a left-hand-side, either one of the ++ or −− operators, and a semicolon. They either increase or decrease the left-hand-side object value by one, and assign that new value back to the left-hand-side. Since a mathematical operation is involved, these two special assignment statements cannot be used with data objects of type *nbr* and *fld* since increasing or decreasing their values is meaningless. The left-hand-side can be field names, variables or array elements but not literals or function names.

5.8 Control flow statements

Various common control flow constructs are provided in the Trend language to change the order of rule execution. These control flow constructs rely on a boolean expression to determine the flow pattern.

5.8.1 Boolean expressions and comparison expressions

Boolean expressions are formed by combining together comparison expressions using logical operators && and ||. && has higher precedence than || but parenthesizes can be used to change the order of evaluations. The evaluation is from left to right within the same precedence level.

Just like in the C programming language, whenever a definite result can be determined during the partial evaluation of a boolean expression, the unevaluated part of the boolean expression will be ignored since it will not change the outcome of the boolean expression in any way. For example, in the following boolean expression

```
1<2 || North:fieldA==fieldA
```

the second comparison expression never gets evaluated since the first is always true and therefore the whole boolean expression is always true. It is not advisable to put a function call in the boolean expression like this since that function call may not get executed at all!

Similarly most of the following boolean expressions will not be evaluated at all if direc is not equal to '>':

```
direc=='>' && (nw:component=='>,3' || nw:component=='0') &&
               (ne:component=='B' || ne:component=='>,1') &&
               no:component=='.'  && ea:component && we:component)
```

Comparison expressions are usually composed of two mathematical expressions or data objects separated by comparison operators. A special case of just having one mathematical expression alone in a comparison expression is taken as testing if that mathematical expression is not equal to zero. For example, a simple comparison expression[6] "a+b" is equal to a more lengthy "a+b != 0" where != means "not equal to." The comparison operators offered in the Trend language are <, <=, ==, >=, >, and !=. Note that *fld* and *nbr* data types can be combined with only == and != comparison operators since it is meaningless to compare the magnitude of these data types.

[6]This is a simple mathematic expression within part of a boolean expression and taken as a comparison expression.

5.8.2 Block statements

Statements can be blocked together using left and right braces, "{" and "}", to form a block statement. A block statement can appear wherever a single statement appears in the following control flow constructs. Thus it permits multiple statements to be put into one single control flow construct.

5.8.3 Conditioning statements

Conditional statements can take either the form

if `( boolean ) statement`

or the form

if `( boolean ) statement1` *else* `statement2`

In the first case if the boolean expression is true the statement will be executed, otherwise the whole *if* statement is skipped. In the latter case statement1 will be executed if the boolean expression is true, otherwise statement2 will be executed. Nested *if* statements are allowed and the dangling *else* problem is resolved the traditional way: associate it with the nearest *if*.

5.8.4 *over* statements

The *over* statement in the Trend language is directed toward scanning over all neighbors for a cell. It takes the following form:

over each { *other* } *nbr*`_variable statement`

In essence, it loops through all neighbors of a cell (including the center cell itself), and assigns each neighbor position index value into the *nbr*_variable. Although not required by the language itself, the *nbr*_variable will usually be referenced in the statement that follows. The optional tag "*other*" can be added to exclude the center cell in the scanning process.

For example, the following code segment determines how many neighbors are in non-quiescent states (nonzero):

```
count=0;
over each other y:
    if (y:component) count++;
```

5.8.5 *while* statements

The *while* statement in the language Trend is exactly the same as the *while* statement in C. It takes this form:

while `( boolean ) statement`

When the boolean expression is true the statement will be executed repeatedly until the boolean expression turns false. If the boolean statement is false on entry to the *while* statement the whole *while* statement will be skipped without execution of the included statement.

Note that no runtime checking for an infinite loop is provided by the simulator. Therefore, the programmer is responsible for making sure the *while* statement will terminate. This is one of the few places where a badly designed Trend program can crash the system[7].

5.8.6 *break* statements

A *break* statement can be used in the statement part of the innermost *over* or *while* statement to forcefully terminate the current looping and jump directly to the next statement after the innermost *over* or *while* statement.

5.8.7 Procedure call statements

Procedure calls are very similar to function calls except that procedures are declared as type *void* and do not return values. Therefore, they are statements by themselves with a terminating semicolon, unlike function calls which must be a part of an expression. Procedure call statements allows multiple code segments to be repeatedly used with different given arguments.

5.9 Default operations

Sometimes it is more convenient to designate *default* rules such that after the execution of all normal rules, if some fields still have not been given their next state values, the default rules can be used to make the decision. Default rules are formed by giving the tag *default* in front of any normal statement except declarations. Often the default rule is simply an assignment statement which sets the current value of a field to be its next state. It simply states that "if none of the rules changes the current value of this field, this field should stay unchanged". The default rule can be used to set whatever value normal rules can set to a field, using all language constructs provided by the language to compute the value.

5.10 The rotated *if* statement

One unique feature of the Trend language is its strong support in writing cellular automata specific rules by providing the *rotated if* command construct. The rotated *if* command is formed with a conditional statement by putting the reserve word *rot* in front of the *if* and by giving *rotatable* literal values, i.e., symbolic literals, in the boolean expression and the statement part of an *if* statement. A conditional statement formed by using the rotated *if* command is called a rotated *if* statement. Only symbolic literal values (including the *int*, *nbr* and *fld* symbolic literals) will be rotated in the boolean expression and the statement part of a rotated *if* statement. The rotated *if* statement greatly exploits the symmetrical characteristics of most cellular automata models and can cut the size of the rule set to 1/4 that size of the same rule set without using the rotated *if* statements.

[7]The other case is an assignment to an out-of-bound array element.

For example, the following function which uses standard nested *if* statements is introduced in Section 5.5. It maps a neighbor position to one of eight pointer symbols, using eight *if* statements.

```
int ntod(nbr x)
    if (x:==no:)  return 'V:d'; // north points south
    else if (x:==ne:)  return 'Q,1:d'; // northeast points southwest
    else if (x:==ea:)  return 'V,1:d'; // east points west
    else if (x:==se:)  return 'Q,2:d'; // southeast points northwest
    else if (x:==so:)  return 'V,2:d'; // south points north
    else if (x:==sw:)  return 'Q,3:d'; // southwest points northeast
    else if (x:==we:)  return 'V,3:d'; // west points east
    else if (x:==nw:)  return 'Q:d'; // northwest points southeast
    else return 0;
```

It can be replaced by using only two rotated *if* statements. In addition, the rotated *if* statements are much more obvious in meaning than the above one: they just return different weak rotational values of the base symbol `'V'` or `'Q'`, according to different input neighbor position arguments.

```
int ntod(nbr x)
    rot if (x:==no:)  return 'V:d'; // north points south
    rot if (x:==nw:)  return 'Q:d'; // northwest points southeast
    else return 0;
```

The term isotropy means something is directionally indifferent. An isotropic cellular automata rule set guarantees that it will produce the same result, properly rotated, from different orientations of the same initial cellular automata configuration. An isotropic cellular automata rule set is important since it makes the transition function of the cellular automata independent of the global orientation of the cellular automata space. Speaking in another way, no matter what specific evaluation order of a cellular automata rule set is taken, the result will be the same if the rule set itself is isotropic.

For some specific cellular automata modelings, such as the emergent self-replication cellular automata structures which will be discussed in Chapter 6, it is especially important that the same outcome will appear on the cellular automata space no matter what initial random cellular automata configuration is. We certainly cannot assume that the first self-replicating molecule on earth knew where the north pole of the earth was, neither can we assume that the first emergent self-replicating structure on the cellular automata space knew where the top side of the cellular automata space was, and acted accordingly. We will see below how the rotated *if* statement can be made to safeguard the isotropy of a cellular automata rule set.

The rotated *if* statement can be tested up to four times for the four different orientations of a reference template, depending on the boolean expression values of the rotated *if* statement. Whenever an orientation of the template makes the boolean expression value true, the statement part of the rotated *if* statement will be evaluated based on the **same** orientation, and the rotated *if* statement ends. Therefore, the outcome of a rotated *if* statement depends on the order of testing the different orientations of a template and can be non-isotropic. That is, if you turn the initial cellular automata configuration clockwise 90 degrees and run the same rule set which uses the rotated *if* statements again, you may not end up having the same results rotated clockwise 90 degrees too

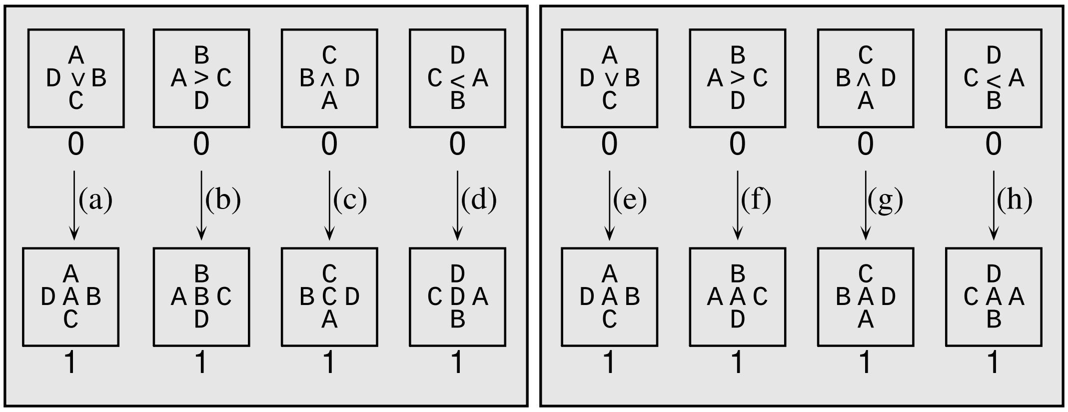


Figure 5.1: The isotropy of rotated *if* statements. Parts (a), (b), (c) and (d) are the effects of a non-isotropic rotated *if* statement on a cellular automata configuration in different orientations. Parts (e), (f), (g) and (h) are the effects of a modified isotropic rotated *if* statement on the same cellular automata configurations. We can see in the later cases that the center cell always gets the same value, no matter how the cellular automata configuration is oriented.

unless some programming precautions have been observed. It is the programmer's responsibility to ensure an isotropic rule set while using the rotated *if* statement[8].

For example, the following non-isotropic rotated *if* statement copies different values to the center cell when the cellular automata space is oriented differently, as seen in part (a), (b), (c) and (d) of Figure 5.1. Since this statement is evaluated with an upright orientation first, the value in the north neighbor always gets copied first, no matter what it is. Here the symbols 'A', 'B', 'C' and 'D' represent strong rotational symmetry states, and the symbol '>' represents a group of four weak rotational symmetry states. Note that to simplify matters, we only consider the outcome of different rotated *if* statements on the center cell; it is assumed that all the other neighbor cells stay unchanged by the effect of some other cellular automata rules not shown here.

```
rot if (no:value)
    value==no:value; // copy active neighbor values
```

An isotropic rule set can be guaranteed if one of the following two conditions is met by all rotated *if* statements used in the rule set and for all possible neighbor configurations that can occur in the cellular automata space.

- Either none of the four rotation orientations makes the boolean expression turn true; or
- Only **one** of the four rotation orientations makes the boolean expression turn true.

If there are more than one of the four rotation orientations which can make the boolean expression turn true, then only one of the orientation will be chosen to evaluate the statement part of a rotated *if* statement, thus the outcome is non-isotropy since the choices can be different for different initial configurations.

[8]But even by **not** using the rotated *if* statement, there is still no guarantee that a rule set will be isotropic. In fact, it is easier to get a non-isotropic rule set by not using the rotated *if* statement.

To make a rotated *if* statement isotropic, the user can always add (&&) a boolean factor into the boolean expression such that he or she knows that only one out of four orientations can make this boolean factor turn true, therefore ensuring that the whole rotated *if* statement is isotropic. This boolean factor can usually be a comparison of directional pointer values in a field. Since directional pointer values are unique in that they only point to **one** direction for each cell, this added boolean factor is a very good isotropy protection scheme.

For example, if the previous non-isotropic rotated *if* statement is modified as below, it becomes isotropic. It copies the same value to the center cell even when the cellular automata space is oriented differently, as seen in part (e), (f), (g) and (h) of Figure 5.1. Since this statement is evaluated according to how the weak rotational symbol '>' is oriented, the value of the neighbor at the back of the arrow always gets copied, no matter how the cellular automata space is oriented.

```
rot if (value=='>,1' && no:value)
    value==no:value; // copy active neighbor values
```

The rotated *if* statement takes the following form:

```
rot { ( alignment_boolean ) } if ( boolean ) statement
```

The "`( alignment_boolean )`" part is optional and has two uses. First, sometimes it is desirable to make the isotropy protection boolean factor stands out from the actual boolean expression in order to avoid confusing the ideas. In that case the user can move the isotropy protection boolean factor into the alignment boolean part. Second, sometimes the user knows that multiple orientations of the template will all make the boolean expression turn true (thus this rotated *if* statement is not isotropic), but he or she wants to give a particular orientation higher precedence to be used to evaluate the statement part. Rather than using the default system rotation order of the template to test a rotated *if* statement, the user can **align** the starting orientation with some pointer values by giving a pointer comparison boolean factor in the `alignment_boolean` part. Note the rotation testing direction is always clockwise and cannot be changed; only the starting orientation can be changed by the alignment part.

For example, the previous isotropic rotated *if* statement can be rewritten in the following way, which still is isotropic and functions the same. Comparing to the non-isotropic example above, the following rotated *if* statement can also be seen as giving precedence to the symbol 'A' by aligning the rotated *if* statement with the symbol '>' first before its boolean condition test begins.

```
rot (value=='>,1') if (no:value)
    value==no:value; // copy active neighbor values
```

From the example above, it may seem that if the optional alignment boolean is added, a rotated *if* statement will become isotropic automatically. This is **not** necessarily the case. For example, the following rotated *if* statement has the optional alignment boolean, and is very similar to the previous statement, but it is not isotropic. Its effect when applied to the same four cellular automata space orientations of Figure 5.1 is given in Figure reffg:nonisotropy. Again, only the effect on the center cell is considered in this figure.

```
rot (value<='>,1') if (no:value)
    value==no:value; // copy active neighbor values
```

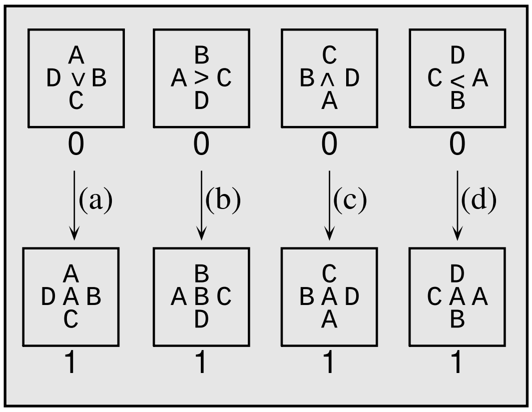


Figure 5.2: Effects of nonisotropic rotated *if* statement. The effects of a non-isotropic rotated *if* statement with optional alignment boolean on the same four cellular automata space orientations of Figure 5.1 is shown in parts (a), (b), (c), and (d).

No matter what the reason, if an `alignment_boolean` part is given for a rotated *if* statement, the rotated *if* statement will now be tested first against the alignment boolean until a true value is obtained. Then starting from the orientation which makes the alignment boolean expression true, up to **four** continuous clockwise rotations of the template will be made to test the boolean expression part until a true value is found and the statement evaluated or none is found and the rotated *if* statement skipped.

An *else* part can follow a rotated *if* statement just like a normal *if* statement. Nested *if* statements are also acceptable inside or outside a rotated *if* construct, but nested rotated *if* statements are not allowed since they are meaningless.

5.11 Compilation and runtime errors

All compilation errors will be shown on the message window when they are discovered. Since Trend is based on a highly interactive system, only the first syntax or semantic error is reported. Users can modify the code right in the text window and recompile. The user saves the final, compilation error free source code before running a simulation when using the text window to compose a Trend program. It is recommended that the user use a text editor such as `vi` or `emacs` to compose the source code, and load the finished program code into the simulator for execution only.

There are basically four kinds of runtime errors, two of which can be caught by the simulator but two of which are fatal:

- The *undefined* error. During every epoch each field in each cell must get a new next state value from the execution of the compiled rule code. If that is not the case, the simulator will report the "undefined" errors, together with the cells which have undefined fields. The undefined cells are highlighted on screen, so the user can check their field values and determine why the assignment is not complete. Most commonly the error is caused by forgetting to assign a default rule for the corresponding field(s).

- The *conflict* error. A conflict error occurs when cellular automata rules assign more than one value to a given field. This is usually caused by an inconsistent or an inaccurately partitioned rule set. This error is usually a hint that some reasoning in the cellular automata rule design is not correct. The checking for runtime conflict errors can be disabled by using the main simulation window. This speeds up the simulation process somewhat since the evaluation process will stop whenever all fields have been given next state values; the rest of the unevaluated cellular automata rules will be ignored. But it is recommended that this feature be disabled only after cellular automata rules have been completely tested and all possible (conflict) errors have been fixed.

- The infinite loop error. When a *while* statement does not terminate during the evaluation process, the infinite loop error occurs. This error is currently not automatically resolved. The user has to abort the execution of the simulation program when this error happens. A timeout feature could be added to the simulator in future revisions to cure this problem.

- The memory fault error. An assignment to an out-of-bound array element will cause this error. The out-of-bound array element can actually reside in the compiled code area, the data area, or the simulator program area itself. The effect of this error is totally unpredictable and it is not checked for in the current simulator. The Trend programmer must be careful not to reference or assign to an out-of-bound array element.

5.12 Examples

Several simple examples are given in this section to show typical uses of some of the Trend language constructs. More complex examples can be found in the following two chapters.

5.12.1 A simple reaction-diffusion system simulation

A reaction-diffusion system is a set of chemical reactions where several catalysts are competing and/or cooperating with each other in a circular manner. For example, in a three catalysts system, catalyst A can help produce catalyst B but inhibits catalyst C, catalyst B can help produce catalyst C but inhibits catalyst A, and catalyst C can help produce catalyst A but inhibits catalyst B. If a mixture of these three catalysts is put together, spiral waves consisting of the three catalysts catching each other's tail will usually occur in the mixture through time.

The reason the spiral wave is forming is because whenever there is a region with high density of catalyst A, catalyst B will be catalytically produced in that region shortly by the help of A. Since catalysts are not only produced and destroyed by chemical reactions, but are also translated by the diffusion process, the concentrated region of B seems to follow the concentrated region of A in the mixture. This will make a spiral wave distribution pattern in the long run. A well-known reaction-diffusion system is the Belousov-Zhabotinsky reaction [Keener & Tyson, 1986].

The following Trend rule set is obtained directly from a CAM Forth program in Section 9.3 of the book [Toffoli & Margolus, 1987]. It is a good example showing that the CAM Forth language can be easily translated into the Trend language. This cellular automata rule set tries to catch the spirit of the reaction-diffusion reaction in a very simple manner. There is only one self-annealing reactant in this simplified system. The existence of a reactant is represented by the one bit field `value`. During each iteration, a cell first scans neighbors to accumulate density information. The Moore neighborhood is used in this rule set. The `alarm`bit is set according to the conditions in the

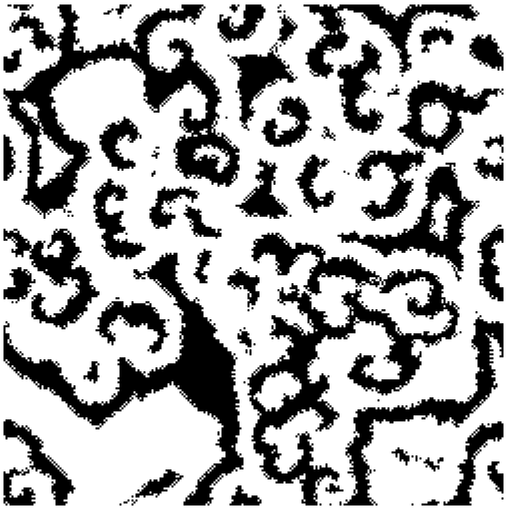


Figure 5.3: A simple reaction-diffusion system simulation. The cellular automata space (250 by 250 cells) is setup by randomly initializing the three data fields value, alarmand counter, which puts individual cells at different phases of the rule set. This picture is taken after about 800 iterations of the simulation.

array table[]. The cell will set the alarmbit when there are over three active neighbors, or when there are exactly two active neighbors. The alarmbit then triggers the counter to countdown, which disables the cell's valueuntil it becomes zero.

Each cell in the cellular automata space is running independently according to this rule set. Because of the interaction between cells, in Figure 5.3, we can clearly see the formation of spirals after the simulation has run a while starting from a randomly distributed initial cellular automata configuration. For more detailed simulations of this sort with more catalysts, the cellular automata rules in [Boerlijst & Hogeweg, 1991] can be used to replace the very simple rules here.

```
/* the alarm condition table */
int table[]={ 0, 0, 1, 0, 1, 1, 1, 1, 1};

int sum; /* the variable used to accumulate neighbor density */
nbr y; /* the dummy variable used in the scanning (over) statement */

/* The default is no change in all fields */
default value=value;
default alarm=alarm;
default counter=counter;

/* scanning neighbor density */
sum=0; /* set accumulator to zero */
over each other y: /* loop through all neighbors */
        if (y:value) sum++; /* if active, increase sum by one */

/* alarm value is set according to the table */
alarm=table[sum];

/* value is reset if counter is counting, otherwise it is set */
if (counter==0)
        value=1;
else
        value=0;
```

```
/* counter is set to 3 if alarm is set and the cell is active */
if (value && alarm)
        counter=3;
else if (counter) /* otherwise, counter counts down toward zero */
        counter--;
```

5.12.2 Backward compatibility

Sometimes it may be desirable to run some old cellular automata rule sets in the new simulator using the Trend language. As said before, many previous cellular automata rule sets are encoded in a tabular format, which defines the cellular automata transition function using numerous (domain, next_state) pairs. Although it is not recommended that cellular automata rules be written this way with the Trend language, the language does have the capability to easily adopt such a table into its rules, so that old rule sets can still be run without too many modifications.

The following Trend rule set implements a well-known self-replicating rule set using tables [Langton, 1984]. The transition function domain values, each of them consisting of the center, north, east, south and west neighbor values in a von Neumann neighborhood, are listed in the table `domain[]`. The corresponding next state values are listed in the table `next[]`. The core of this rule set is actually very short and simple; it just loops through the table trying to find a matching domain value for the current cell, and then sets the next state value accordingly when such a matching domain value can be found on the table. The looping rules are independent of the table and can be used with many other tables to implement different old cellular automata rule sets. The only thing that needs to be modified is the table length constant, which is 207 for Langton's table [Reggia *et al.*, 1992].

The looping rules may seem inefficient at first glance, since they always sequentially scan through the table to find the matching domain value; they could be replaced by some clever algorithms, such as hashing or binary search, to do the search. Since the simulator has its own fast lookup caching mechanism, it is actually **not** necessary to do any clever table search in the rule set itself. Within one full replication cycle (151 epochs for Langton's loop), the simulator will have all table information in its own cache. After that, cell evaluations will no longer need to go through the compiled rule code but can obtain the next state values directly from the simulator cache table.

This rule set correctly implements Langton's self-replicating loop. The result is given in Figure 5.4. Note that here we choose to use the actual state values instead of symbolic states to represent the loop in order to facilitate comparison with the tabular rule set below. Other than using different symbols to represent states, this Langton's loop is exactly the same as the one in Figure 2.4 at page 8.

```
/* the domain table, in Center, North, East, South and West order */
int domain[]={00000, 00020, 00220, 20210, 20272, 20202, 20212, 20242,
20042, 20120, 12702, 72021, 02127, 12420, 42021, 02124, 42201, 20024,
27220, 21022, 10212, 17202, 11212, 22271, 11272, 22211, 22000, 01722,
71120, 12221, 20001, 00030, 20270, 20342, 30002, 00023, 20720, 72012,
03214, 24122, 22277, 07721, 12210, 20122, 22200, 10027, 12402, 12211,
24220, 12227, 72220, 20007, 12271, 21722, 00012, 10001, 00001, 01002,
10024, 41120, 22244, 04421, 10021, 11121, 12124, 11127, 12224, 42220,
20004, 20312, 30012, 13221, 13224, 20302, 30042, 43220, 20112, 10012,
```

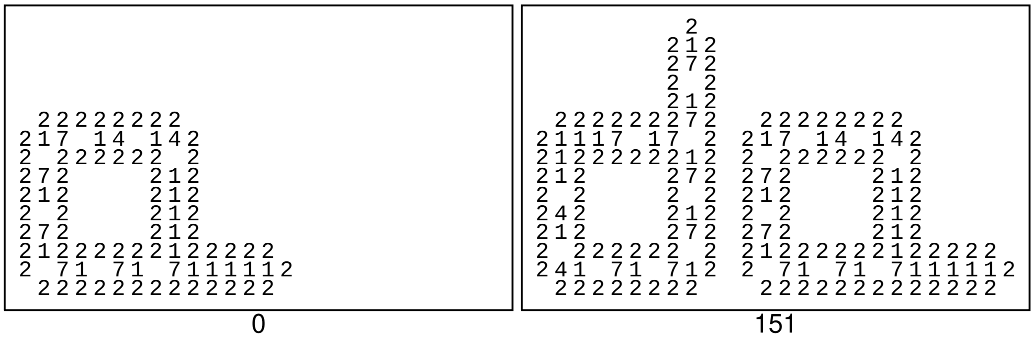


Figure 5.4: Langton's self-replicating loop. The initial loop at epoch 0 produces a replica of itself in epoch 151.

```
20172, 20102, 20712, 00202, 01020, 02220, 22202, 22212, 27020, 22272,
11232, 02320, 31020, 23202, 11027, 02021, 01120, 12020, 22152, 22261,
72520, 52027, 01625, 62120, 12026, 25202, 26202, 12527, 52221, 22105,
02621, 25002, 00302, 22057, 07521, 52020, 00205, 07512, 25001, 25021,
22062, 10226, 10232, 02321, 31220, 22003, 23002, 22067, 07621, 62000,
00006, 06002, 20057, 00032, 10127, 30001, 00052, 02517, 50020, 12327,
00050, 20552, 50022, 02527, 52240, 25025, 45202, 22077, 72320, 30007,
10542, 50021, 25024, 00062, 12621, 60001, 20742, 72502, 50027, 07214,
25020, 10510, 11240, 12512, 20510, 50052, 12542, 55021, 00070, 00720,
70070, 00027, 77202, 10070, 12324, 42320, 30004, 30062, 00013, 12624,
63121, 10006, 26002, 20227, 00522, 50230, 35102, 12670, 62121, 26227,
72501, 52127, 12425, 25220, 20520, 52022, 02125, 52120, 12115, 25120,
24221, 22103, 22162, 11126, 52121, 11125, 22032, 21261, 62212};

/* the corresponding next state table of the domain table above */
int next[]={0, 0, 0, 2, 2, 2, 2, 2, 3, 2, 7, 0, 1, 4, 0, 1, 0, 2, 2,
2, 1, 7, 1, 2, 7, 2, 2, 1, 0, 1, 2, 0, 2, 2, 2, 0, 2, 0, 1, 2, 2, 1,
1, 2, 2, 7, 4, 1, 2, 7, 1, 1, 7, 2, 2, 1, 2, 2, 4, 0, 2, 1, 1, 1, 4,
7, 4, 1, 2, 2, 3, 1, 4, 2, 1, 1, 2, 1, 2, 2, 2, 0, 2, 0, 2, 2, 3, 2,
1, 2, 1, 1, 0, 5, 0, 6, 2, 2, 5, 2, 1, 0, 6, 0, 2, 5, 0, 2, 1, 0, 0,
5, 1, 2, 0, 1, 2, 2, 2, 3, 7, 1, 0, 6, 2, 2, 1, 1, 3, 2, 5, 0, 7, 3,
5, 5, 2, 7, 0, 1, 5, 1, 4, 2, 0, 2, 1, 6, 7, 5, 2, 2, 1, 1, 2, 1, 2,
1, 2, 1, 4, 1, 7, 0, 7, 2, 1, 2, 7, 2, 0, 7, 4, 6, 1, 2, 2, 7, 1, 1,
2, 2, 2, 2, 1, 7, 5, 2, 0, 2, 5, 2, 2, 0, 5, 2, 2, 2, 2, 6, 2, 1, 2,
1, 6, 1, 5};

/* counter for looping through all table entries */
int i;

/* default is no change in a cell if no applicable rules */
default state=state;

i=0; /* clear counter */
while (i<207) { /* loop through all table entries */
  /* if found a match, then get the next state value accordingly */
  rot if (domain[i]==state*10000+no:state*1000+ea:state*100+so:state*10+we:state) {
```

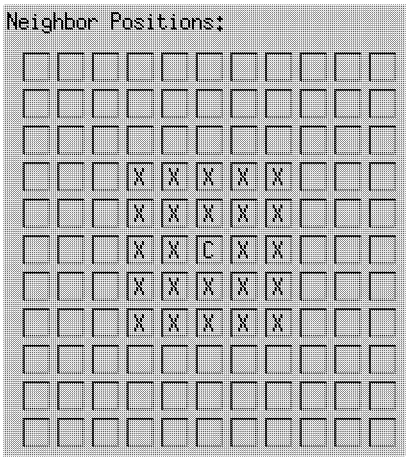


Figure 5.5: The Extended Game of Life template. In addition to the 9 standard Moore neighbors, 16 new neighbors around the original Moore neighborhood are also added to this template.

```
    state=next[i];
    break;
  }
  i++; /* increase counter */
}
```

5.12.3 Extended template Game of Life rules

It is interesting to see how the familiar Game of Life rules can be modified to use a larger neighborhood template than the Moore neighborhood. In this example, we extend the neighborhood template to include also the secondary neighbors in addition to the immediate neighbors, as shown in Figure 5.5. Each cell still carries a bit field as in the standard rules.

The construction of the Trend rule set is almost identical to the standard one we have seen in Section 5.1, the only difference lies in the condition for the death and birth rules. In this modified new rule set a cell will die if it has more than 9 or less than 6 active neighbors, and a cell will be born if it has exactly 7 or 8 active neighbors. We can see that the scanning rules are still the same as before, despite the fact that in the extended neighborhood template there are now more neighbors. This presents the power of the new simulator, which allows an arbitrary new neighborhood template to be defined and used, unlike previous simulators, which usually provide a limited number of predefined templates and cannot allow the creation of new ones. This also shows the power of the versatile *over* statement, which can easily accumulate neighbor information without even mentioning neighbor names in the statement itself.

```
int count; /* variable for accumulating active neighbor count */
nbr y; /* dummy nbr pointer to scan the neighbors */

default life=life; /* default is no change for a cell */

/* find out how many active neighbors are there */
count=0; /* clear the counter */
over each other y: /* scan all neighbors */
    if (y:life) count++; /* add one if alive */
```

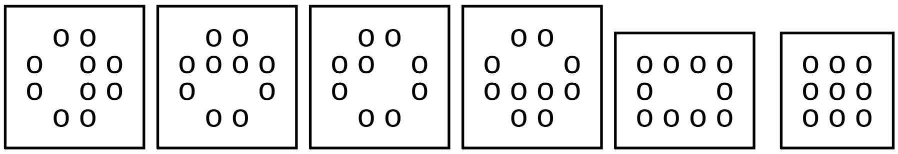


Figure 5.6: Several block patterns found in the modified Game of Life simulation.

```
/* The death rule. If there are over 9 or below 6 active neighbors, */
/* an active cell will die */
if (count<6 || count>9)
    life=0;

/* The birth rule. If there are exactly 7 or 8 active neighbors, */
/* an active cell will be born */
if (count==7 || count==8)
    life=1;
```

Running the extended template Game of Life rules on several randomly initialized cellular automata configurations immediately let us recognize several well-known patterns which have counterparts in the standard Game of Life simulations. First, some *blocks*, which are fixed patterns in the cellular automata space, can be easily identified. Several blocks are shown in Figure 5.6. Most Game of Life configurations, if not all, will reduce to several blocks in the space at the end of the simulation.

Another familiar Game of Life pattern is the *flipper*, which is actually a pair of stable patterns that are changing into each other with a period of two. One flipper found in the extended template Game of Life simulations is shown in Figure 5.7.

Additionally, three *glider* patterns, with periods 3, 4 and 5, are shown in Figure 5.8. Glider (a) is moving toward the upper right direction, with a period of 3. Glider (b), which is very similar to glider (a) but a little bit smaller, is also moving toward the upper right direction with a period of 4. The last glider, (c), moves in a horizontal direction toward the left, which actually resembles a *spaceship* in the standard Game of Life terminology. It should be noted that it is very hard to get a naturally occurring spaceship in the standard Game of Life simulation, but in the extended template Game of Life simulation, the horizontally moving gliders can be seen easily with just several simulation runs.

5.12.4 Finding the cellular automata Voronoi diagram

A Voronoi diagram among some anchor points in a two dimensional space is the collection of points in the space which are in equal distance to their two closest surrounding anchor points.

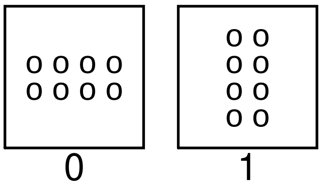


Figure 5.7: A flipper in the extended Game of Life simulation. These two patterns repetitively change into each other in a fixed cellular automata space location.

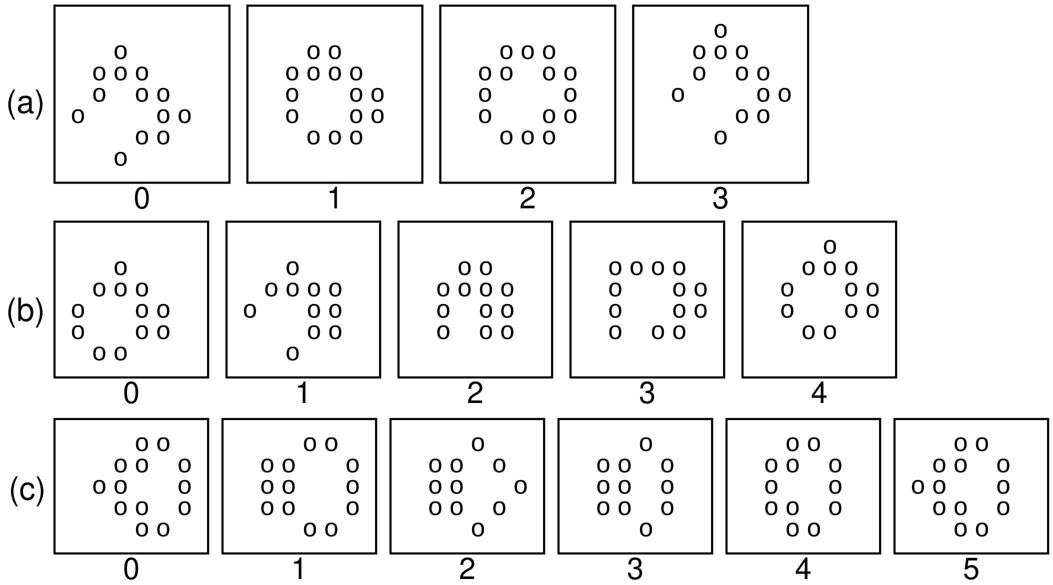


Figure 5.8: Some gliders in the extended template Game of Life simulation. (a) A period 3 glider moving toward the upper right corner. (b) A period 4 glider moving toward the upper right corner, too. (c) A period 5 glider moving toward the left horizontally.

Therefore, the Voronoi diagram seems to divide the space into regions centered around those anchor points. For example, the Voronoi diagrams for two, three, four and five randomly chosen anchor points are shown in Figure 5.9.

In this subsection we will see how a Voronoi diagram for the cellular automata space can be constructed using cellular automata rules. Note that distances in a cellular automata space are computed using the L_∞ metric, i.e., $dist(p, q) = \max(|p_x - q_x|, |p_y - q_y|)$. First let us see how it works in Figure 5.10. In epoch 0, some random points are set in a cellular automata space of 250 by 250 cells. The cellular automata rule set makes an expanding square wave go out from each anchor point, as seen in epoch 5 and all the following epochs. When waves from two anchor points collide, they will from a Voronoi segment at the crash points. Since waves are expanding at the same speed from both anchor points, we know that points on the Voronoi segment are in equal distance to both anchor points. The collection of all Voronoi segments between anchor points forms the Voronoi

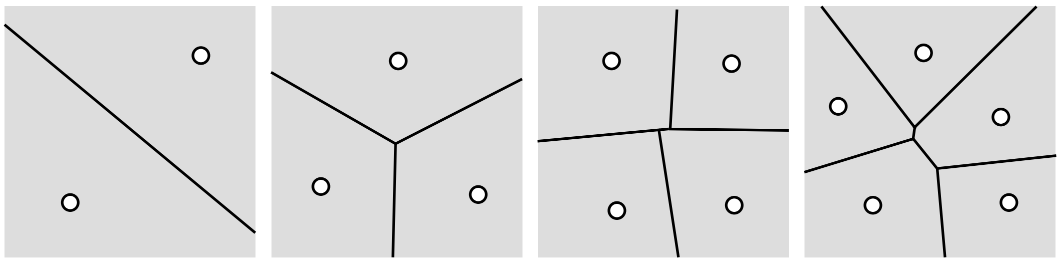


Figure 5.9: The Voronoi diagram. A Voronoi diagram divides the space into regions centered around individual anchor points. The diagram itself is represented by line segments in the space. The Voronoi diagrams for two, three, four and five anchor points are shown.

diagram for the cellular automata space[9]. One way to examine if the resulted Voronoi diagram is correct is to see if all of its segments are connected together. If there is any unconnected segment, the Voronoi diagram is not fully constructed or maybe incorrect. In the following discussion this criterion is used to check the correctness of the automata rule set.

We can see in Figure 5.10 that gradually, each wave claims a portion of the cellular automata space for its anchor point, and the free space of the cellular automata space is shrinking. By epoch 40, the Voronoi diagram has been fully established, and nothing will change after that.

The following is the rule set for constructing the Voronoi diagram of a cellular automata space with random anchor points. This cellular automata rule set references the Moore neighborhood template and uses only one data field, "value". Eight weak rotational symmetric states are used to denote the wave. Three strong symmetric states are used to denote the anchor point, the Voronoi point, and the anchor point after epoch 0. The symbol '>' and its rotated forms are used to represent the four weak rotational symmetric states denoting an expanding square toward quadrilateral directions. The symbol 'Q' and its rotated forms are used to represent the four weak rotational symmetric states denoting an expanding square toward diagonal directions. Recall from Section 5.5 that the extra "leg" of the letter 'Q' is used to denote diagonal directions, due to the lack of more suitable arrow symbols in the usual character set. The symbol '*' is used to denote the anchor point. The symbol '%' is used to denote the Voronoi point. The symbol 'S' is used to denote the same anchor point after epoch 0.

Sometimes it is easier to explain the rules using accompanying pictures. There are ten cases shown in Figure 5.11 which will be used to illustrate the rules in the following comments surrounding the rules.

```
nbr y;   // variable used for scanning neighbors
int sum; // variable counting incoming waves toward a quiescent cell
int ptr; // variable storing the new direction of a quiescent cell

default value=value;  // default is no change to any cell

/* Function ntod() maps neighbor positions to the eight weak
rotational symmetric states, which determine the direction a wave is
expanding. From the point of view for a quiescent cell, this function
is used to determine if some nearby waves are moving toward
itself. This function has also been discussed in the section about
rotated if statement in this chapter. */
int ntod(nbr x)
    rot if (x:==we:) return '>:value';
    else rot if (x:==nw:) return 'Q:value';
    else return 0;

if (value==0) {    // rules for quiescent cells

/* The enclosed rules determine the next state value for a quiescent
cell. The idea is to scan the neighbors of the quiescent cell in order
```

[9] Note that the cellular automata Voronoi diagram is different to the geometric Voronoi diagram due to different distance metrics being used. In cellular automata space expanding square waves are used to find the Voronoi segments; in geometric space the segments can be viewed as found by expanding cycles.

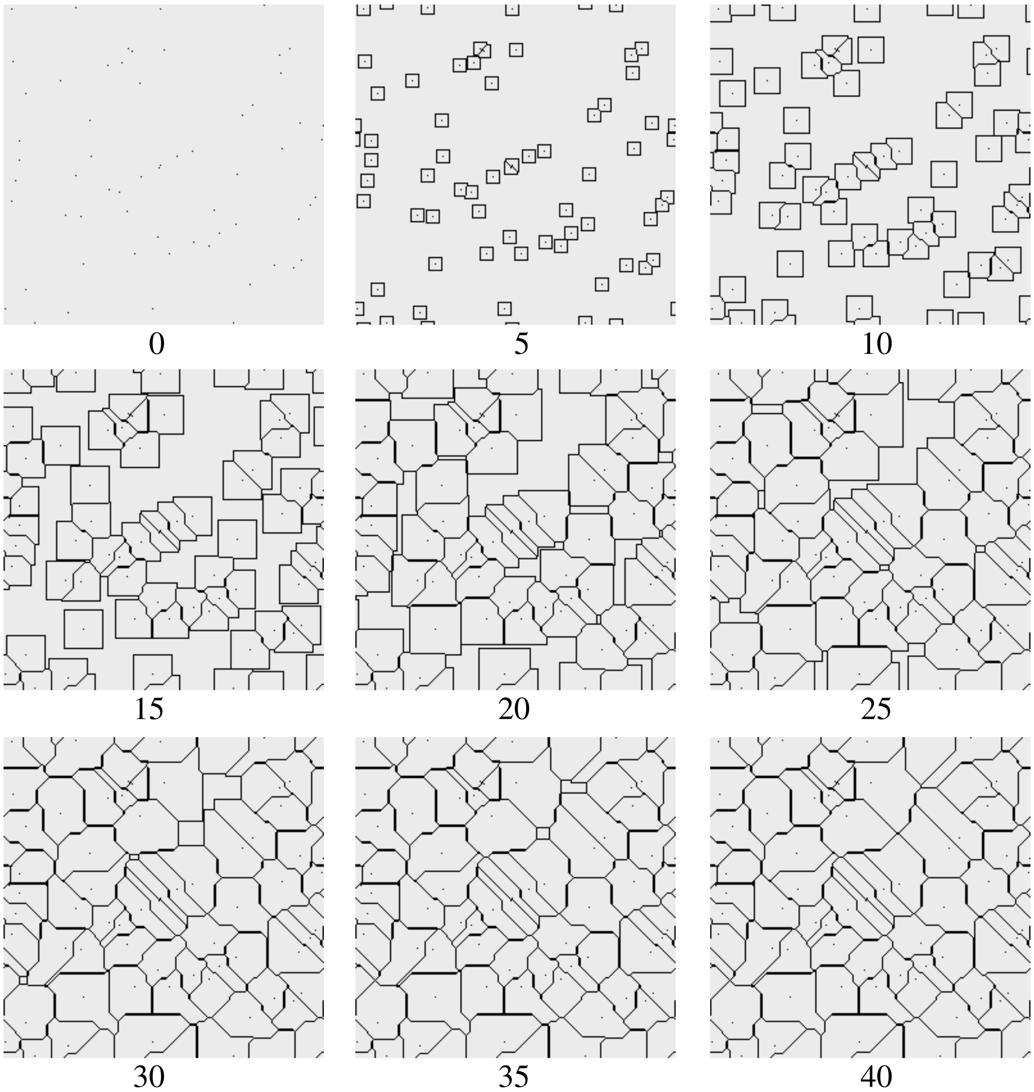


Figure 5.10: The cellular automata Voronoi diagram. The Voronoi diagram is found by making expanding square waves out of each anchor point. During initialization some random anchor points are spread into the 250 by 250 cellular automata space. Each cell is represented by a pixel in this figure. Note that we only differentiate active and quiescent states in this figure. As usual, the cellular automata space is wrapped around the four sides in a torus shape. Active cells are drawn with dark color while quiescent space is shown in light gray color. In epoch 5 we can see that small squares are expanding out from anchor points. Some squares have already collided, forming tiny Voronoi segments. In all the following epochs, we can see that those squares are getting bigger and bigger. They keep expanding unless they are stopped by collisions with the other squares, in that case, permanent Voronoi segments are formed. The Voronoi diagram is gradually shaped up. Finally, in less than 40 epochs, the Voronoi diagram for this cellular automata space is fully constructed.

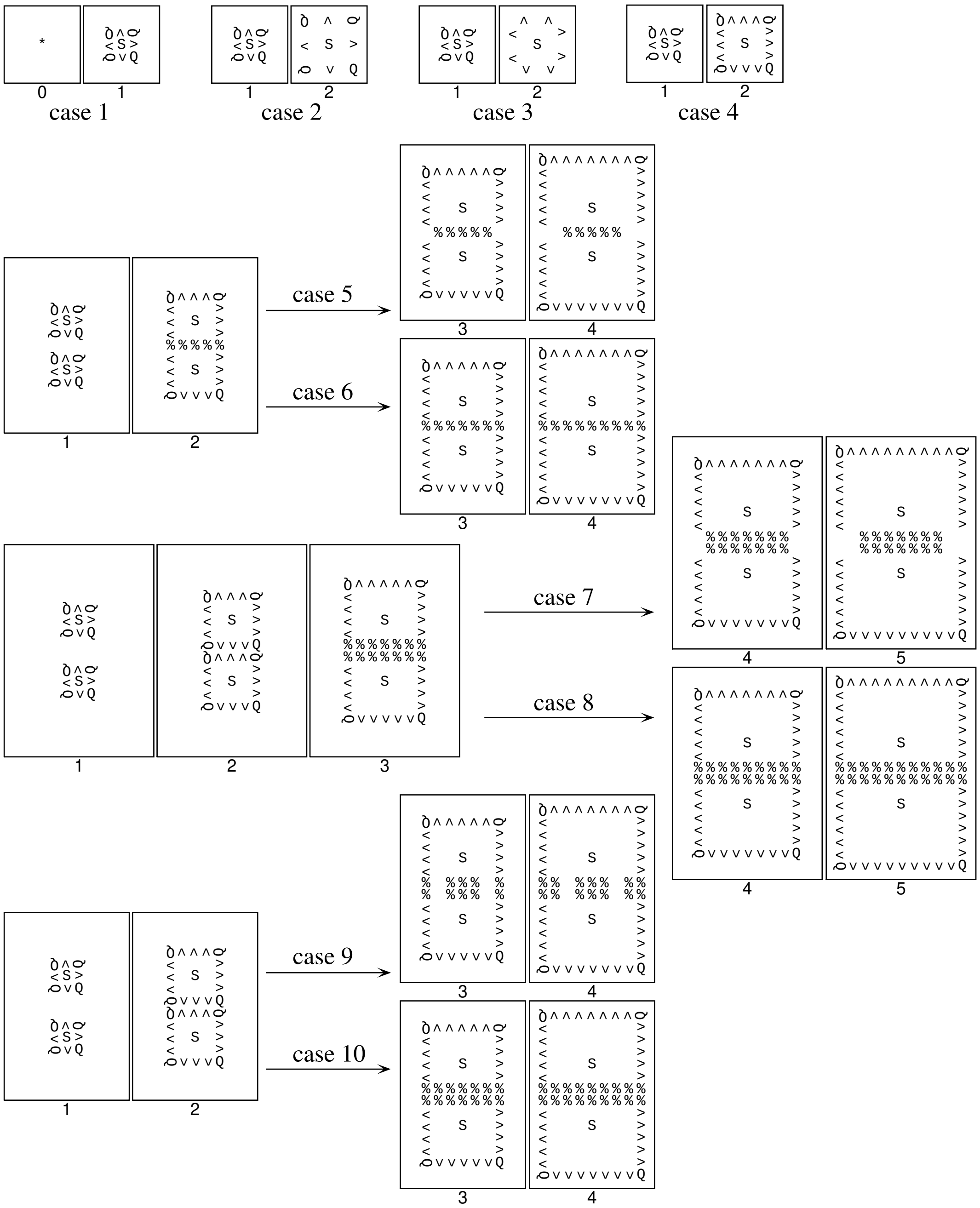


Figure 5.11: The role of the cellular automata rules. Ten cases are illustrated in this figure to show the purpose of various cellular automata rules. Refer to the comments in the rule set for details.

```
to determine if there is any wave moving toward it. The variable 'sum'
records how many of such waves are found. If it is more than 1 after
the scanning process, a collision of waves occurs, and the quiescent
cell must be converted to carry the Voronoi point state '%'. If there
is only one incoming wave, the quiescent cell will be converted to
carry one of the eight expansion pointers, depending on which
direction the wave is coming from. The new pointer value is
temporarily stored in variable 'ptr'. */

    sum=0; // clear the incoming wave counter

    over each other y: { // scan neighbors for incoming waves

        /* This rule starts the expanding wave around an anchor point,
        as seen in case 1. Whenever there is an anchor point
        '*' around a quiescent cell, that quiescent cell will be
        converted to an expansion pointer in the next epoch. Note that
        the anchor point '*' will be immediately converted to a
        nonfunctional state 'S' by the following rules in order to
        prevent multiple expanding waves from the same anchor
        point. */
        if (y:value=='*') {
            sum++;
            ptr=ntod(y:);

        /* This rule keeps expanding the wave, as seen in case
        2. It is very similar to the rule above except that the
        direction of the wave also has to be verified. The
        function ntod() is used again to determine if a wave is
        directing toward the quiescent cell. */
        } else if (y:value==ntod(y:)) {
            sum++;
            ptr=y:value;

        /* This rule keeps expanding the wave, but at the corners. See
        case 3 for its effects. Note that the pointer of the neighbor
        does not actually point at the quiescent cell. This rule, when
        combined with the previous one, makes a continuously expanding
        square, as seen in case 4. */
        } else rot if (y:==no: && (y:value=='Q' || y:value=='Q,1')) {
            ptr='>,1:value';
            sum++;
        }
    }

    /* After the scanning, if there is only one incoming wave, set the
    new expansion pointer accordingly. */
    if (sum==1)
        value=ptr;

    /* Otherwise, there is a collision. This quiescent cell should
```

```
    be part of a Voronoi segment. Set the Voronoi state '%' instead */
    else if (sum>1)
        value='%';

    /* This rule keeps the Voronoi segment expand after it is
    formed. There are three special cases in which the Voronoi
    segment has to expand after it is formed in order to ensure
    connection with the other Voronoi segments. The first case is when
    two waves meet at exactly the same row or column of quiescent
    cells. That will generate a single line Voronoi segment. But the
    problem is, this Voronoi segment will not be connected to the
    other segments if it does not keep expanding toward its two
    ends, as seen in case 5. The first line of the OR'ed conditions
    makes sure the Voronoi segment is still expanding, as seen in case
    6. The other two OR'ed lines of conditions work similarly, but for
    the double lines Voronoi segment which is formed when two
    expanding waves reach each other at the same time. The behavior of
    a double lines Voronoi segment with and without these two lines of
    conditions are shown in case 8 and 7, respectively. */
    else rot if (we:value=='%' &&
            (nw:value=='>' && sw:value=='>' ||
             nw:value=='%' && sw:value=='>' ||
             nw:value=='>' && sw:value=='%'))
        value='%';

} else if (value=='*') // rules for anchor points

    /* The anchor point should always change to something else after
    epoch 0, or multiple waves will come out of the same point. */
    value='S';

else rot if (value=='>') { // rules for quadrilateral pointers

    /* A pointer cell in the expanding wave will return to the
    quiescent state in the next epoch, unless there is a collision, as
    in cases 7, 8, 9 and 10. In these cases, it changes instead to a
    Voronoi point '%' due to the collision. */
    if (ea:value)
        value='%';
    else
        value=0;

} else rot if (value=='Q') { // rules for diagonal pointers

    /* Similarly, the pointers at the four corners of a square
    will return to the quiescent state unless they collide with other
    waves. This rule has extra conditions to make sure two expanding
    waves will not cross each other without changing their corners
    to the Voronoi point state, as shown in case 9. The corrected
    behavior when the two extra OR'ed conditions are added, is shown
    in case 10. */
```

OPCODE	VALUE	LEFT	RIGHT	TRUE	FALSE

Figure 5.12: A virtual machine instruction. Each instruction of the Trend virtual machine has six fields, as shown.

```
    if (se:value || ea:value=='Q,1' || so:value=='Q,3')
        value='%';
    else
        value=0;
}
```

5.13 The virtual machine code

We have seen several examples of the Trend language rule sets in this chapter. It is time to see how the source rules are converted by the compiler to the virtual machine code which the simulator actually uses to compute the cellular automata transition function. As explained in the previous chapter, a virtual machine instruction is represented by a `ParseNode` record, which can be viewed as a machine instruction having 6 fields as shown in Figure 5.12.

The OPCODE field encodes the machine instruction operator. The VALUE field contains any immediate value if used by the OPCODE. The LEFT and RIGHT fields contain the address of the operands for the OPCODE. They may not both be used for a unary OPCODE. The operands are data fetching machine instructions by themselves. The fields TRUE and FALSE are used for control flow. They store the address of the next instruction to be executed after the current instruction. For most OPCODE's which do not generate a boolean outcome, the TRUE field is followed by the instruction. For a boolean OPCODE, the TRUE or the FALSE field is followed depending on the boolean outcome.

Valid OPCODE's and their meanings are listed in Table 5.1. In addition to these OPCODE's, there will be an extra "PDUMMY" code which is used only in the following code listing to represent temporary storage used by the compiler. It is not actually part of the virtual machine instruction set.

To facilitate explanation some tags are used as examples in the table. Here "`field`" means a cellular automata data field name, "`fvar`" means a field variable name, "`nvar`" means a neighbor variable name, "`var`" means a data variable name, "`array`" means an array name, "`expr`" means an arithmetic expression, and "`nbr`" means a cellular automata neighbor name. In addition, VALUE means the VALUE field in the machine instruction.

The best way to know how the Trend source rules are translated into the virtual machine code is to look at an example. We choose the reaction-diffusion rule set introduced in Section 5.12.1 as the example. After all original comments are stripped off, this rule set is reproduced below for easy reference. The region comments are added to associate part of the rule set to portions of the translated code. See below.

```
int table[]={ 0, 0, 1, 0, 1, 1, 1, 1, 1};
int sum;
nbr y;
// ---------------------- region F
default value=value;
```

MNEMONIC	OPCODE	Meaning of the instruction
PFIELD1	1	data field access of the form “`field`”
PFIELD2	2	data field access of the form “`fvar`”
PFIELD3	3	data field access of the form “`nvar:field`”
PFIELD4	4	data field access of the form “`nvar:fvar`”
PADDR	5	variable access, VALUE has the variable address
PARRAY	6	array access, VALUE has the array address
PADD	7	arithmetic addition operation
PSUB	8	arithmetic subtraction operation
PMUTL	9	arithmetic multiplication operation
PDIV	10	arithmetic division operation
PAND	11	bitwise AND operation
POR	12	bitwise OR operation
PXOR	13	bitwise Exclusive OR operation
PVALUE	14	immediate data access, VALUE has the data
PCALL	15	subroutine call operation
PROT	16	rotated if rotate operation
PROTE	17	rotated if end of rotation operation
PBREAK	18	direct jumping operation, usually used in loops
PSET1	19	variable assignment of the form “`var=expr`”
PSET2	20	array assignment of the form “`array[var]=expr`”
PPLUS1	21	variable increment of the form “`var++`”
PPLUS2	22	array increment of the form “`array[var]++`”
PFSET	23	assignment to the data field, field index in VALUE
PFPLUS	24	increment to the data field, field index in VALUE
PDEF	25	beginning of default rules, a no return jump
PNEQ	26	boolean testing operation “not equal”
PNZE	27	boolean testing operation “not zero”
PGT	28	boolean testing operation “greater than”
PGEQ	29	boolean testing operation “greater or equal”
PEQ	30	boolean testing operation “equal”
PLEQ	31	boolean testing operation “less or equal”
PLT	32	boolean testing operation “less than”
PRETURN	33	subroutine call return operation
PROTS	34	rotated if start of rotation operation
PRVALUE	35	immediate data access with rotation, for “`nbr:`”
PFMINUS	36	decrement to the data field, field index in VALUE
PMINUS1	37	variable decrement of the form “`var--`”
PMINUS2	38	array decrement of the form “`array[var]--`”
PRFIELD1	39	data field access of the form “`nbr:field`”
PRFIELD2	40	data field access of the form “`nbr:fvar`”
PMOD	41	arithmetic modulo operation

Table 5.1: Virtual machine instructions and their meanings.

```
default alarm=alarm;
default counter=counter;
// ----------------------- region A
sum=0;
// ----------------------- region B
over each other y:
        if (y:value) sum++;
// ----------------------- region C
alarm=table[sum];
// ----------------------- region D
if (counter==0)
        value=1;
else
        value=0;
// ----------------------- region E
if (value && alarm)
        counter=3;
else if (counter)
        counter--;
```

The compiler generated code from the rule set above is listed below. It is a direct memory excerpt after being properly disassembled and formated. Various OPCODE's are converted to their mnemonic labels to facilitate reading. As said before, the PDUMMY label represents temporary storage used by the compiler, which is not part of the instruction set, and is neither used nor referenced by the machine code itself. Entry point at the end denotes the starting machine instruction with which an evaluation should begin.

```
ADDR:   OPCODE  VALUE   LEFT    RIGHT   TRUE    FALSE
-------------------------------------------------------
600584: PDUMMY  0       0       0       0       0
600608: PDUMMY  0       0       0       600584  0
600632: PDUMMY  1       0       0       600608  0
600656: PDUMMY  0       0       0       600632  0
600680: PDUMMY  1       0       0       600656  0
600704: PDUMMY  1       0       0       600680  0
600728: PDUMMY  1       0       0       600704  0
600752: PDUMMY  1       0       0       600728  0
600776: PDUMMY  1       0       0       600752  0
600800: PVALUE  0       0       0       0       0
600824: PDUMMY  0       0       0       0       0
600848: PFIELD1 514224  0       0       0       0
600872: PFSET   68      600800  600848  600968  0
600896: PVALUE  1       0       0       0       0
600920: PDUMMY  1       0       0       0       0
600944: PFIELD1 514240  0       0       0       0
600968: PFSET   89      600896  600944  601064  0
600992: PVALUE  2       0       0       0       0
601016: PDUMMY  2       0       0       0       0
601040: PFIELD1 514256  0       0       0       0
601064: PFSET   112     600992  601040  0       0
601088: PADDR   625168  0       0       0       0
```

```
601112: PVALUE  0       0       0       0       0
601136: PSET1   126     601088  601112  601208  0
601160: PADDR   625208  0       0       0       0
601184: PVALUE  1       0       0       0       0
601208: PSET1   147     601160  601184  601280  0
601232: PPLUS1  147     601160  0       601280  0
601256: PVALUE  9       0       0       0       0
601280: PGEQ    0       601160  601256  601520  601376
601304: PADDR   625208  0       0       0       0
601328: PDUMMY  0       0       0       0       0
601352: PFIELD3 514224  601304  0       0       0
601376: PNZE    0       601352  0       601424  601232
601400: PADDR   625168  0       0       0       0
601424: PPLUS1  166     601400  0       601232  0
601448: PVALUE  1       0       0       0       0
601472: PADDR   625168  0       0       0       0
601496: PARRAY  625172  601472  0       0       0
601520: PFSET   175     601448  601496  601616  0
601544: PDUMMY  2       0       0       0       0
601568: PFIELD1 514256  0       0       0       0
601592: PVALUE  0       0       0       0       0
601616: PEQ     0       601568  601592  601688  601760
601640: PVALUE  0       0       0       0       0
601664: PVALUE  1       0       0       0       0
601688: PFSET   210     601640  601664  601832  0
601712: PVALUE  0       0       0       0       0
601736: PVALUE  0       0       0       0       0
601760: PFSET   225     601712  601736  601832  0
601784: PDUMMY  0       0       0       0       0
601808: PFIELD1 514224  0       0       0       0
601832: PNZE    0       601808  0       601904  602048
601856: PDUMMY  1       0       0       0       0
601880: PFIELD1 514240  0       0       0       0
601904: PNZE    0       601880  0       601976  602048
601928: PVALUE  2       0       0       0       0
601952: PVALUE  3       0       0       0       0
601976: PFSET   258     601928  601952  602120  0
602000: PDUMMY  2       0       0       0       0
602024: PFIELD1 514256  0       0       0       0
602048: PNZE    0       602024  0       602096  602120
602072: PVALUE  2       0       0       0       0
602096: PFMINUS 288     602072  0       602120  0
602120: PDEF    0       0       0       600872  0

Entry point: 601136
```

The TRUE and FALSE fields for all instructions always refer to instruction addresses within the code listing itself. The LEFT and RIGHT operand fields also refer to data fetching instructions within the code listing. The VALUE field for some data fetching instructions, nevertheless, may contain addresses of pre-declared variables, arrays, or data fields within neighboring cells. These addresses are not within the code listing addressing space. Instead, they are allocated elsewhere.

For this particular simulator session, we find the following variable or field addresses assignments: `int table` at 625172, `int sum` at 625168, `nbr y` at 625208, `value` at 514224, `alarm` at 514240, `counter` at 514256. Note that the PFSET instruction assigns new values to cellular automata data fields, and it uses a logical index number instead of absolute memory addresses to refer to fields. The index numbers are 0 for `value`, 1 for `alarm` and 2 for `counter` for this particular rule set.

It is easier to view the code listing in a graphical format rather than read it off line by line on the code listing. For that purpose the code listing is manually converted into a graph in Figure 5.13. The OPCODE labels and VALUE fields are retained in this graph, but the operand references and control flow directions are converted to arrow lines for better clarification. Other than these conversions, this graph is exactly the same as the code listing above. The graph has also been divided into six regions related to the original Trend source rule set, also marked with region comments. These regions are enclosed by dotted lines in the graph. By comparing instructions in each region with the corresponding source rules, it is much easier to understand how the compiler translates source rules into individual virtual machine instructions.

Region (A) is translated from the first assignment statement in the source rule set. It sets zero to the variable `sum`. Region (B) is translated from the *over* statement in the source rule set. Note that machine instructions to assign 1 to the variable `y` and to increase the value of `y` are added by the compiler; they are not prescribed in the original source rule set. Region (C) is translated from the array assignment statement, region (D) is translated from the first if-else statement group in the source rule set, and region (E) is translated from the second if-else-if statement group. Finally, region (F) is translated from the default rules.

5.14 Comparison to other cellular automata languages

The Trend cellular automata programming language introduced in this chapter contains many new cellular automata specific language constructs which are not found in other cellular automata programming languages. The rotated *if* statement is a major invention in the Trend language, which fully exploits the rotational symmetry of the cellular automata space, and therefore potentially reduces the size of an isotropic cellular automata rule set to only 1/4 the size it would be without using the rotated *if* command. The Trend language and its associated cellular automata simulator have been carefully designed to support developing cellular automata rule sets which are both isotropic and conflict free. Preventive features, like the conflict catching and reporting mechanism in the evaluation module, and the aligned rotated *if* statement, cannot be found in other languages, including CAM Forth and the *Cellang* language used in the *Cellular* system.

The Trend language presents a standard C like syntax which is familiar to most users and easy to use. This is unlike CAM Forth, which uses a postfix notation that sometimes can be hard to understand. In the Trend language all neighbor and field names automatically become reserved words in the language once they are defined in the "Template Design" window. These reserved words are given special treatment by the compiler and can be quite handy in describing cellular automata rules at a higher conceptual level. Trend is also unlike the *Cellang* language in the *Cellular* system, which can only use relative indexing to describe neighbors (which can be hard to recognize sometimes).

Trend allows the definition of symbolic literal values. Such symbols are used both to display the cellular automata space on screen and to represent individual states in the Trend rule set. Therefore, the direct correspondence between what the user writes in the rule set and what he

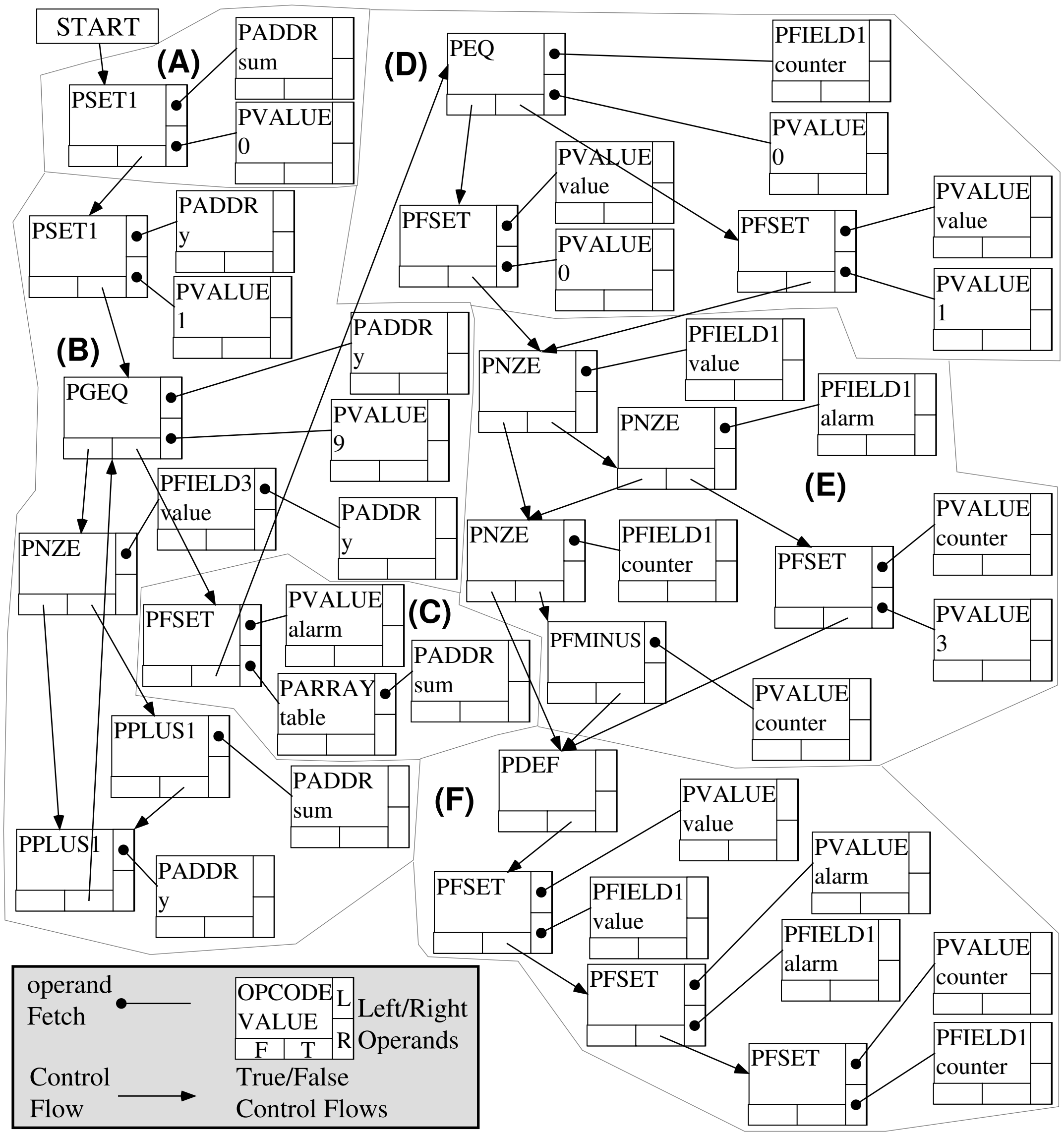


Figure 5.13: A graphical view of the compiler generated code. Regions enclosed in dotted lines are associated with statements in the original Trend source code.

sees on the screen greatly simplifies efforts to recognize and correct rule set errors. Even more, the use of symbolic literals in the rotated *if* statement further enhances the usefulness of the rotated *if* statement. These features are also not available in other cellular automata languages.

It is worth mentioning, however, that the *Cellang* language does have some interesting features which are not available in the Trend language, such as the special `time` and `random` system variables, support of multi-dimensional cellular automata, and the object oriented *agents* mechanism. As will be discussed in the following section, it is usually not hard to implement a higher dimensional cellular automata model, but it is hard to visualize its contents on screen. Therefore, higher dimensional support will not be added into the Trend system before a suitable visualization method can be devised. The other extra features of *Cellang* generally do not belong to the standard cellular automata model, but are extensions to the standard cellular automata model by the *Cellular* system. When designing the Trend language and its associated simulator, a strong desire has been put into following the standard model of a uniform, time indifferent and self-contained cellular automata space. Therefore, those extra features of *Cellular* were neither required nor pursued in this work.

5.15 Discussion

The Trend cellular automata programming language was designed from the ground up with two hopes in mind: to be as powerful as possible for cellular automata specific programming needs, and to be as similar to the C programming language as possible. For cellular automata rule set design we need specific language constructs to exploit the symmetry and regularity of the cellular automata space, yet such features are not available in general purpose programming languages. By being as similar to the C language as possible, we can ensure that users do not have to spend too much time learning the Trend language since the C or C++ programming languages are familiar to most people nowadays. It has to be noted that although the Trend language is made to be similar to C, its compiler is different from a C compiler due to the fact that cellular automata are parallel systems but C programs are generally sequential. For this reason the Trend compiler was implemented completely from scratch.

The Trend language currently supports three data types: *int*, *nbr* and *fld*. The *nbr* and *fld* are special data types used to denote neighbor positions and fields, so the only data type which can be accepted by a cell is the integer data type *int*. The integer data type is the most natural data type for cellular automata cells, and is the only data type needed by the two major cellular automata rule set developments presented in the following two chapters. Although it is not obvious at this moment that other data types are needed, adding floating number data type to the Trend language is planned for its next revision. Actually, since all data types are stored in the bit fields in a cell, it does not matter to the cellular automata model if those fields represent integers or floating pointer numbers. It is just how the cellular automata rules manipulate those data fields that makes a difference. Having more data types available to the programmer can certainly help in designing cellular automata rule sets with great flexibility.

The one-pass compiler of Trend currently imposes a strict, no-forward-reference requirement on the rule set. Although this is not a limitation to the power of the Trend language, it may be inconvenient for some occasions. This restriction will also be addressed in the next revision. The compiler can be modified by either adding some forward reference resolving data structures and keeping the one-pass only style, or by changing to a two-pass compiler. The first choice is certainly better than the second one since the speed of the compiler is very important to the interactive user.

The Trend compiler generates a set of virtual machine code currently. The benefit of using the virtual machine code rather than the real machine code is portability, which has been mentioned before in the previous chapter. Running the virtual machine code requires another layer of interpretation by the evaluation module of the simulator, which can slow down the simulation speed up to three times. Although this speed penalty of using the virtual code is not a major issue when compared to the benefit of being able to port the simulator easily to other major computer platforms without the need to modify the compiler at all, it may become an issue when simulation needs are increased. If speed becomes very important, the code generation module of the compiler can be modified to generate actual machine code directly. Another solution is to make one more pass through the virtual machine code by the compiler to convert it to the native machine code of the host computer.

The Trend language is a general purpose cellular automata programming language. For any general purpose programming language, the potential application of the language is often beyond the imagination of the language designer. Even familiar users of the language still occasionally discover new usages of the language. It is possible that some clever Trend language programmers can discover many new applications in the future. A library of useful Trend language applications can be maintained to avoid re-engineering them by other new Trend language programmers.

Chapter 6

Emergent Self-Replicating Cellular Automata Structures

It is desired in this research to be able to discover a cellular automata rule set which allows generation of self-replicating behaviors from a randomly initialized cellular automata space. Several conditions must be met by the rule set:

- There should be no assumption about the initial composition of the cellular automata space, except that there be a reasonable number of non-quiescent cells. Of course if the simulation starts with a completely quiescent, or near completely quiescent space, the whole cellular automata space will remain or return to a completely quiescent state in a short time, and nothing interesting will appear.
- There should be a well-defined self-replicating behavior which is easily observable by the human experimenter.
- The self-replication should be non-trivial. The self-replicating process should not rely primarily upon the physics of the cellular space (i.e., the cellular automata rule set) to support its replication. Otherwise, a simple cellular automata rule which just says "copy your neighbor" will suffice to generate replicating individual cells. This is not self-replicating.
- In order to distinguish trivial self-replication from non-trivial self-replication, Langton's examination condition must be passed by the rule set [Langton, 1984]. According to this condition, a self-replication process is non-trivial if its replication process is self-directed by its own stored instructions, and during the replication process there is an easily identifiable instruction *transcription phase* and an instruction *translation phase.*

The following sections will describe a rule set which satisfies all of the conditions above. Due to the complexity of the rule set, it is more feasible to introduce the rule set one part at a time, and then put together everything at the end. Therefore, signals and data fields used by the rule set are first described, and then how signals flow and turn and why they can support the replication of loops of arbitrary sizes are explained. After that, it is shown how initial loops are formed. Finally, we will see how loops can grow in size. A listing of the complete rule set is provided in at the end of this chapter for reference.

6.1 An augmented self-replicating loop rule set

To start, the functions of the self-replicating loop rule set [Reggia *et al.*, 1993a] are used as a basic framework. They are augmented and new features are added to get what we want: an emergent self-replication rule set. Doing it this way has the advantage of working on a framework of self-replicating behaviors which have been studied thoroughly and are well-known. The Moore neighborhood template is used for the rule set.

Although designing the rule set uses reference to the functions, or "behaviors" of a previous rule set, it is still a completely new rule set since it is necessary to add at least the following features to the original function set:

- A size independent self-replication rule set. The original self-replicating loops need to have an independent rule set coded for each self-replicating loop of a different size. To allow emergent self-replicating structures to emerge and grow in size, we need a cellular automata rule set which can support the self-replicating process of any structure with any size.

- A bootstrap procedure. The original self-replicating loops start their replicating process from a template which is artificially placed in the cellular automata space at the initial stage. But now our purpose is to allow randomly formed initial cellular automata space configuration to yield self-replicating structures. Therefore, we need a logically sound and general pathway which will lead from the random initial cellular automata configuration to the first self-replicating structure in the cellular automata space.

- Enabling growth and mutation. Even after we have the pathway from random initial cellular automata configuration to the first self-replicating structure, if the self-replicating structure cannot change in size and position once formed, the whole process will still not be interesting. We need to allow existing self-replicating structures to grow and gradually change to a bigger size.

As we will gradually see in the following sections, the new cellular automata rule set does encompass all the above required features.

6.2 A running example

To make the detailed rule set description that follows easier to understand, a continuous example of the emergent self-replication rule set is shown in Figures 6.1, 6.2, 6.3 and 6.4 running in a randomly initialized 40×40 cellular automata space using an initial component density of 25%. Note that all boundaries are wrapped around so that if a signal sequence goes off the right boundary it will reappear on the left boundary and vice versa. Similarly, if a signal sequence goes off the top boundary it will reappear on the bottom boundary again.

Emergent self-replicating structures have been obtained before epoch 500 in this example (see Figure 6.1). The size of the structures has the tendency to grow bigger and bigger until the structures are too big to fit comfortably in such a small world (40×40 only). Big loops will easily annihilate each other and return themselves to monomers if they are too big. Since all dying loop cells return to monomer format and can trigger generation of loops again, this cellular automata space does not show any sign of ceasing activity even at epoch 7500. In fact, apparently it never repeats itself, so we do not know when the world will stop its evolution, even at epoch 7500.

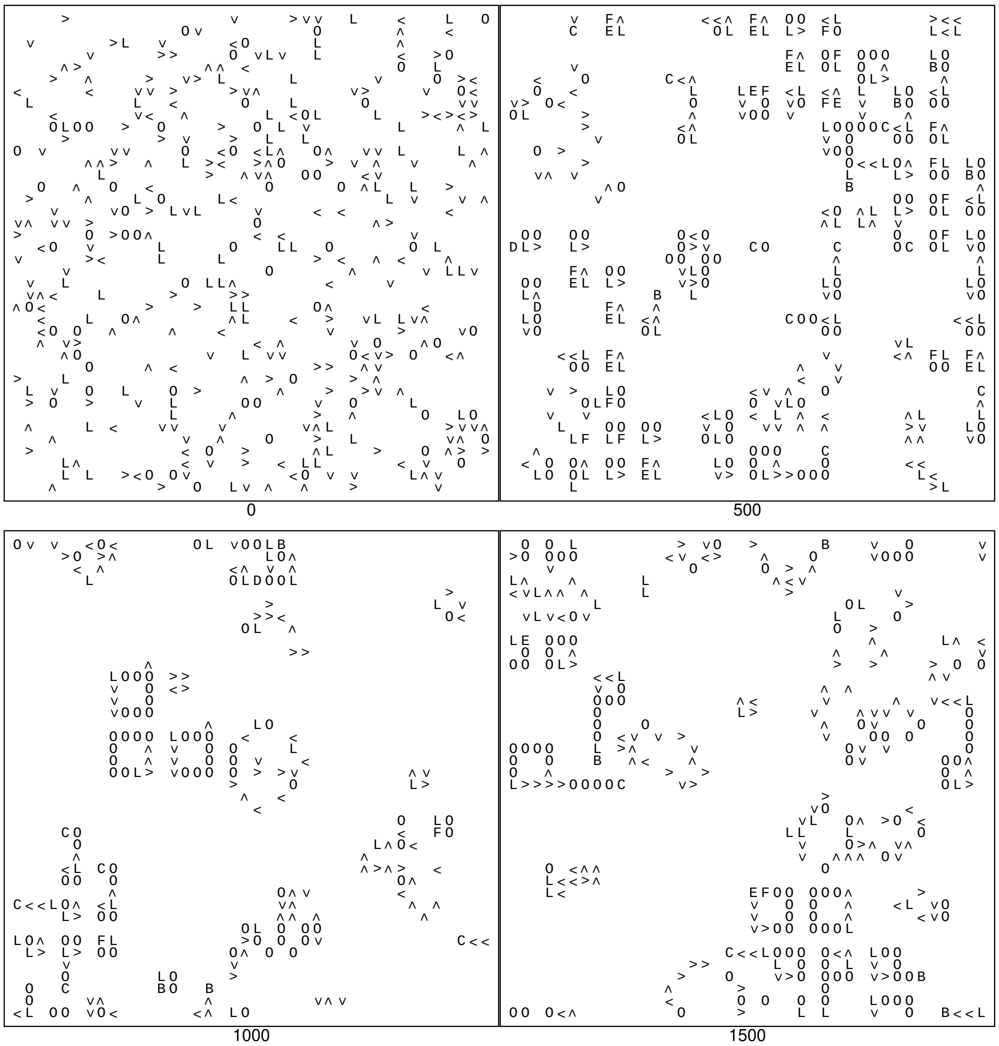


Figure 6.1: A running example (part 1). The simulation starts with a randomly allocated cellular automata space with 25% non-quiescent occupancy. By epoch 500 many 2 by 2 and some 3 by 3 loops have formed. In epoch 1000 most smaller loops are gone, replaced by some 4 by 4 loops. In epoch 1500 a 4 by 4 loop is about to generate a 5 by 5 loop in the middle left region.

In the following sections we will see how this emergent self-replication rule set is constructed. After that, the properties of the rule set and its simulation results will be discussed.

6.3 Components, signals and signal sequences

In a cellular automata space, each cell can hold a different state value. This value, if non-quiescent, constituents the "component" of a cellular automata structure. A cellular automata structure can be just a single cell, i.e., it has no connection with the other non-quiescent cells in the neighbors, and in that case we call it a *monomer*. On the other hand, a cellular automata structure can have several non-quiescent cells which are functionally connected together and behave as a

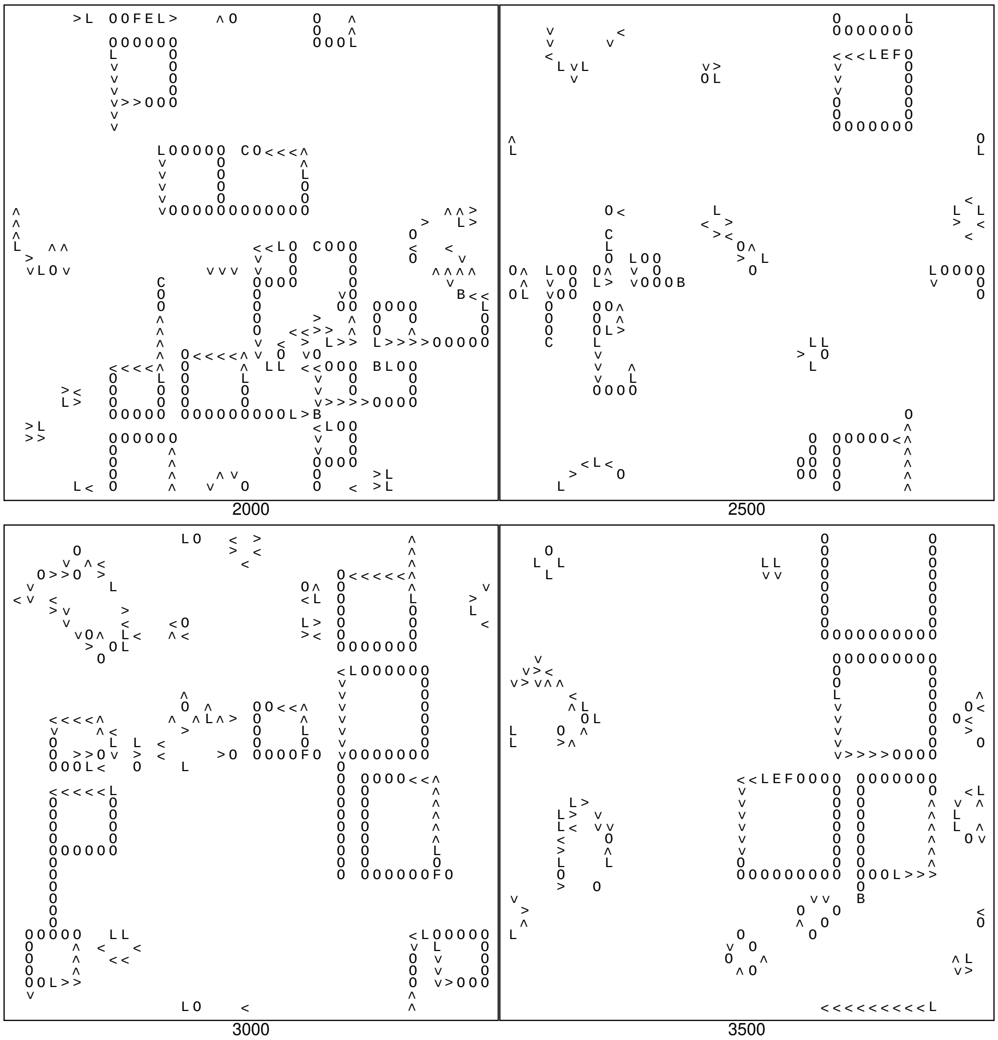


Figure 6.2: A running example (part 2). By epoch 2000 the biggest loop is now 6 by 6. In epoch 2500 that becomes 7 by 7. When the size of loops grows, the number of loops decreases. In epoch 3000 the biggest loop is 8 by 8 and it is about to generate a 9 by 9 loop. In epoch 3500 an amazingly big 10 by 10 loop can be seen in the upper right region of the space. This is 1/4 the linear size of the cellular automata space!

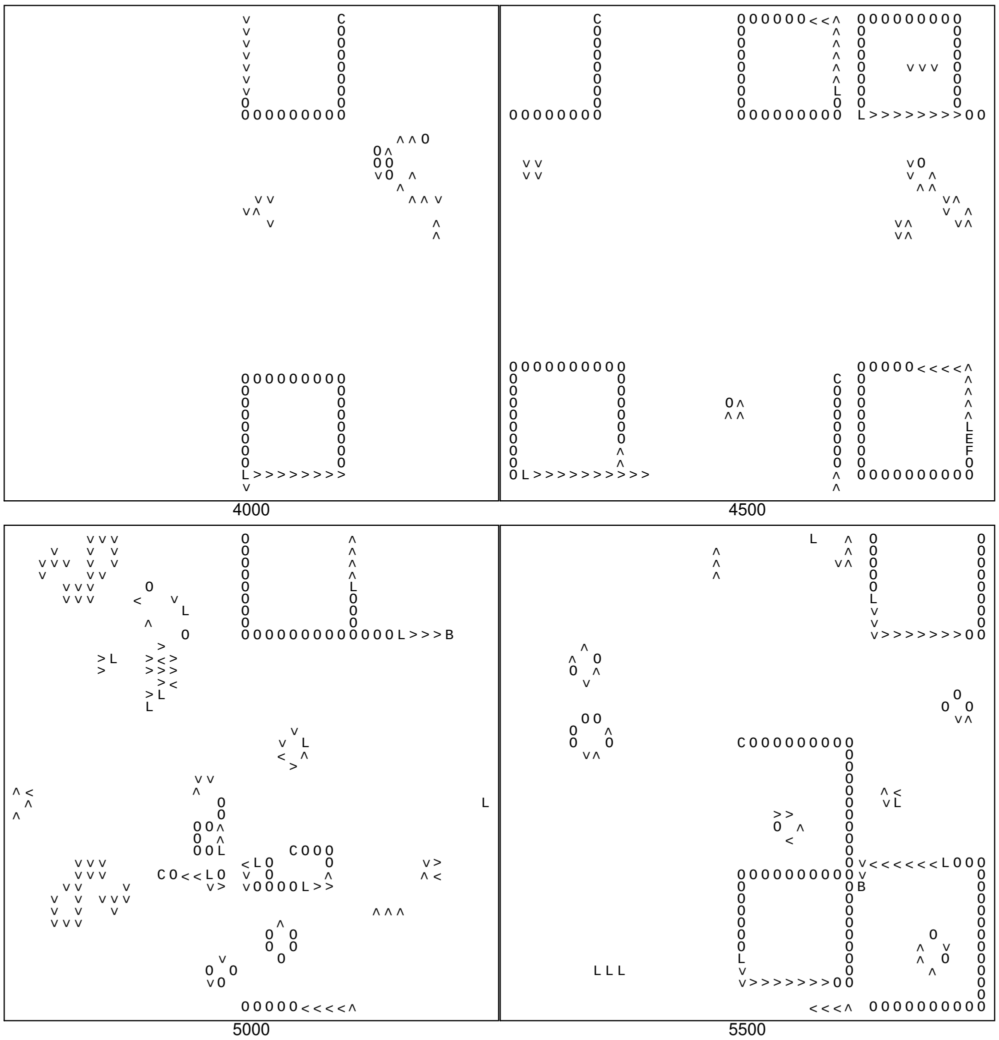


Figure 6.3: A running example (part 3). By epoch 4000 it seems that loop size has reached its limitation in this cellular automata space, so loops start to annihilate each other easily. Only one gigantic 9 by 9 loop is still replicating. In epoch 4500 10 by 10 loops are coming back again. Two such loops are in the lower left and right region of the space. In epoch 5000 again gigantic loops annihilate each other and only one 10 by 10 loop is left. Some smaller loops are in the space, generated from monomers. In epoch 5500 it seems that gigantic loops dominate the cellular automata space, but unfortunately they collide.

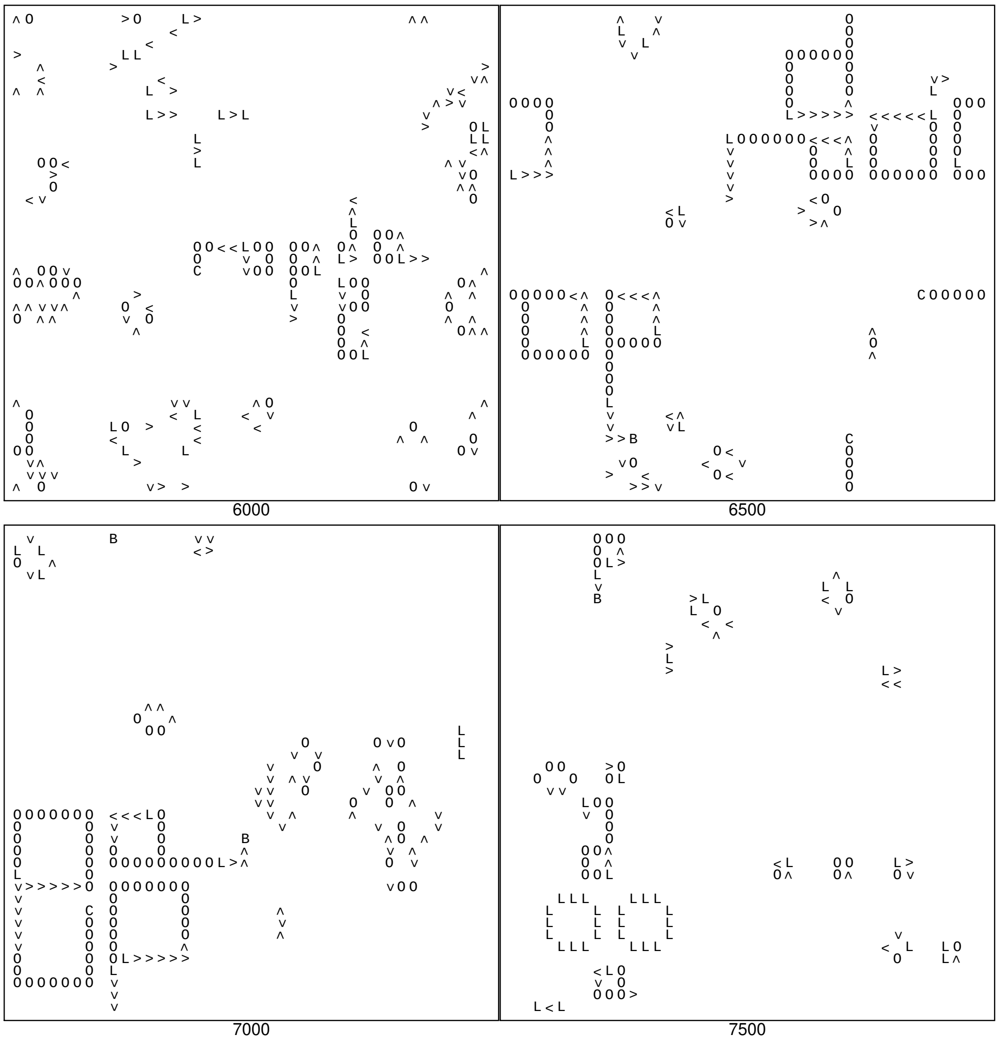


Figure 6.4: A running example (part 4). By epoch 6000 we finally see no more gigantic loops since they have all died due to space competition and mutual annihilation. We see some smaller loops resuming the growth process again. In epoch 6500 they become bigger. In epoch 7000 it seems that 7 by 7 loops dominate the space but by epoch 7500 they all die. Still, there are 3 by 3 loops in the space and we do not know when (if ever) this cellular automata space will cease its activity.

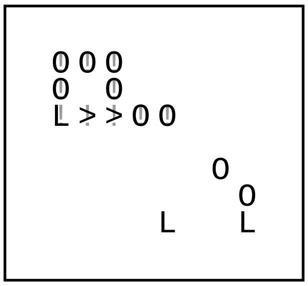


Figure 6.5: Sample cellular automata space. Quiescent states are represented by empty white space. Non-quiescent states are represented by characters. Some state values, like $>$, L, etc., are also signals. A sequence of signals becomes a signal sequence. Bound cells are defined by the bound bits which are shown in light gray color.

whole. In the latter case we call the structure a "multi-cell" cellular automata structure or simply a cellular automata structure, and we call its cells *bound cells*.

Some cell states have special meanings. Each of them will direct some specific steps during the self-replication process and is viewed as moving through the other components within the cellular automata structure body, like water flowing in a pipe. Therefore, those states are not only components of cellular automata structures but also *signals* for the self-replicating process. A sequence of signals on a cellular automata structure comprises the *signal sequence* which completely describes and controls the self-replication process.

For example, in Figure 6.5, all non-quiescent states are visible characters. Quiescent states are represented by empty white spaces. Among non-quiescent states, those which overlap a bound bit are in bound cells. They are at the upper left region of the figure. The definition of the bound bit will be given shortly in the next section. It can be seen that there are three signals in this cellular automata structure, i.e., L>>.

In the cellular automata model considered here, the following components are used:

`O > L C B E F D`

Among them $>$ `L E F` are signals. Except the $>$ signal, all the components have strong rotational symmetry. The function of each component is briefly outlined below. The actual use of each component will become clear in the following sections which detail the behaviors of the cellular automata structures.

O This is the building block of cellular automata structures. It allows passage of a signal sequence through, providing a pathway for the flow of information. Because of this it is also called the **Block O**.

> This is the extrude signal, which directs the expansion of a cellular automata signal pathway into the quiescent space. This is the only weak rotational signal used in this rule set and is actually represented by four rotational symmetric values $>$, $\vee$, $<$ and $\wedge$ which are designated in the rule set by '>', '>,1', '>,2' and '>,3'.

B This is the birth component left by the extrude signal $>$. A quiescent neighbor will first be converted into this state before it becomes a normal part of a cellular automata structure. Because of this it is called the **Birth B**.

L This is the turning signal. It changes the direction of expansion of a signal pathway by 90 degrees counterclockwise.

C This is the corner component left by the turning signal L. A signal > going through it will be rotated 90 degrees counterclockwise and form a corner. Because of this it is called the **Corner C**.

E F This pair of signals in sequence direct the branching of a signal pathway.

D This is the detachment component which separates the parent and child cellular automata structures during the replication process. Because of this it is called the **Detach D**.

Of course, there is always the quiescent state which is denoted by '.' when referenced in rules; quiescent states are shown by white space in all the figures.

6.4 Functional divisions and data fields

To ease the rule set design effort a functional division of data fields are usually used. This concept came from the book by [Toffoli & Margolus, 1987]. Basically, the original bit depth of a cellular automata cell (in our case 8 bits) is functionally divided into different *fields* (in our case four fields: 4, 2, 1 and 1 bit each) such that each field encodes different meanings and functions (to the human rule writer). The utilization of field division greatly simplifies the cellular automata rule programming effort, and makes the resulting rules much more readable.

The component field introduced in the previous section is the primary data field which accounts for most of the normal operations of the cellular automata structures. It takes four bits to encode its twelve possible state values.

There are additional data fields to encode for rare or occasional special situations in the cellular automata space or to store additional information in some cell. These additional data fields are outlined below. Their usages and purposes will be introduced in the following sections.

special This is a two-bit field which denotes special situations that arise occasionally in the cellular automata space. There are four possible cases:

- '.' No special situation.
- '*' A branching signal sequence (EF) will be generated.
- '-' A cell will not allow the signal sequence to pass through it, thus effectively deleting the signal sequence.
- '#' A bound cell is in the dissolve mode, and will become a monomer in the next epoch.

growth This one bit field, if set (denoted by '+'), records the stimulus hidden in a cell which may cause the existing signal sequence to grow in length.

bound This one bit field, if set (denoted by '!'), marks a cell as part of a multi-cell cellular automata structure, otherwise the cell is a monomer.

Again, like the components, all the functions of these additional data fields will become clear in the following sections which detail the cellular automata structure behavior. A very good visualization of how the cell state is divided is shown in Figure 6.6.

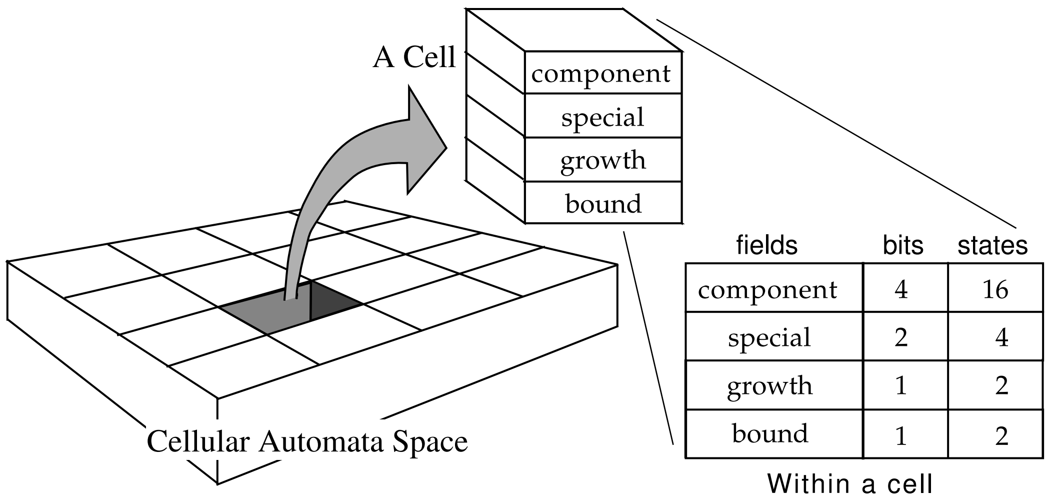


Figure 6.6: The data fields used in the emergent self-replication rule set. In this application, a cellular automata state variable in each cell is horizontally sliced into four different bit groups called *fields*. Each field represents a specific piece of information in the cellular automata model. Different bit-depths are assigned to different fields as indicated. The total number of bits used is eight, which is the size of the state variable in each cell.

6.5 Signal flow and path extrusion

A basic feature of a self-replicating cellular automata structure is to allow a signal sequence to be transmitted to another area of the cellular automata space. The signal sequence should be preserved during transmission to keep its original meaning. The general format of the signal sequence which will be transmitted in our model is of this form L>>>>, where an arbitrary number of signals > are followed by a signal L (the signals flow toward the right here; so they are read from right to left).

The building block of a cellular automata structure is the component O which allows the signal sequence to flow "through it". A typical signal sequence flow is shown in Figure 6.7.

The Trend cellular automata programming language described in the previous chapter is used to write the rules in this and the following chapter. The neighbor position prefixes used are no, ne, ea, se, so, sw, we and nw, which stand for **no**rth, **n**orth**e**ast, **ea**st, **s**outh**e**ast, **so**uth, **s**outh**w**est, **we**st and **n**orth**w**est, respectively, of the Moore neighborhood template. North is up, east is to the right, etc.

With this convention, the rules that implement the signal sequence flow are these:

```
if (component=='O')
     rot if (we:component=='>')
          component='>';
rot if (component=='>')
     if (we:component=='L')
          component='L';
     else if (we:component=='>')
          component='>';
```

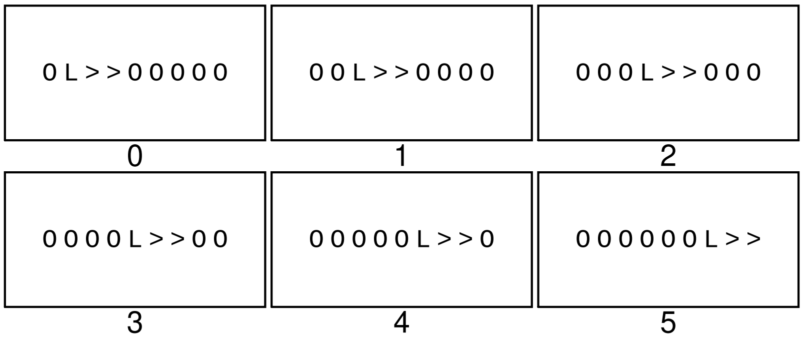


Figure 6.7: The signal sequence flow in a cellular automata structure. Each box here is a snapshot of the same region in the cellular automata space during different iteration steps. Numbers under each box are denoting the relative iteration steps. Signal $>$ is followed by either a signal $>$ or by the signal L. Signal L itself always changes to Block O. Block O itself will be changed to $>$ if pointed at by $>$.

```
if (component=='L')
        component='O';
```

A little explanation is needed for this first rule example. The first outer *if* statement says that if there is a signal $>$ as the west neighbor of a component O which points to it (as is the case in Figure 6.7), the component O will change to signal $>$ in the next epoch. The rotated if prefix *rot* on the inner *if* statement helps expand this condition to the other three possibilities, i.e., a $\vee$ in the north, a $<$ in the east or a $\wedge$ in the south.

The second outer *if* statement says that if the west neighbor of a $>$ signal is a L signal, the $>$ signal will change to L, but if the west neighbor there is instead a signal $>$, it will stay as $>$. Again, the rotated prefix *rot* expands the condition to the other three cases: a signal $\vee$ with a L or a $\vee$ on the north neighbor, a signal $<$ with a L or a $<$ on the east neighbor, and a signal $\wedge$ with a L or a $\wedge$ in the south neighbor.

The last *if* statement simply says L always changes to O.

The use of the rotated if statement (Section 5.10) is very important in writing cellular automata rules. It simplifies the effort by requiring only one description of a condition and automatically converts the condition to the other three orientations. Without the rotated if statement we would need to provide a much more awkward nested *if* statement to encompass all four orientations. Otherwise our signal sequence would only flow toward the right but not toward the top, left or bottom directions.

In addition to being able to pass a signal sequence around in its own body, a cellular automata structure should also be able to extrude toward the quiescent space so that it can grow in size, as shown in Figure 6.8. The rules to implement the extrusion are the following.

```
if (component=='B')
    rot if (we:component=='>')
        component='>';
```

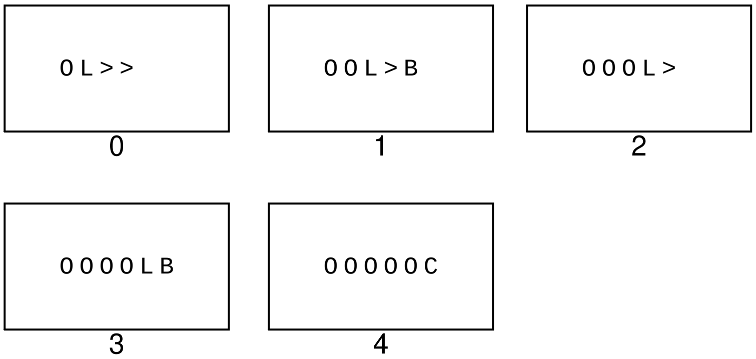


Figure 6.8: Extrusion of a cellular automata structure. The signal > will extrude a path toward a quiescent space by changing a quiescent neighbor cell into the Birth B. The signal L will change the tip of an extrusion path to the Corner C, which will cause the direction of extrusion to be rotated 90 degree counterclockwise. See Section 6.6 for rules about this.

```
if (component=='.')
     rot if (we:bound && we:component=='>' &&
              (nw:component=='.' || nw:component=='>,1' ||
               nw:component=='L'))
          component='B';
```

The first rule says that Birth B can pass along signal > too, just like the Block O. The second rule says that a quiescent state will be changed to Birth B when pointed at by a > signal. Since signal > is meaningful and can direct extrusion into quiescent neighbors only when it is part of a cellular automata structure, the rule needs to test the `bound` bit to see if it is not a monomer. The reason for the three test conditions in the second rule will be explained in the following section, after the turning and branching rules are introduced.

6.6 Signal turning and branching

A signal sequence can not only be transmitted straight ahead, it can also be transmitted around a corner, thus changing the direction of an extrusion, as shown in Figure 6.9. The additional rules to make the turning of a signal sequence transmission are these:

```
if (component=='O')
     rot if (so:component=='>')
          component='>,3';
rot if (component=='>')
     if (we:component=='>,1')
          component='>';
```

The first rule is invoked when going from epoch 2 to epoch 3 in Figure 6.9 and the second rule is invoked when going from epoch 3 to epoch 4 of the same figure. Note that the second rule is rotated before use in this case.

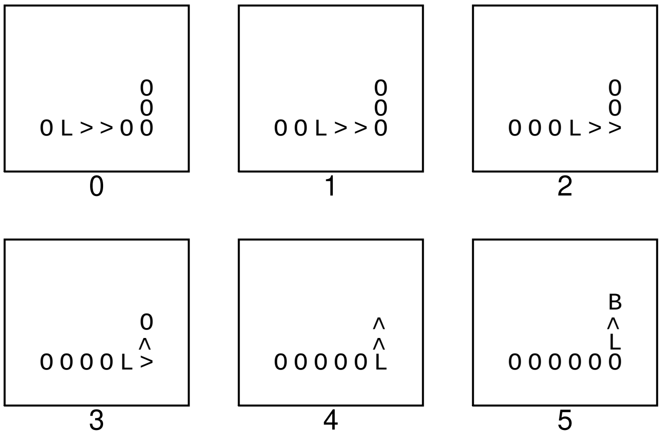


Figure 6.9: Signal sequence turning at the corner. A signal sequence can change its direction of transmission by going through a corner.

A corner itself is formed in two stages. First the signal L at the tip of an extrusion path changes the tip to a Corner C, as shown in Figure 6.8, after that, Corner C will rotate an incoming signal > counterclockwise 90 degrees, as shown in Figure 6.10, thus forming a corner. The additional rules to make this happen are these:

```
rot if (component=='>')
    if (no:component=='>,1')
        component='>';
if (component=='B')
    rot if (we:component=='L')
        component='C';
if (component=='C')
    rot if (we:component=='>')
        component='>,3';
```

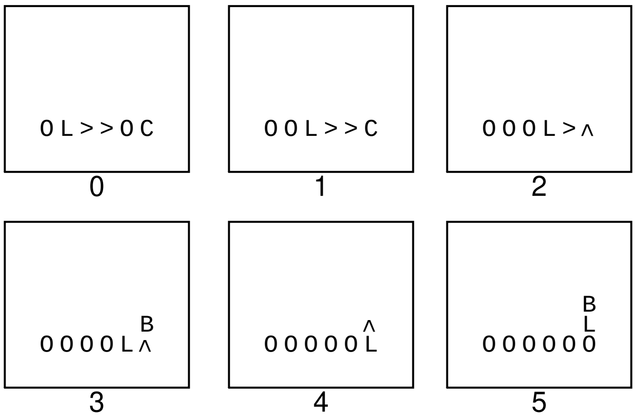


Figure 6.10: The formation of a corner. Corner C will rotate an incoming signal > 90 degrees counterclockwise, thus forming a corner.

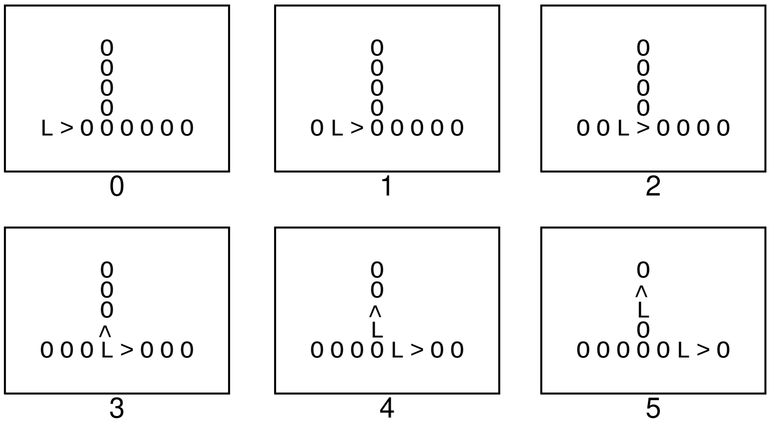


Figure 6.11: The branching of a signal sequence. When going through a dividing pathway a signal sequence will be duplicated.

The first rule allows subsequent turning of all $>$ signals, as shown in epoch 2 to 3 of Figure 6.10. The second rule changes the tip of an extrusion path into the Corner C as stated, which is shown in epoch 3 to 4 of Figure 6.8. The third rule rotates the incoming signal $>$ into $\wedge$ as shown in epoch 1 to 2 of Figure 6.10.

It should now be clear why the three test conditions of the last rule in the previous section about quiescent state changing to Birth B is necessary. Look at epochs 3 and 4 of Figure 6.9. We certainly do not want the quiescent neighbor to the right of the corner cell to change to a Birth B in epoch 4, despite the fact that it is pointed at by a signal $>$ in epoch 3. This and other similar situations require explicit listing of conditions about when a Birth B can be formed. Specifically, only in three conditions a quiescent state pointed at by a signal $>$ will change to Birth B. These conditions all are based on checking the value of its Northwest neighbor. The first case is a quiescent Northwest neighbor as shown in the two extrusion cases of Figure 6.8 for straight extrusion. The second case is shown in epochs 2 and 3 of Figure 6.10 for extrusion through a corner. The third case is similar to case 2 but for an extremely small loop with the L$>$ sequence (so that L followed right after $>$).

A signal sequence will be duplicated when going through a branching pathway, as shown in Figure 6.11. No new rule needs to be written to support this behavior. How the branching pathway itself is formed will be explained in Section 6.8.

6.7 An endless signal sequence source — the loops

Given that signal sequences can be turned around in cellular automata space, we can utilize this behavior to make a circular pathway, or a *loop*, for signal sequences, so that a signal sequence can be preserved within the loop and repeat itself. The loop, together with the branching of the signal pathway, makes an endless signal sequence source possible. In fact, that is the fundamental idea of the self-replicating loops [Reggia *et al.*, 1993a]. A typical example of the loop is shown in Figure 6.12.

```
000
0 0
L>>0000000
```
0

```
000
0 ^
0L>>000000
```
1

```
00^
0 ^
00L>>00000
```
2

```
0<^
0 L
000L>>0000
```
3

```
<<L
0 0
0000L>>000
```
4

```
<L0
v 0
00000L>>00
```
5

```
L00
v 0
v00000L>>0
```
6

```
000
L 0
v>00000L>>
```
7

```
000
0 0
L>>00000L>B
```
8

Figure 6.12: The signal sequence loop. A signal sequence can be preserved within a loop. This signal sequence can then be repeatedly sent out through the branching pathway.

6.8 Branch formation

As shown in the previous section, a signal sequence can be repeatedly sent out by utilizing a loop together with a branching signal pathway. In this section I will demonstrate how this branching signal pathway can itself be formed by a cellular automata signal sequence.

The signal sequence to form a new branch, or *arm*, of the signal pathway consists of the two transient signals E and F following the normal signal L. The signal sequence EF itself is formed by the help of a special value '*' in the special field (see Figure reffg:efields). When a normal signal L goes through a cell which has its special value set to '*', the signal L will change to E in the next epoch. Visually this appears as if the signal L "pulls out" a trailing EF signal sequence after it. When this trailing EF signal sequence reaches the next corner of the loop, it will break out the loop and form a new branch of the signal pathway, as demonstrated in Figure 6.13.

The rules to implement the "pulling out" of the EF signal sequence (epochs 1, 2, 3 and 4 of Figure 6.13) are the following. Signal L has two choices: if it sits on top of a special value of '*' it will change to signal E no matter what; otherwise it will change to E if followed by E. The extra conditions in the rule for signal L make sure the EF sequence will not be copied through the corner, as shown in epochs 4 and 5 of Figure 6.13. Signal E will always change to signal F and clear the special value. Signal F will always return to O (but it reappears in the current position of E).

```
if (component=='L')
    if (special=='*')
        component='E';
    else rot if (so:component=='E' && sw:component!='F'
                && (no:component=='.' || we:component=='.'))
        component='E';
```

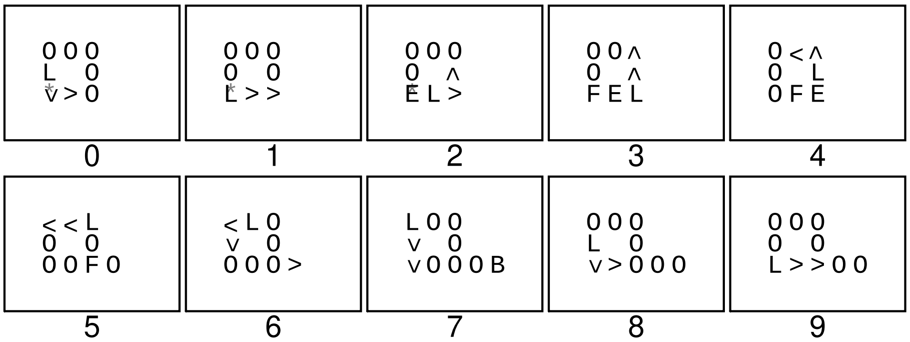


Figure 6.13: The branching out of a signal pathway. Note the special value '*' shown with a light gray color in the lower left corner of the loop at epoch 0. (Where the '*' comes from is explained later in the text.) When signal L reaches that cell it will change to the signal E instead of to the usual Block O, as shown in epoch 1 and 2. The signal E itself will "pull out" the signal F in the next epoch (epoch 3). It will also erase the special value '*' so that next time when signal L goes there again there will be no more EF signal sequences. This EF signal sequence will follow the signal L to the next corner of the loop, as shown in epoch 4. It will then form a branch, or arm, at that corner as shown in epochs 5, 6, 7 and 8. Signal L will never carry the EF sequence through the corner.

```
if (component=='E') {
        component='F';
        special='.';
}
if (component=='F')
        component='O';
```

The rules to implement the breaking out of a loop corner to form a new branch (epochs 5, 6, 7, and 8 on Figure 6.13) are the following.

```
if (component=='.')
    rot if (so:special==0 && so:component=='E' &&
          se:component=='.' && no:component=='.')
        component='O';
if (component=='O')
    rot if (so:component=='F' && se:component=='.' &&
            no:component=='.' && ea:component=='.' &&
            we:component=='.')
        component='>,3';
rot if (component=='>')
        component='O';
if (component=='B')
        component='O';
```

Signal E will turn the quiescent neighbor cell facing its moving path into an O. Several conditions are added to the rule to make sure that only the correct quiescent neighbor will be turned into an O, but not the other quiescent neighbors around E.

The O then sees the incoming F signal and will change to signal >, which will in turn extend the arm by one more cell and complete the branching. Again, since there are many O's around the signal F (three in epoch 5 alone), we need to add more conditions to the rule to make sure that only the right one (the O in the branching arm) will be turned into the signal >.

In this case signal > and Birth B will just fall back to Block O.

The only thing left unexplained is when and why the special field will be set to the value '*'. Basically, when a cellular automata structure is completing its replication process and has its arm close on itself, it will generate a D component, or Detach D. The Detach D is a component for separation between the parent cellular automata structure and the child structure. When seeing the Detach D in the neighbor, a cell's special field will be set to the value '*' by the following rule.

```
rot if (component!=0 && special=='.' &&
      (ea:component=='D' && no:component && we:component ||
       we:component=='D' && no:component && ea:component))
        special='*';
```

Again, conditions are added to make sure only two special fields, one in the parent and one in the child, will be set to '*'. This will cause the formation of a new arm in the other corner a few epochs later so that both the parent and child cellular automata structures can continue their own replication process. The formation and function of Detach D will be explained in the following section.

6.9 Loop closing and separation

The basic procedure by which a cellular automata structure achieves self-replication is to extrude an arm, turn the arm after a certain length of extrusion, continue extruding and turning until the arm closes on itself. At that stage, a new loop which is capable of preserving signal sequences has been formed (see Section 6.7). Now the original cellular automata structure has to separate itself from the new loop. This is achieved by setting a Detach D in the connecting cell, as shown in Figure 6.14. Note the setting of '*' values in the special field for both loops at epoch 35, which will cause new arms to be formed in the other corners later and a replication cycle to complete.

The rules to set Detach D and the rule to remove it are the following:

```
rot if (component=='>')
    if ((nw:component=='>,3' || nw:component=='O') &&
          (ne:component=='B' || ne:component=='>,1')
          && no:component=='.' && ea:component &&
          we:component)
        component='D';
```

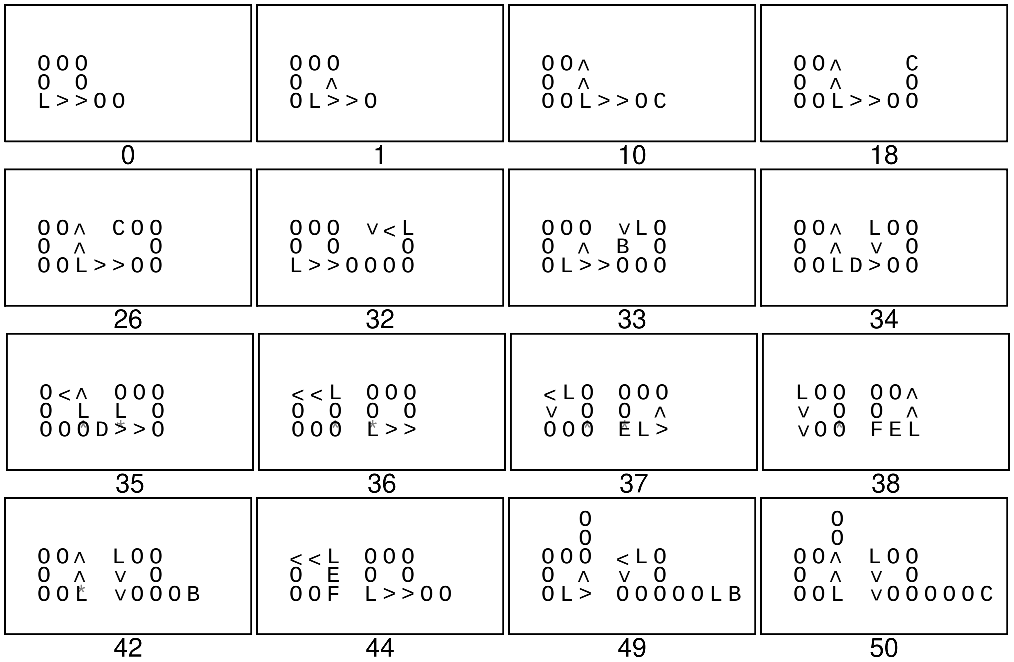


Figure 6.14: The replication and separation of loops. The original cellular automata structure (epoch 0) goes through a series of extruding and turning processes (epochs 1, 10, 18, 26, and 32) until finally its arm is closing on itself (epoch 33). When this happens Detach D will be formed in the connecting cell between these two loops (epoch 34). The Detach D will in turn trigger the settings of '*' values in the special field for both loops, shown in a light gray color in epoch 35. After seeing the setting of '*' values in its periphery, Detach D itself will return to quiescent state to complete the separation process (epoch 36). In epoch 37 and 38 the right (child) loop is going through the branching process, which has been demonstrated in Figure 6.13, that will cause a new arm to be formed to its lower right corner later in epoch 44. In the same epoch we also see the complete replication of a child loop. The left (parent) loop will go through the same process too and a new arm will be formed to its upper right corner later in epoch 49. It will return to its original state (but rotated 90 degrees) in epoch 50.

```
if (component=='O')
     rot if ((nw:component=='>,3' || nw:component=='O') &&
            (ne:component=='B' || ne:component=='>,1')
            && no:component=='.' && ea:component &&
            we:component)
          component='D';
if (component=='D')
     rot if (ea:special)
          component='.';
```

Normally the Detach D will come from the signal > rule, but in the extended replicating case, where a smaller loop is generating a bigger loop, the Detach D will come from the Block O rule. The extended replicating case will be explained in the following paragraphs. Note that lots of conditions are added to the two rules to make sure that only the correct 'O' or '>' will change to

D. Detach D will disappear right after seeing the special fields in its neighbors are set.

Usually a cellular automata structure will replicate another structure with the same size and signal sequence as itself as shown in Figure 6.14, but this can be changed if the cellular automata structure contains a signal sequence which will generate a bigger structure than itself. This is called an *extended* replication case. A cellular automata structure's signal sequence can be modified to generate different structures than itself by an active growth field sitting on one of its cells during the arm branching process, which will be explained in Section 6.12. The growth field is set when a loop dies, which will also be explained in Section 6.12.

When an extended replication is under way, there is a timing problem, as shown in Figure 6.15. The problem is that due to the different sizes of parent and child structures, there will be a partial signal sequence copied into the new loop, together with a complete, correct signal sequence. This partial signal sequence must be erased in order to guarantee a healthy new loop. This is achieved by setting the special field to the value '-' in the closing corner cell of the new loop. The Block O there will detect an extended replication in progress and will set its special field accordingly. Once the special field is set to '-', the Block O will stop copying signal sequences, thus effectively erasing any signal sequence going through it, as shown in Figure 6.15 at steps 39, 40 and 41. This erasing process will end only when the Block O sees an incoming signal L, which is the tail of the partially copied signal sequence which must be erased. Once it sees this incoming signal L, it will reset the special value from '-' to '*', to start the process of generating a new arm, which is what a new loop would have done already if it was not of different size to its parent.

The rule to set the special field to '-' is this:

```
if (component=='O')
    rot if (no:component=='B' && nw:component=='.' &&
          (we:component=='O' || ea:component=='L'))
        special='-';
```

The rule to inhibit the normal copying of signal sequence for Block O and the changing of '-' to '*' once it sees incoming signal L is this:

```
if (component=='O')
    if (special!='-')
        doing normal copying...
    else rot if (we:component=='L')
        special='*';
```

Self-replication rules which support extended replications are new in this research and are very important since only by the support of them can the emergent self-replicating structures diversify their sizes and shapes in the cellular automata space. They are also better than the old self-replication rules since they abstract the self-replication phenomena out of any particular shape and size of the self-replicating structures; they are truly *general purpose* self-replicating rules.

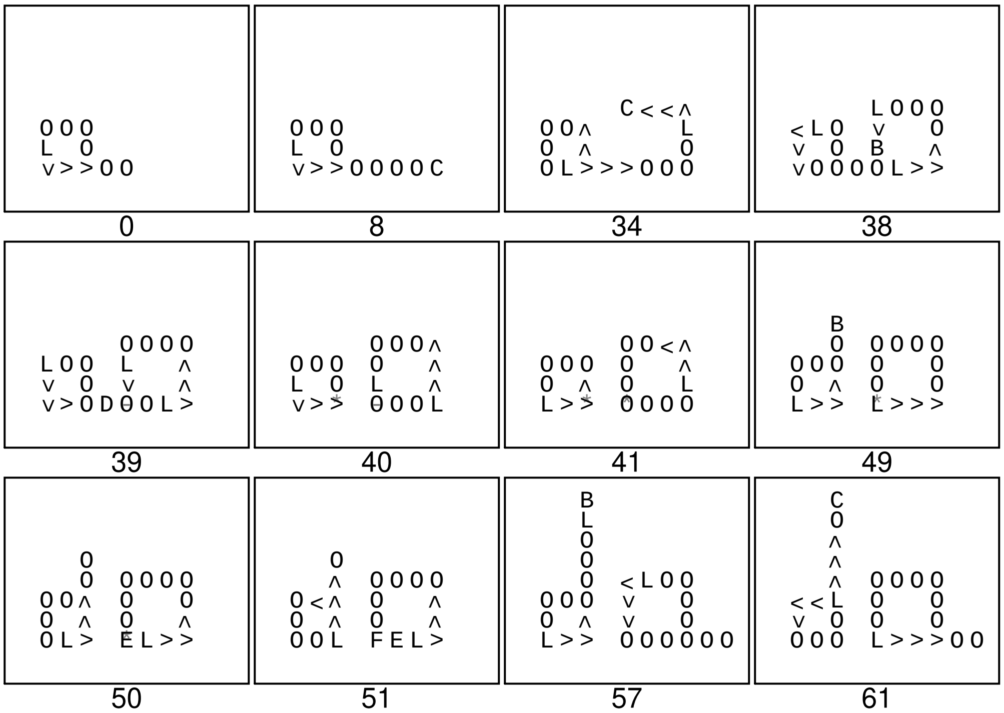


Figure 6.15: An extended replication of loops. Note that in epoch 0 the parent loop has one more > signal than it normally has (compared to Figure 6.14), which will cause a bigger loop to be generated. How this extra > signal gets added will be discussed in Section 6.12. In epoch 8 we see this signal sequence generate a longer arm. In epoch 34 the arm has completed about 3/4 of its closing process. In epoch 38 the arm is closing on itself, which causes two things to happen: the Detach D will be set in the connecting cell as usual, and a special '−' value will be set to the special field, shown in a light gray color in epoch 39. This setting of the special field will immediately stop the Block O from copying the signal sequence until it sees an incoming signal L in epoch 40. Then it will set the special field to the value '*' as shown in the same light gray color in epoch 41 (L is not copied). From then on the new arm branching process is under way as shown in epoch 49, 50 and 51. Note that the parent (left) loop completes its arm branching process and starts a new replication cycle in epoch 50. The extended signal in the child (right) loop makes it take longer time to finish the replication. In epoch 57 the new arm of the child loop is formed and finally in epoch 61 the bigger child loop is completed.

6.10 Failure detection and clean up

In the original self-replicating loops [Reggia *et al.*, 1993a], all cellular automata space behaviors are predictable. Therefore, the cellular automata rules do not need to account for random behaviors. Put another way, while writing the rules the cellular automata rule set designer has complete control over the behaviors occurring in the cellular automata space, including the initial state.

In contrast, in this endeavor, the very first assumption is that a rule set designer does not have any *a priori* knowledge about the interactions between cellular automata structures in the cellular automata space, or even what the cellular automata space will begin in epoch zero. Although the rules considered here are correct and can reliably direct a structure to do replication or extended replication when working **alone**, it does not guarantee that a structure will not run into another structure, or two structures will not try to replicate into the same region of the cellular automata space. These factors are all randomly determined, not by the designer.

Because of this, we need to assume that not all designated regular procedures like extrusion, branching, closing, etc. of the cellular automata structure will always be followed without interruption or disturbance from the other structures. To account for this unpredictability, and to come to the rescue when something goes wrong, we need to build into the cellular automata rule set rules which will detect any failed regular procedure and clean up the cellular automata space after the failure.

It is this fail-safe feature of the new rule set which distinguishes it significantly from the original self-replicating loop rule set.

A failed situation happens when something prevents the regular replicating procedures from continuing, such as an obstacle in the extrusion path, or two loops colliding into each other. When such a situation occurs it is marked by the fail-safe rules using the value '#' in the `special` field. When a cellular automata structure has any of its cells go into this "fail" mode, the structure will be plagued by this "fail" mark in a very short time, as shown in Figure 6.16, and will *dissolve* completely. Note that Detach D has the ability to block the fail mark from passing through it, thus protecting a failed child from its parent or vice versa.

When a cell of a cellular automata structure goes into the *dissolve* mode, it will lose its `bound` bit and become a *monomer*. Once becoming a monomer the cell will be governed by the monomer rules. The `bound` bit, monomers and monomer rules will be explained in Section 6.13 below.

The following rules govern the spreading of fail marks '#' and the dissolve of `bound` bits in cellular automata structures. Note that Detach D will **not** copy the fail mark. The testing condition `component` is to check if it is non-zero, or non-quiescent, which is equal to test *if* `(component!='.')`. This test is necessary since the fail mark will only be set for non-quiescent components.

```
if (special=='#')
        bound=0;
else rot if (component && component!='D' &&
          (no:special=='#' || ne:special=='#'))
        special='#';
```

Each of the following rules checks for a specific failure situation and sets the fail mark once such a fail situation is found.

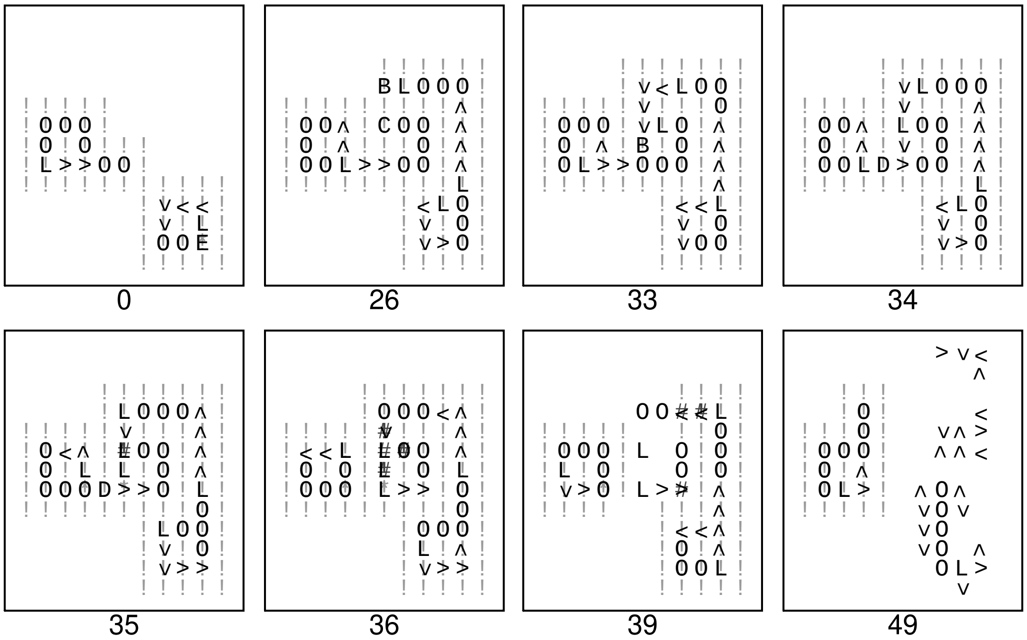


Figure 6.16: The dissolve of cellular automata structures. The bound bit, if set, is shown as a light gray exclamation mark '!' in the background. It denotes the part of the cellular automata space which belongs to a cellular automata structure. In epoch 0 we have two cellular automata structures both starting their replicating cycles. In epoch 26 both cellular automata structures have attained certain sizes. In epoch 33 the smaller inner new loop has closed on itself, but unfortunately the new outer loop is just about to touch the inner one. In epoch 34 the separation Detach D is formed at the connecting cell. At the same epoch the upper left L cell of the inner loop sees a fail situation and sets its fail mark '#' in its special field in the next epoch (shown in light gray in epoch 35). In epoch 36 the Detach D has disappeared and therefore the left most original loop is secured, while the fail mark is spreading in the inner and outer loops. The spreading of the fail mark '#' in the cellular automata structures is very fast (epoch 39), such that in epoch 49 all but the leftmost cellular automata structure has been dissolved completely into monomers. The original left loop is not influenced and will continue its replication in the new direction.

The first rule checks for abnormal cell neighbor density. A normal cellular automata structure cell should not have too few (less than 1) or too many (more than 5) neighbors, unless a collision has occurred which pushes cells together. The function `AbnormalNeighbor()` checks for that.

```
if (component && AbnormalNeighbor())
    special='#';
```

The definition of the function `AbnormalNeighbor()` is given below:

```
int AbnormalNeighbor() {
    count=0; // clear the counter
    // count the no.  of non-quiescent neighbors.
    over each other y:
        if (y:component) count++;
```

```
        // if zero or more than 5, it's abnormal!
        if (count && count<=5)
            return 0;
        else return 1;
    }
```

All the rest of the failure checking rules bear a similar resemblance to the following example, i.e., they are all elaborated forms of previous rules which were introduced already for various regular functions. More code is added to check for abnormal situations. For example, the following rule for passing signal > in Block O was introduced in Section 6.5:

```
if (component=='O')
    rot if (we:component='>')
        component='>';
```

To check for failures more conditions are added to see if there is more than one > signal which wants to move into the same O. If that is the case it must be a collision and the fail mark should be set. Otherwise everything goes normally and the signal > is copied.

```
if (component=='O')
    rot if (we:component=='>')
        if (no:component!='>,1' && no:component!='>,2' &&
            ea:component!='>,2' && ea:component!='>,3' &&
            so:component!='>,3' && so:component!='>')
             component='>';
        else
             special='#';
```

Based on the same reasoning the rule to turn a signal > at the corner (Section 6.6) now becomes:

```
if (component=='O')
    rot if (so:component=='>')
        if (we:component!='>,1' && we:component!='>' &&
            no:component!='>,2' && no:component!='>,1' &&
            ea:component!='>,3' && ea:component!='>,2')
             component='>,3';
        else
             special='#';
```

Each of the following rules checks for an unwanted situation for each different component type and sets the fail mark accordingly.

Birth B should not be within a line of components:

```
if (component=='B')
    rot if (no:component && so:component)
        special='#';
```

Signal > should not catch up with signal L:

```
if (component=='L')
    rot if (we:component=='>')
        special='#';
```

EF signal sequence should not reach Corner C:

```
if (component=='C')
    rot if (no:component=='E' || no:component=='F')
        special='#';
```

The E before the F should have made a new Block O, unless something prevents it from doing so, in that case the loop has failed to make a new arm, and then it is a failure:

```
if (component=='F')
    rot if (no:component=='O' &&
          (ea:component=='O' || ea:component=='L') &&
          so:component=='.' && we:component=='.')
        special='#';
    else
        component='O';
```

6.11 The minimum loop

Our examples in previous sections show cellular automata structures of various sizes. This leads one to ask: what is the smallest possible cellular automata structure capable of self-replication? In order to facilitate emergent self-replicating behaviors we would like the self-replicating structures to be as small as possible so that the chance of getting them from monomers can be high. A previous research endeavor in minimizing self-replicating loops has led to a self-replicating structure with only 5 cells [Reggia *et al.*, 1993a]. Our rules so far can support the self-replication of loops 3 by 3 cells and larger. That seems to be the smallest loop we can get since the signal sequence passing rules we have so far require that signal sequence flows not be adjacent to each other. But the actual limiting factor is the number of cells in a loop. For the arm extrusion sequence (EF) to work we need on some occasions two more cells in addition to the number of cells to hold the normal replication sequence. We also need to have at least one more cell if we want to allow extended replication cases so that loops can grow in size. For a 3 by 3 cells structure the total number of required cells is six. For a 2 by 2 cells structure that number becomes five, a seemingly impossible case since it only has four cells.

After some careful studies it is found that we can reduce the size of the loop even further by adding some more rules to the rule set we have gotten so far. This makes possible a self-replicating structure as small as only 2 by 2 cells. The added rules watch for special cases for 2 by 2 loops since signals in a 2 by 2 loop are so close to each other that requires special signal passing rules in addition to those we already have. This smallest self-replicating loop can be called the *Adam Loop*. A whole new world of self-replicating cellular automata structures can be generated from this loop. In Figure 6.17 we show an Adam Loop and its replication process.

This added rule turns the signal L after the signal > around the corner as seen in epoch 18 and 19 of Figure 6.17 for the child loop.

```
rot if (component=='>')
    if (no:component=='L')
        component='L';
```

The following rule copies signal L to B's position as seen in epoch 17 and 18 of Figure 6.17.

```
if (component=='B')
    rot if (we:component=='L' && no:component=='O')
        component='L';
```

The following rule rotates the signal > in a tight corner as shown in epoch 15 and 16 of Figure 6.17.

```
if (component=='C')
    rot if (we:component=='>,1')
        component='>,3';
```

Finally, this rule mobilizes the signal > behind the F signal as shown in epoch 2 and 3 of Figure 6.17. It has the added failure checking code.

```
if (component=='F')
    rot if (no:component=='>,2' && ea:component)
        if (ea:component=='E' || so:component=='O')
            component='>,1';
        else
            special='#';
```

6.12 Growth stimuli

As shown in the previous section, the smallest loop has only 2 by 2 cells. The previous smallest self-replicating loop has 5 cells [Reggia *et al.*, 1993a]. It is made possible in this work since the number of states used in this model, 256, is far more than the 8 states used in the previous research. To get bigger cellular automata structures from this 2 by 2 loop, mechanisms are needed to let the smaller loop generate bigger loops. This is achieved by the growth stimulus bit in the growth field together with the help of several more rules.

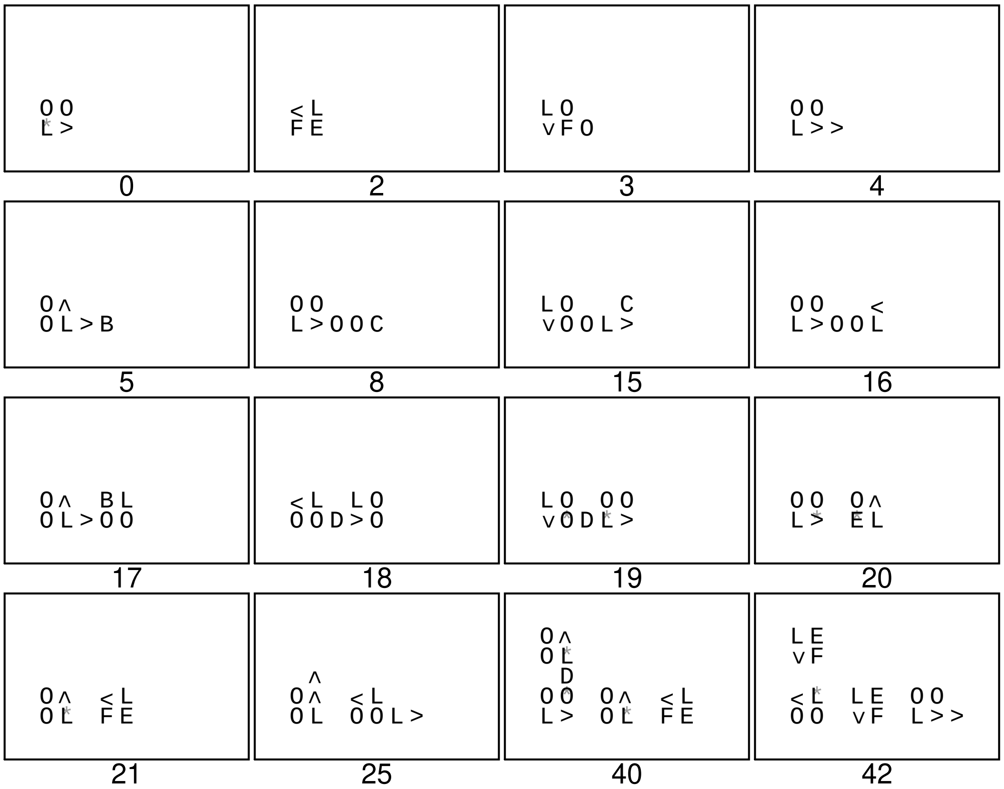


Figure 6.17: The smallest self-replicating cellular automata structure in this model. This structure starts as only a 2 by 2 loop with its lower left cell's special field set to '*' for extruding an arm. In epoch 2 the arm branching sequence EF has been generated and is approaching the next corner. In epoch 3 the branching starts. In epoch 4 and 5 the branching sequence and the original self-replicating signal sequence are precisely combined to continue extruding the new arm. In epoch 8 one side of a new loop has been established. Epochs 15 and 16 show the intermediate stages of its replicating process. In epoch 17 the new arm is closing on itself and in Epoch 18 the separation Detach D is formed as usual. In epoch 19 the two branching special values '*' are set for both loops. Note that although the child (right) loop has not detached from the parent (left) loop, it has completely replicated the parent loop in epoch 19. In epoch 20 the Detach D disappears, completely separates the parent and child loop. In epoch 21 the parent loop also completes its replication cycle and is rotated 90 degrees counterclockwise. In epoch 25 both loops extrude a new arm and start their own replication cycle. In epochs 40 and in epoch 42 both loops are again completing their replication cycle. We get four loops, all are still actively replicating in epoch 42.

First we need to devise a way to set the growth bit. Remember that we do not have any *a priori* knowledge about what will happen in the cellular automata space, and therefore we cannot preset the growth bit at any fixed position. Even to start with randomly assigned growth bits may seem a bit artificial since the positions of the growth bits will not be changed during the whole simulation.

The function of generating the growth bit is thus included in the cellular automata rule set. Whenever there is a signal L dying, it will leave a growth bit at its location. This way, the generation of the growth bit itself becomes part of the behavior of the cellular automata space. The rule to do this is as follows:

```
rot if (component && component!='D' &&
        (no:special=='#' || ne:special=='#')) {
      special='#';
      if (component=='L') growth='+';
}
```

The growth bit is utilized during the arm branching phase of a cellular automata structure (see Section 6.8). It is a two step strategy. First, if a signal > sees a growth bit in its place and it is the last > before the signal L, it will **not** copy the signal L behind it as it normally does. Instead, it stays at its current value > for one more epoch, thus effectively increasing the size of the signal sequence by one. The signal L will disappear temporarily since it is not copied, but will reappear when the signal > sees a trailing signal F and the growth bit in its position. The growth bit will be reset after signal L is regained, so the same growth bit will not cause another growth stimulus.

To sum up, if a loop is stimulated by a growth bit, its replicating code is expanded to make bigger loops than itself. Without any growth bit stimulus, loops always replicate themselves. When a loop dies, it leaves a growth bit behind, and when a loop expands, it uses a growthbit. This is an interesting ecological balance factor in the cellular automata universe.

The growth behavior is shown in detail in Figure 6.18 and the rules are shown below. Note that there is one special case considered in the rules. For a 2 by 2 loop to generate a 3 by 3 loop, it needs the signal sequence L>>. When it is expanding its arm, it also needs the signal sequence EF. Together that is five signals, more than the four cells in a 2 by 2 loop. To solve this limiting problem the growth bit is used to temporarily hide the signal L when the EF signal sequence is generating a new arm for the 2 by 2 cells. The signal L will be regained after the EF signal sequence has done its job of creating a new branch, as stated above. Therefore, a 2 by 2 loop can generate only 3 by 3 loops; nothing bigger than 3 by 3 can come out from it directly. Normally for the other loops which are bigger than 2 by 2 the rules allow them to generate loops more than one cell bigger than themselves.

```
rot if (component=='>')
    if (we:component=='L')
        if (nw:component=='E' && (growth || no:component=='>,3'))
            growth='+';
        else
            component='L';
    else if (growth && nw:component=='F') {
        component='L';
        growth=0;
    }
```

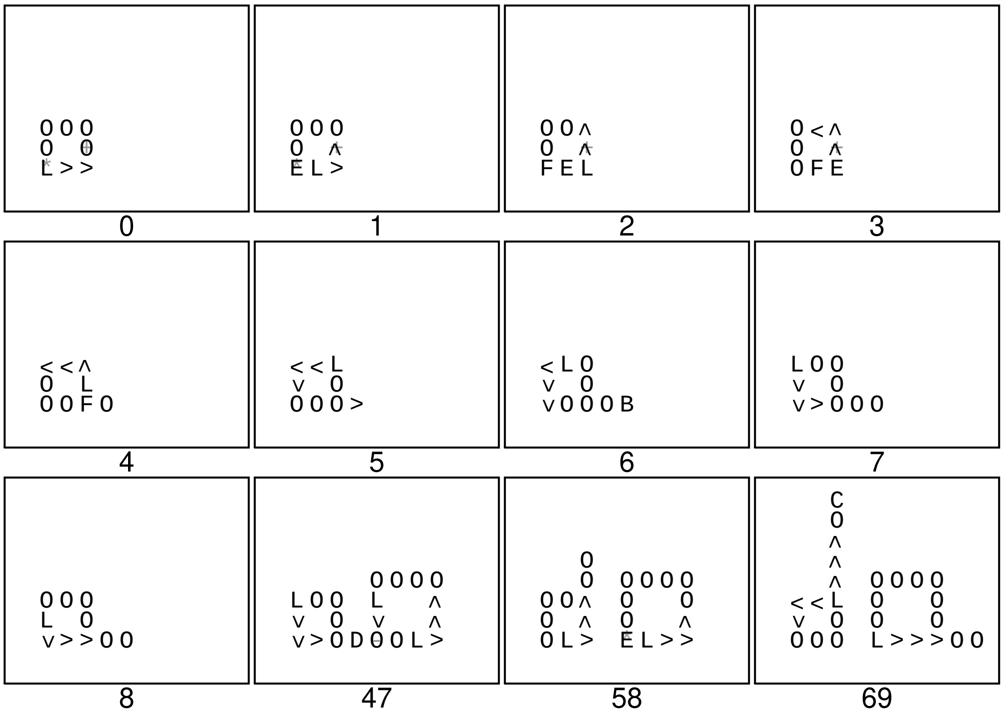


Figure 6.18: The growing of a loop. In epoch 0 the branch special flag in the lower left cell and the growth bit in the middle right cell are both set. In epoch 2 the normal arm branching EF signal sequence is generated. In epoch 3 the signal sequence becomes >>> and the signal L disappears temporarily due to the growth bit. In epoch 4 the signal L reappears and the growth bit is erased. Epochs 5, 6 and 7 show the continuation of the arm branching process. In epoch 8 the parent loop returns to its normal position and is about to start the replication cycle. Note that now it has one more > signal than it normally has. In epoch 47 a whole new loop bigger than the original one is generated. In epoch 58 the two loops have separated and the original one is just about to start another replication cycle. Finally in epoch 69 the new and bigger loop has finished its arm branching effort and is starting its own replication cycle too.

6.13 Monomer behavior and the game of life

All of our discussion in previous sections is focused on how self-replicating structures function and how the smallest loop can bootstrap to generate larger and larger loops. But we still need to get the smallest loop first before anything can happen. The cellular automata space starts at epoch zero with randomly distributed components, which are all monomers. Even though the smallest loop has only 2 by 2 cells, the chance to get it at epoch zero is still minimal. We need to have some means to stir up monomers in the initial cellular automata space so that they will translate and change state. Through time, the chance to get the smallest loop will be much more higher.

The rules used to implement these monomer actions are motivated by the well-known "*Game of Life*" rules by John Conway [Gardner, 1970]. They state the following conditions under which a cellular automata monomer will be modified:

- If a quiescent cell has exactly three active neighbors, it will itself become active in the next epoch. Its active value will be determined randomly by the neighbors.

- If an active cell has exactly two or three active neighbors, it will stay active; otherwise, an active cell will return to quiescent state in the next epoch.

A few modifications to the old Game of Life rules are required in order to determine the active state value of a birth cell. The old rules deal with binary cellular automata states so there is only the choice of 1 or 0, but the cellular automata cells here have more than one active state and the probability of generating any active state value to is desired to be about equal. This is achieved by using a `value` accumulator variable during the scanning phase of the rules, and using its value to determine the new state value if a new cell will be generated according to Conway's rules:

```
if (bound==0) {
    count=0; value=0;
    over each other y: {
        if (y:component) {
            count++;
            value = value + y:component;
        }
    if (count<2 || count>3)
        component='.';
    if (component=='.' && count==3)
        component = (value+1)%6+1;
}
```

Note that as stated before, we use the `bound` bit to determine if a cellular automata cell is a monomer, or if it belongs to a multi-cell cellular automata structure. A cellular automata cell will be governed by the monomer rules if its `bound` bit is **not** set. The outer *if* statement tests if the cell is a monomer (so its `bound` bit is 0). If it is a monomer, the inner *over* command will scan all its neighbors and count the number of active neighbors in the `count` variable and also add up active neighbor state values in the `value` variable. The `value` variable will be used later if it is determined that a new active cell will be born.

The first *if* statement after the scanning statement tests to see if the current cell will return to (or stay at) the quiescent state due to too few or too many active neighbors. It is the classic Game of Life rule for death.

The second *if* statement tests if the current cell is quiescent but has exactly three active neighbors. If that is the case a birth will happen. This is also the classic Game of Life rule for birth. The only difference here is that the `value` is used in the calculation to determine what active state value will be set to this new birth cell.

An example of this monomer rule set is given in Figure 6.19.

6.14 Emergence of first replicates

Finally, I need to link the monomer rules to the self-replicating rules from previous sections, such that an emergent self-replicating structure can spontaneously appear in the cellular automata space. This is done by adding a last set of cellular automata rules.

The original *over* statement for the monomer rule (see previous section) is modified to include a new statement to check for any bound cell around a monomer cell. If any bound cell is found

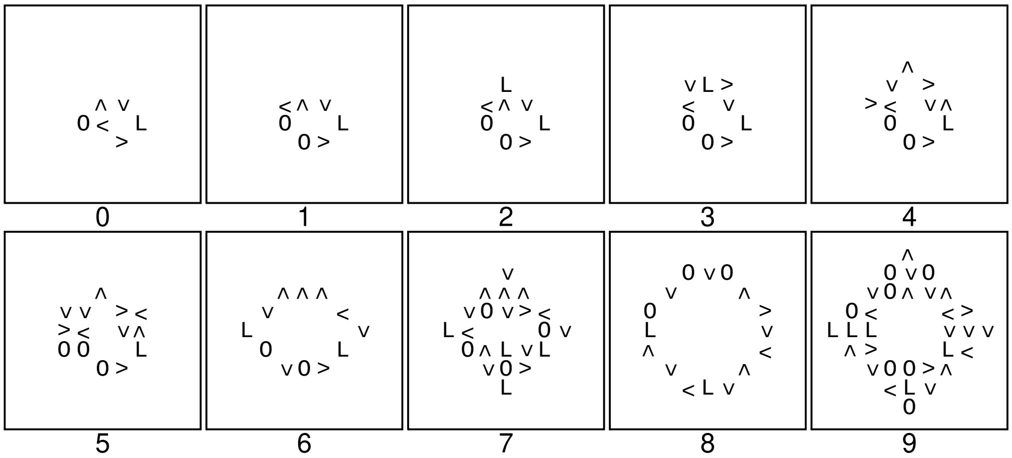


Figure 6.19: The monomer stirring examples. In epoch 0 some random active states are put into the cellular automata space. In epoch 1 the middle ’<’ component dies because it has more than 3 active neighbors. In the same epoch two new components, one ’O’ and one ’<’, are born since there is exactly three active neighbors in their positions. Note that the state value ’O’ and ’<’ are determined by state values of the active neighbors. In epoch 2 only one new ’L’ component is born. The stirring process continues in this fashion indefinitely.

around a monomer cell, this statement will set the `count` value to a magic number 99 to denote such a situation. The reason to do that is because all monomers around a multi-cell cellular automata structure should be removed to guarantee safer operations of the replication process of the multi-cell structure.

```
count=0; value=0;
over each other y: {
    if (y:component) {
        count++;
        value = value + y:component;
    }
    if (y:bound && y:component && y:special!='#') {
        count=99;
        break ;
    }
}
```

After the scanning process done by the *over* statement a set of test rules are added to the original monomer rule set. Each of these rules checks for one special condition and invokes its associated monomer actions. These conditions are exclusive to each other, so the *else if* construct is used among these rules.

The first rule checks to see if there is any bound cell around a monomer by looking at the `count` value accumulated in the scanning phase. As stated above, a monomer component will be destroyed if it is adjacent to a bound cell. In addition, its `bound` bit will be set on (represented by symbol '!' in the rule code) to prevent further monomer growth in its position. If we visualize this we will always see a clear band of '!' marks around a multi-cell loop, because of this rule.

```
if (count==99) {
    component='.';
    bound='!';
}
```

The second rule checks for a nonzero special field and will reset it to zero if found. It effectively delays the application of normal monomer rules by one epoch. This delay is necessary to protect a cell being dissolved from a multi-cell cellular automata structure from being destroyed by its own fellow neighbors belonging to the same structure which are still being dissolved.

```
else if (special)
    special=0;
```

The last three added rules watch for the formation of the smallest loop configuration (the 2 by 2 loop, see Section 6.11) in the cellular automata space during the course of monomer activities. Once such an "Adam Loop" configuration is found, all four members of it will set their own bound bit simultaneously and produce an active smallest loop in the next epoch. **This is how the first self-replicate is formed!** These three rules each work for one of the three different components in an Adam Loop, after the previous two rules.

```
else rot if (component=='>' && we:component=='L' &&
        nw:component=='O' && no:component=='O') {
    bound='!'; special='*';
}
else rot if (component=='L' && ea:component=='>' &&
        no:component=='O' && ne:component=='O')
    bound=1;
else rot if (component=='O' &&
    (so:component=='L' && se:component=='>' &&
         ea:component=='O' ||
     so:component=='>' && sw:component=='L' &&
         we:component=='O'))
    bound=1;
```

Finally, if none of the above special cases is found, the familiar Game of Life conditions are checked just like the monomer rules of previous section.

```
else if (count<2 || count>3)
    component='.';
else if (component=='.' && count==3)
    component = (value+1)%6+1;
```

An example of how the new monomer rule set works and how it leads to the first self-replicating structure is demonstrated in Figure 6.20. This completes the discussion of my emergent self-replicating cellular automata rule set.

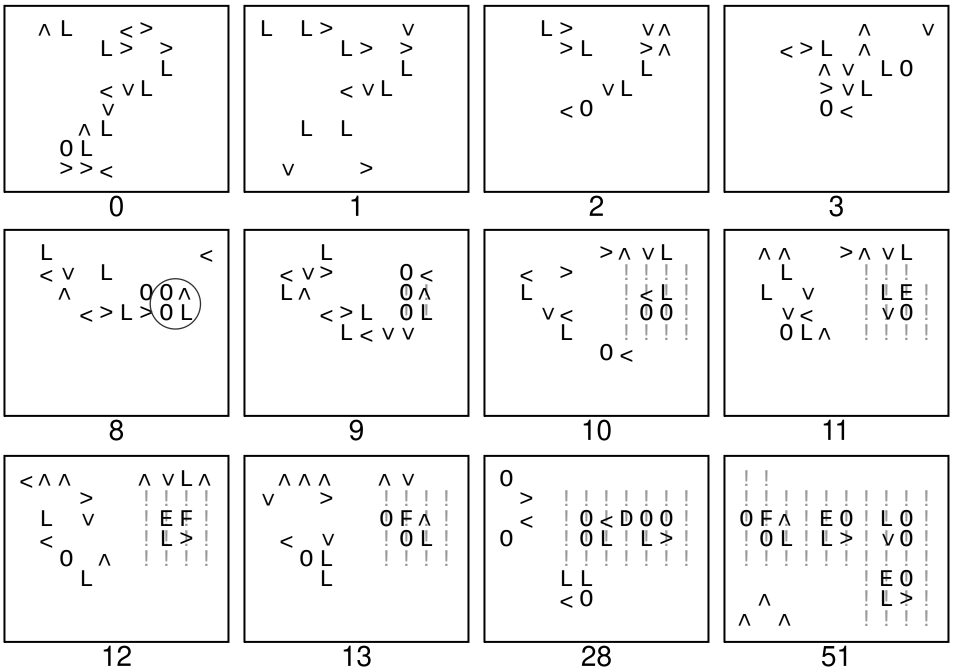


Figure 6.20: The emerging of the first self-replicating structure. Remember that self-replicating structures are marked by a non-zero bound bit, or an '!' mark in the space. Therefore, to look for emergent self-replicating structures we just need to look for marked '!' regions in the cellular automata space. Note that bound bits are shown only in this figure and in Figure 6.16 when necessary but not in the other figures of this chapter, to avoid interference with the other symbols since their bound bit status are clear from the context. In epoch 0 a randomly generated initial space is given. This space shows normal monomer activities until epoch 8, when the smallest loop configuration (the Adam Loop, as circled) randomly appears in the cellular automata space. In epoch 9 this configuration turns into a functioning self-replicating loop in the cellular automata space when its four cells set their bound bit simultaneously (marked by four '!' marks in the right region of the space). This loop immediately clears its peripheral cells and begins its arm branching process (epochs 10, 11, 12 and 13). In epoch 28 its first sibling is about to be separated from itself. In epoch 51 four loops are obtained and all are actively engaging in their replication processes.

6.15 Properties of the emergent rule set

In this section the properties of the emergent self-replicating rule are explored through a systematic battery of simulations. Most of the results are presented by graphs.

6.15.1 How the simulation is conducted

In order to study the properties of the emergent self-replicating rule set, a series of simulation runs were conducted. Cellular automata space sizes of 50 by 50, 100 by 100, 150 by 150 and 200 by 200 were chosen for simulations. For each space size, initial monomer densities of 10%, 20%, 30%, 40% and 50% were used. For each combination of space size and initial monomer density, a certain number of simulations each with different random initial configurations were run for 30000

		Space Size			
		50 by 50	100 by 100	150 by 150	200 by 200
Initial Monomer Density (%)	10	1 till 20158 1 till 19527 2 till 30000 1 till 95	5 till 30000	5 till 30000	5 till 30000
	20	3 till 30000 1 till 10418 1 till 29788	5 till 30000	5 till 30000	5 till 30000
	30	1 till 8424 1 till 8230 1 till 18621 1 till 17004 1 till 26459	5 till 30000	2 till 30000	5 till 30000 1 till 29874
	40	2 till 30000 1 till 7140 1 till 16887 1 till 28108	5 till 30000	1 till 30000	2 till 30000
	50	1 till 30000 1 till 15488 1 till 22946 1 till 21092 1 till 9194	5 till 30000	None	None

Table 6.1: Summary of simulations with different parameters. For each combination of space size and initial monomer density, a number of simulations were done, each with a different initial random configuration. Some simulations stopped before reaching epoch 30000 because their cellular automata spaces had ceased activities. If a simulation reached epoch 30000 it was stopped manually. Table entries are the actual number of epochs each simulation reached. Therefore, "1 till 20158" means 1 simulation is run until epoch 20158, and "5 till 30000" means 5 simulations are run and they all stop at epoch 30000.

epochs unless the cellular automata space ceased its activity before reaching epoch 30000.

The number of simulations for each category and the actual number of epochs each simulation reached are summarized in Table 6.1. Since it appears that the initial random pattern and the initial monomer density are not influencing factors of the simulation characteristic (see the following subsections), fewer simulations were conducted for some combinations of space sizes and initial monomer densities.

It can immediately be seen in Table 6.1 that it was harder to support longer term evolution of the emergent self-replicating loops in the smaller cellular automata space size. Many simulations for the 50 by 50 world size ceased their activity before reaching epoch 30000. Despite the shorter life span of simulations for this space size, they still generated self-replicating loops except for one simulation with a 10% monomer density, where the cellular automata space failed to produce a working minimum self-replicating loop before all monomers stopped their activities early at epoch

95. Since only one out of a total of 81 simulations failed to generate self-replicating loops, apparantly it is easy to generate self-replicating loops with the emergent self-replication rule set.

6.15.2 How the data were collected

Since it was not feasible to collect the simulation data manually, a carefully designed sophisticated data collecting module was built into the simulator to analyze the cellular automata space configuration on-the-fly while the simulation was running. The data collecting module is smart enough that it not only counts the number of cells of certain types, it also recognizes some higher level structures. Replicating loops were intelligently identified by the data collecting module individually on the space, and their sizes were recorded. Accumulated data for each simulation epoch contained the number of active cells, the number of active bound cells, the number of growth stimuli bits, and the size of individual loops in the cellular automata space. The size data were later batch processed by a condensing utility program to determine the average size of loops for each epoch. Therefore, the final data for analysis were the number of active cells, the number of bound cells, the number of growth bits, the number of loops and the average size of loops, for each epoch.

6.15.3 The influence of random initial configurations

First it is necessary to determine if simulations having the same space size and initial monomer density but different initial configuration will behave very differently. Based on all of the simulations for each different category, the answer is NO; their behavior was not influenced by the initial random configuration. All simulations sharing the same parameters except the initial random configuration reveal the same characteristic behavior. Therefore, this indicates that the initial random pattern does not substantially influence the characteristic of the emergent self-replicating behavior of the rule set. This can be seen in the four examples of Figure 6.21 and Figure 6.22. Note that different types of components were evenly distributed initially to make sure the initial cellular automata configuration was at equilibrium.

In the first example, curves representing the number of active cells are drawn for four different simulations using a space size of 100 by 100 and an initial monomer density of 10%. The behavior of those curves are very similar. The number of active cells for all four simulations grows from 500 and settles roughly at 2600 at epoch 3000. The second example shows curves representing the growth bits for the four simulations over the whole course of 30000 epochs. It is clear that the four simulations have almost identical behavior despite having different initial random configurations.

In the third example and the fourth example, curves representing the number of loops in the cellular automata space and the average size of loops for another four simulations using a space size of 200 by 200 and an initial monomer density of 30% are drawn for comparison. Again, behaviors of those curves are very similar. The similarity of behaviors is found throughout all simulations using the other space sizes and initial monomer densities. I conclude that the initial random configuration does not substantially influence the behavior of the emergent cellular automata rule set.

6.15.4 The influence of initial monomer densities

Since the initial random configuration do not appear to influence the behavior of simulations, next the behavior of simulations having different initial monomer densities but the same cellular

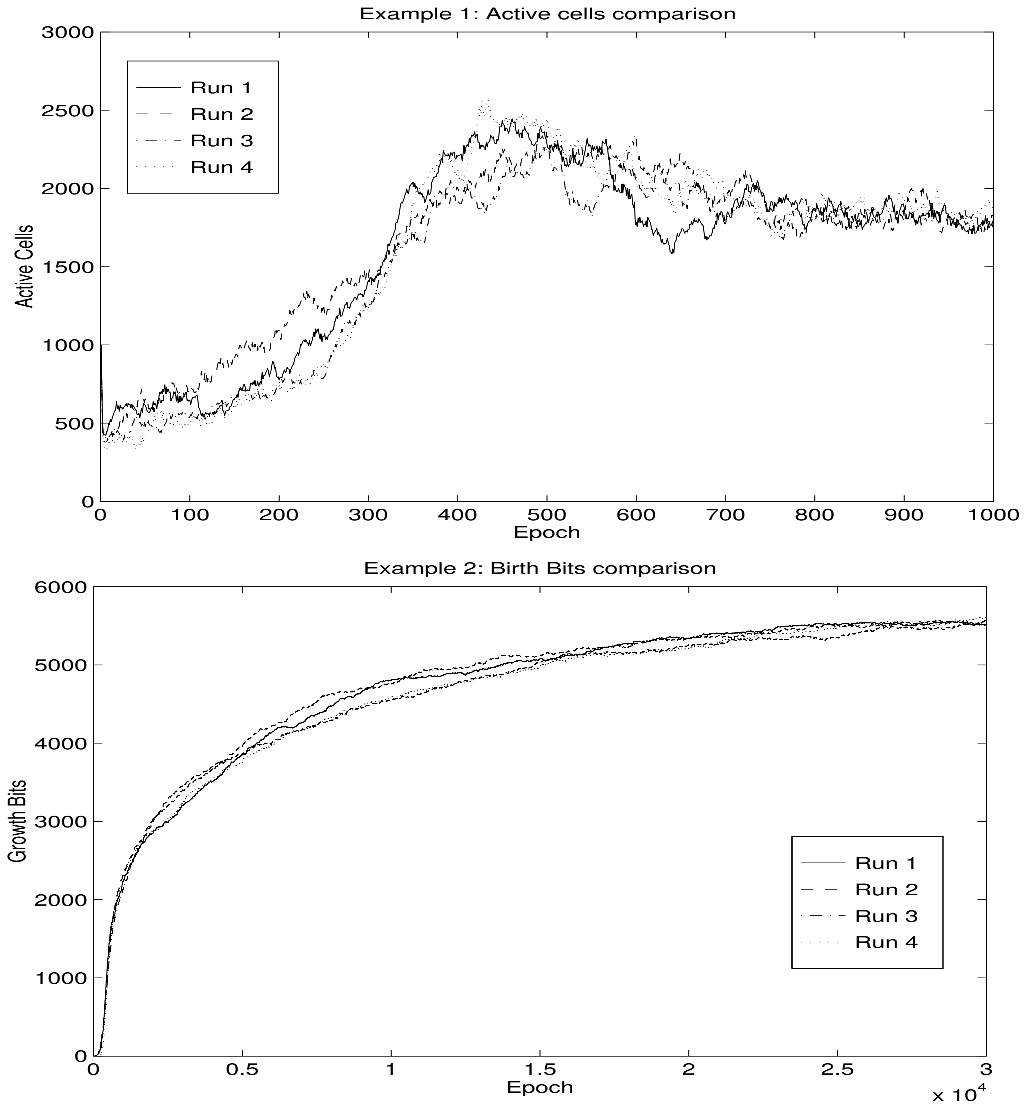


Figure 6.21: Comparison of simulations having different initial configurations, part one. In the first two examples, simulations for a 100 by 100 cellular automata space with initial monomer density of 10% are compared. In example 1, four different simulation runs each with a different random initial configuration are compared. Shown in the graph are the number of active cells during each epoch for the first 1000 epochs. We can see that although these four simulations have different curves, their trend of change are very similar. In example 2, the number of growth bits in the cellular automata space for the same four simulations are compared for the whole length of the simulation. Again, these curves are very much the same.

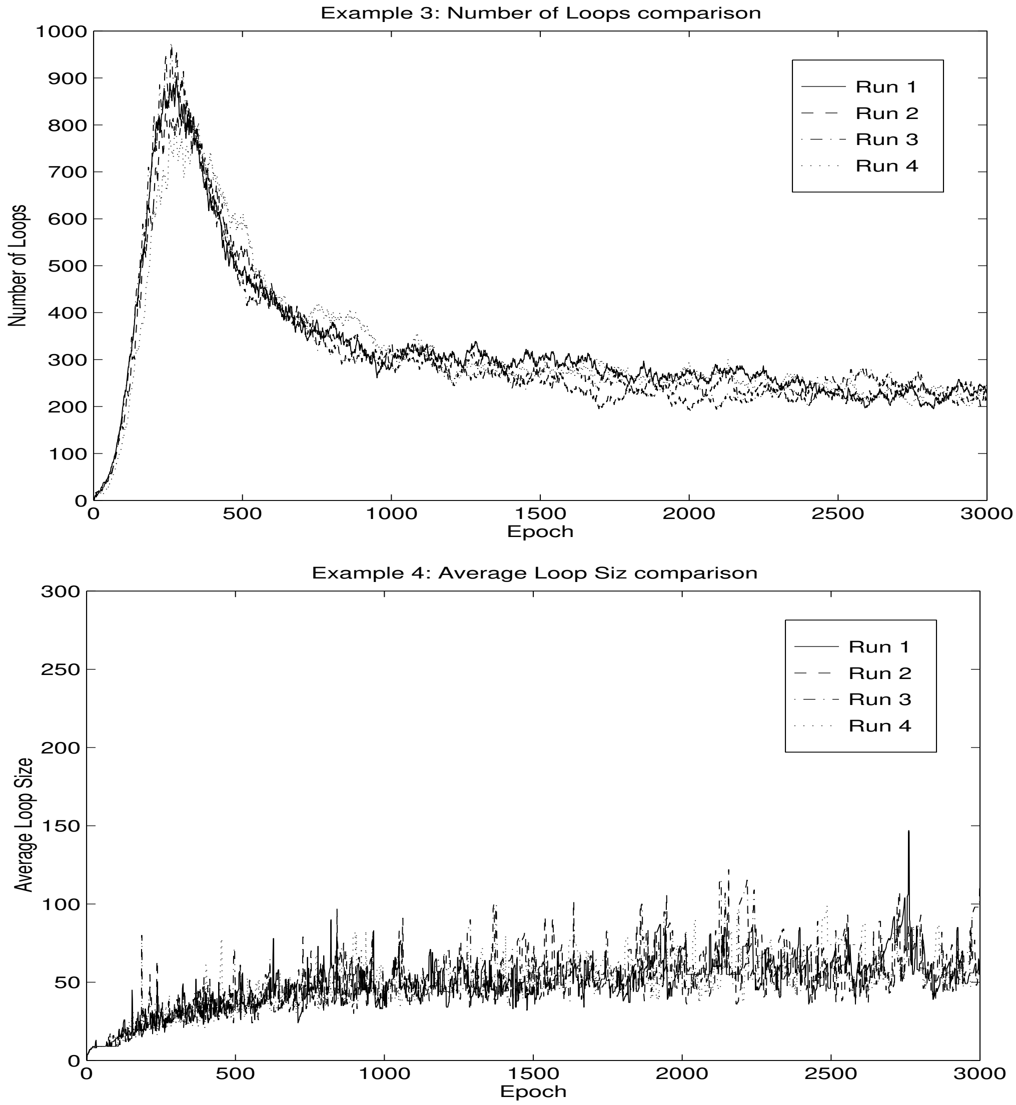


Figure 6.22: Comparison of simulations having different initial configurations, part two. In the next two examples, simulations for a 200 by 200 cellular automata space with initial monomer density of 30% are compared. In example 3, the number of loops during each epoch for the first 3000 epochs are shown for four different simulations. In example 4, the average loop size for each simulation is shown. We can clearly see that although all curves are different, they behave very much the same way.

automata space is compared, using one case for each density only. Surprisingly, the initial monomer density also does not significantly influence the emergent rule set behaviors.

In the example shown in Figure 6.23, the number of active cells is compared for four simulations using a cellular automata space size of 150 by 150, each with an initial monomer density of 10%, 20%, 30% and 40%. The first 1000 epochs are shown in graph A for a closer view, and the first 10000 epochs are shown in graph B to give an overall view. It is clear that no matter what initial monomer density the simulation starts with, the number of active cells gradually approximates the same trend for all four simulations.

In the second example shown in Figure 6.24, the number of loops in the cellular automata space during each epoch are compared for four simulations each using an initial monomer density of 10%, 20%, 30% and 40%, for a cellular automata space size of 100 by 100 cells. The first 1000 epochs are shown in Graph A and the first 6000 epochs are shown in Graph B. We can see that although the curves are somewhat different for the four simulations, especially at the initial stage, they gradually become almost the same after 1000 epochs. It is again seen that the initial monomer density variation does not make any significant difference beyond epoch 1000.

In our last example shown in Figure 6.25, the number of active bound cells (i.e., active cells which belong to a loop) are compared. This example shows four simulations for a 200 by 200 cellular automata space with initial monomer densities of 10%, 20%, 30% and 40%. Again, although initially these four curves are different, they gradually exhibit the same trend. This similarity occurs at all the other cellular automata space size and for all the other data collected. It is concluded that the initial monomer density does not significantly influence the behavior of the emergent self-replication rule set either.

6.15.5 The influence of cellular automata space size

The next thing we consider is whether different cellular automata space sizes will influence the behavior of the emergent self-replication rule set. Of course, since the space size differs, the number of active cells, etc., cannot be in the same data range. But we will see that if we normalize all data by dividing them by the size of the cellular automata space, the ratio results are again similar.

Figure 6.26 shows the active cells comparison. The number of active cells in a cellular automata space is divided by the space size ratio, i.e., the number of cells for a 50 by 50 cellular automata space is divided by 1, the number of cells for a 100 by 100 cellular automata space is divided by 4, the number of cells for a 150 by 150 cellular automata space is divided by 9, and the number of cells for a 200 by 200 cellular automata space is divided by 16. The scaled active cell numbers are then plotted in this figure. All simulations start with a 10% initial monomer density. We can see from both the short term and long term graphs that the behavior of the emergent self-replication rule set as reflected in the number of active cells per area is linearly scalable and independent of the cellular automata space size.

In the second example of Figure 6.27, the number of growth bits are scaled and compared for simulations having four different cellular automata space sizes. The initial monomer density is 30%. Again, although there are slight variations, the general trend for all curves is the same. Therefore, we can conclude that the growth bit behavior of the emergent self-replication rule set is also scalable and space size independent.

In example three, the short term and long term behaviors of the number of loops in the cellular automata space are compared. These simulations all start with an initial monomer density of 40%.

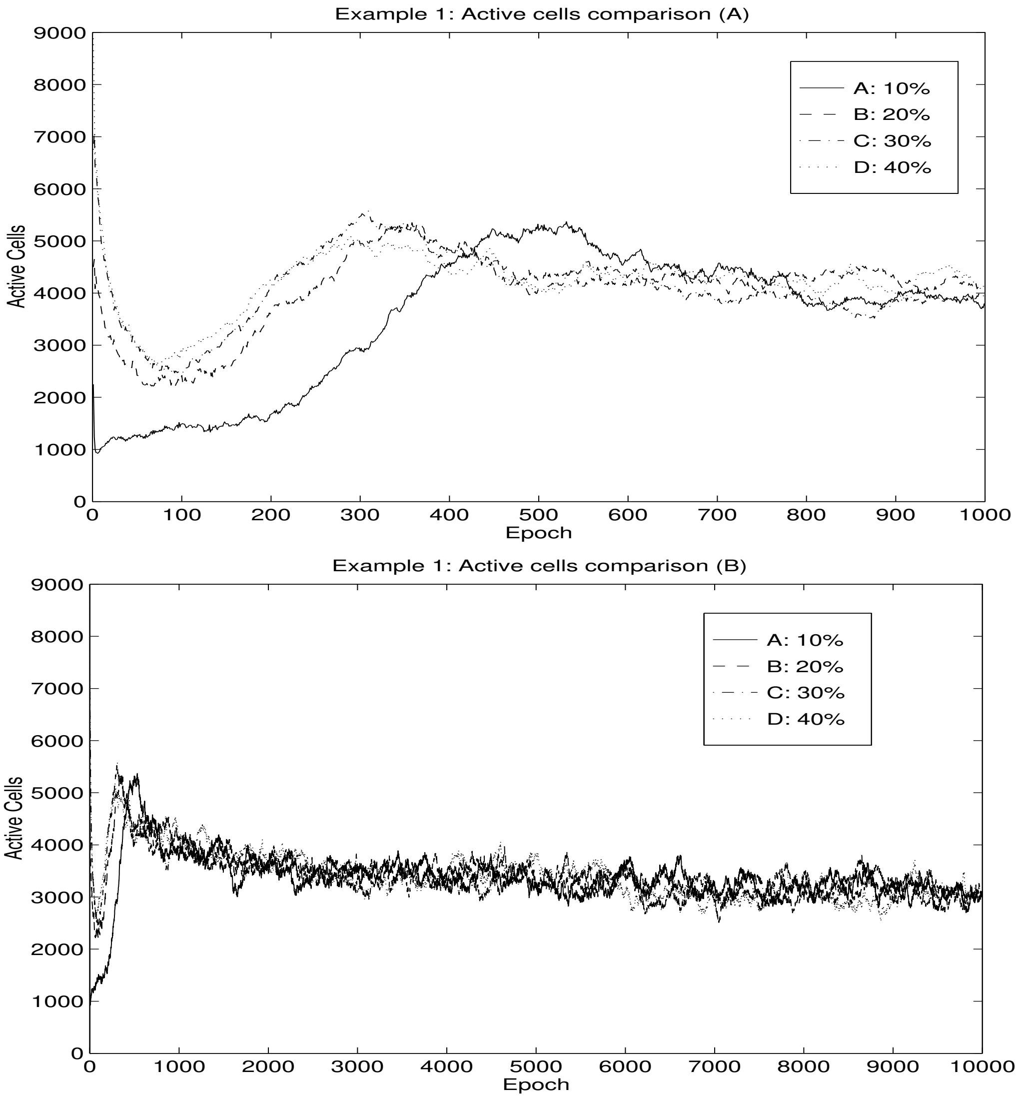


Figure 6.23: Comparison of simulations having different initial monomer densities, part one. In this example simulations for a cellular automata space of 150 by 150 cells are compared. The four different initial densities used for the four simulations are 10%, 20%, 30% and 40%, as shown. Graph A is a closer look at the first 1000 epochs and Graph B shows the first 10000 epochs, both are representing the number of active cells in the cellular automata space. It is clear that these four simulations have similar behaviors.

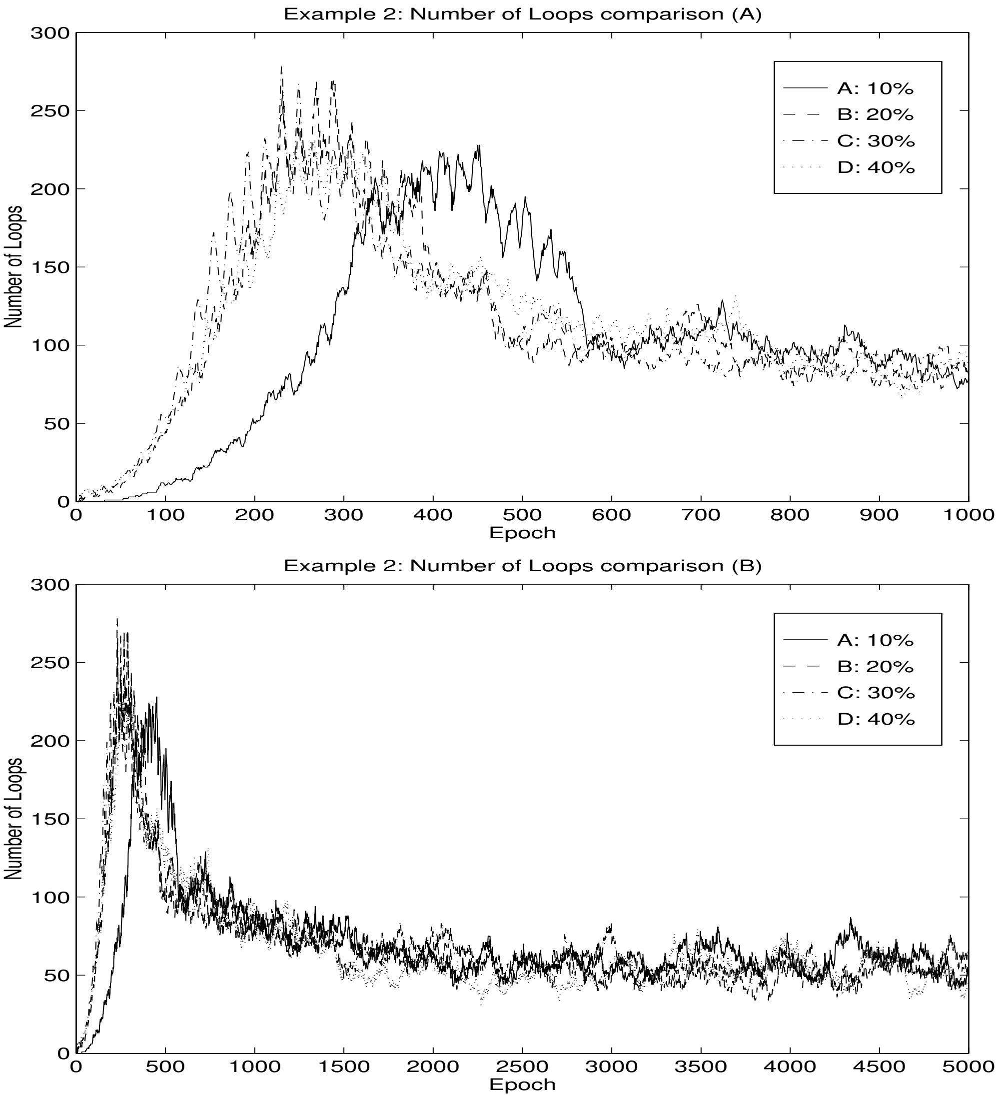
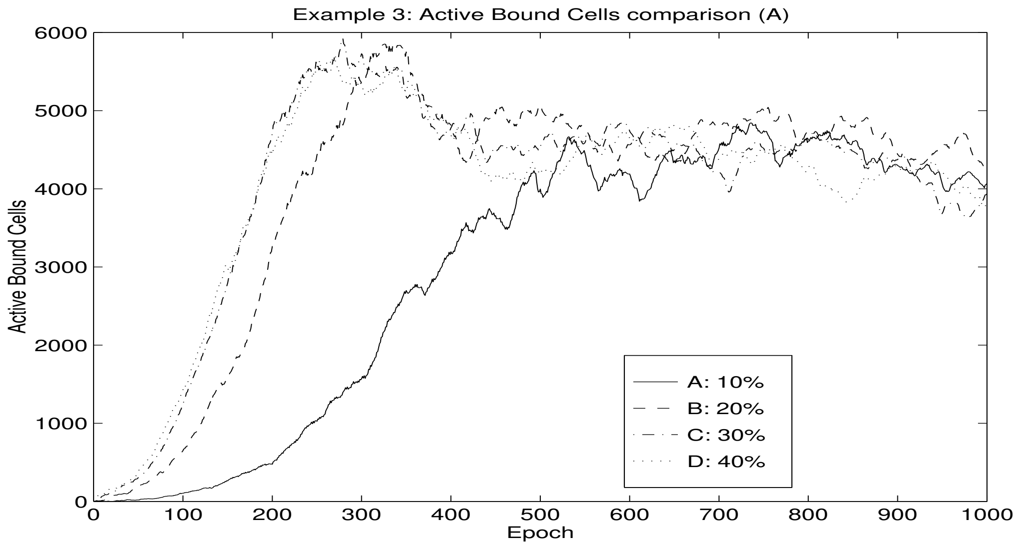


Figure 6.24: Comparison of simulations having different initial monomer densities, part two. In this example simulations for a cellular automata space of 100 by 100 cells are compared. The four different initial densities used for the four simulations are 10%, 20%, 30% and 40%. The number of loops are compared in this example. Graph A gives closer look at the first 1000 epochs and Graph B compares the first 6000 epochs. It is clear that these four simulations have similar behaviors.



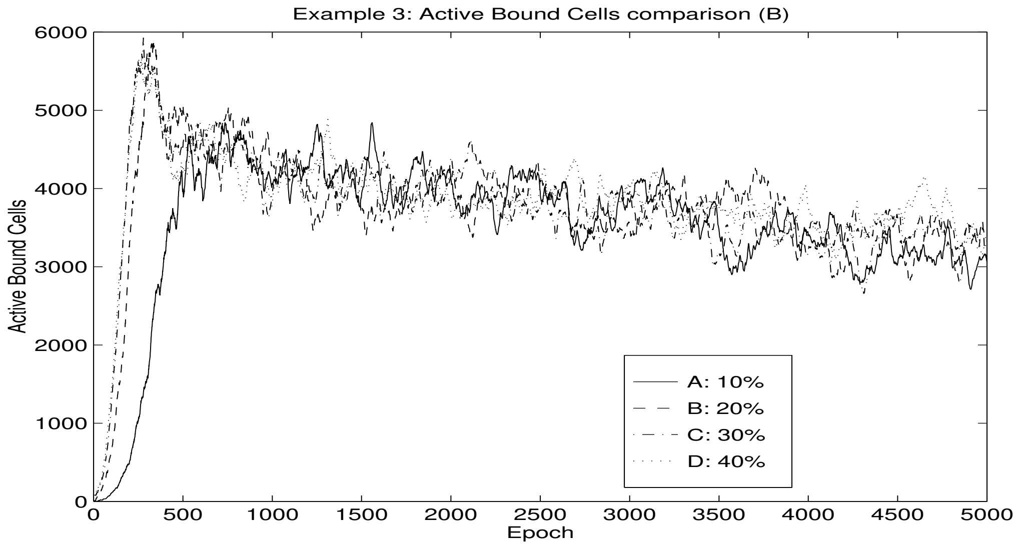


Figure 6.25: Comparison of simulations having different initial monomer densities, part three. In this example simulations for a cellular automata space of 200 by 200 cells are compared. The four different initial densities used for the four simulations are 10%, 20%, 30% and 40%. The number of active bound cells are compared in this example. Graph A gives closer look at the first 1000 epochs and Graph B compares the first 6000 epochs. It is clear that these four simulations have similar behaviors.

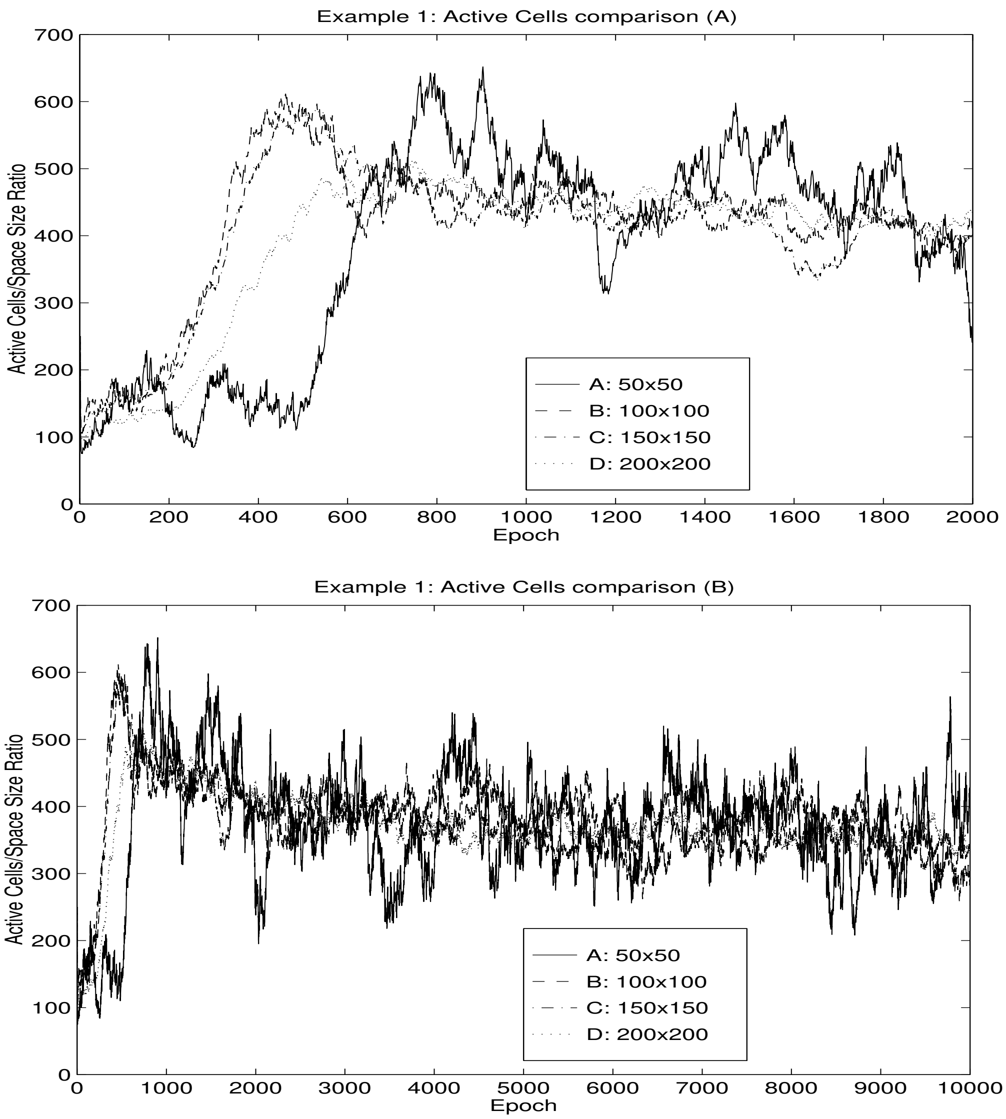


Figure 6.26: Comparison of simulations having different cellular automata space size, part one. In this example simulations with a 10% initial monomer density but four different cellular automata space sizes are compared. The number of active cells in the space is scaled by the cellular automata space size ratio before being plotted in the graphs. We can see that all curves are very similar, although the smaller the cellular automata space size, the more noise in the curves, which is understandable in view of the small cellular automata space (50 by 50 cells) where regional fluctuations have profound impact on the global data values. Larger cellular automata spaces tend to even out the effect of local variations.

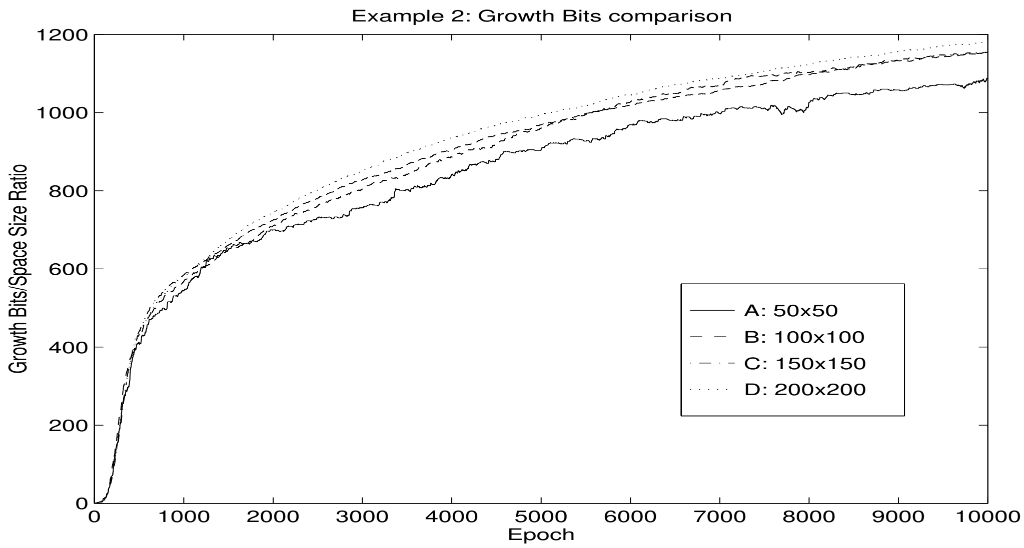


Figure 6.27: Comparison of simulations having different cellular automata space size, part two. The number of growth bits in the cellular automata space for four different simulations are compared. The starting monomer density is 30%. It is obvious that the growing pattern of the curves are very coherent.

It is clear that curves for smaller cellular automata spaces tend to have greater variances, but still, the general trends are the same. This and other similar behaviors reflected in other simulations show that the number of loops per unit area for the emergent self-replicating rule set is also independent of the cellular automata space size.

In the previous three examples, simulations having the same initial monomer density were compared in order to limit changing factors in the comparison. Since we have found that the initial monomer density does not influence the general behavior of the rule set, actually it can be seen in the fourth example shown in Figure 6.29 that even simulations having different initial monomer densities are very coherent. The growth bit curves for four simulations having four different cellular automata space sizes as before and four different initial monomer densities 10%, 20%, 30% and 40% were compared. The growth bit data is scaled by the cellular automata space size ratio in the graph shown. The similarity of the curves shows again that both of our conclusions are correct, that the growth bit is space size independent and that the initial monomer density is not influential. Curve A has the largest variance due to the smallest cellular automata space size.

It seems that the only thing which is not linearly scalable is the average loop size for different simulations. In Figure 6.30, both the scaled and un-scaled average loop size curves are drawn for four simulations having an initial monomer density of 30%. It can be seen that although a larger cellular automata space tends to allow larger loops, the scaling is not linear. In the lower graph where average loop sizes are not scaled by the cellular automata space size ratio, the curve for a 200 by 200 space seems to be higher than the other curves. When scaled in the upper graph, it becomes the lowest in the four curves. Obviously, the scaling factor for the average loop size is somewhere between 0 and 1, but much less than 1.

To sum up, the following properties of the emergent self-replication rule set have been observed:

- Different simulations having the same cellular automata space size and initial monomer den-

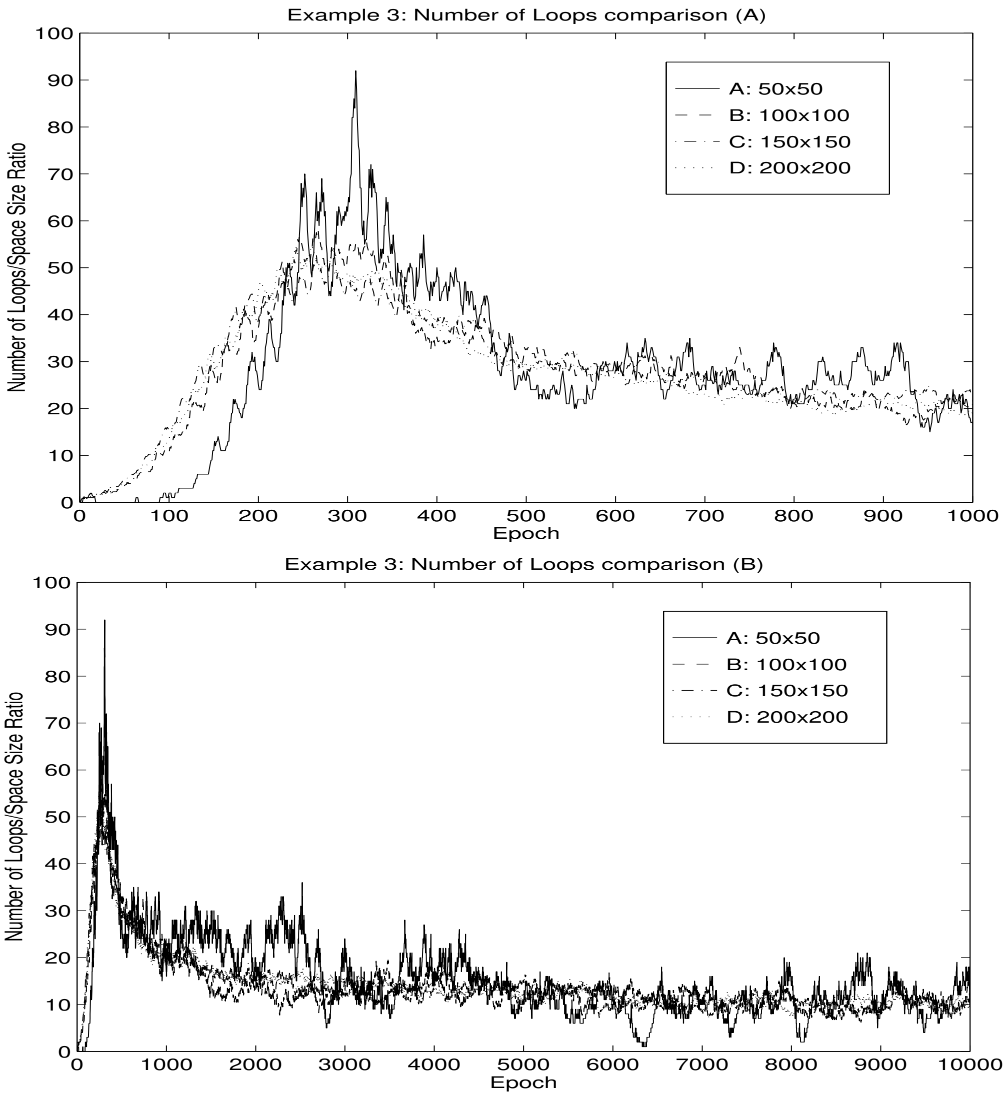


Figure 6.28: Comparison of simulations having different cellular automata space size, part three. The number of loops in the cellular automata space is scaled by the space size ratio for the four simulations. It is clear that the pattern of change is very similar despite the different cellular automata space size.

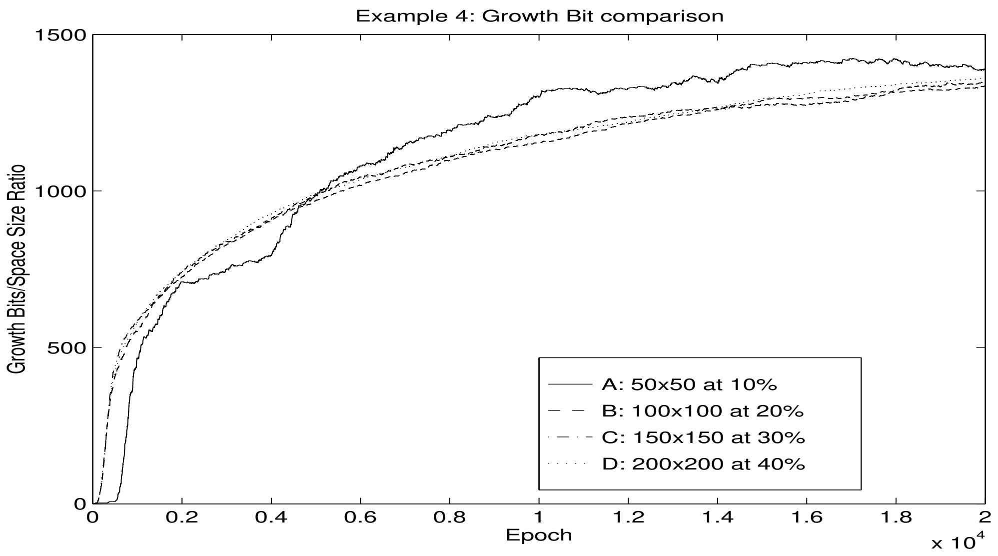


Figure 6.29: Comparison of simulations having different cellular automata space size, part four. The scaled number of growth bits for simulations having different cellular automata space size and initial monomer densities are compared.

sity but different random initial configurations behave the same;

- Different simulations having the same cellular automata space size but different initial monomer densities behave the same; and
- Different simulations having different cellular automata space sizes and initial monomer densities behave the same in all aspects except the average loop size once these data are scaled according to the cellular automata space size ratio.

Because of these observations, in the following discussions of various behaviors of the emergent self-replication rule set, we will use simulations with a 200 by 200 cellular automata space size as examples, presuming that the other simulations will behave similarly. These simulations produce curves more stable than simulations with a smaller space, due to their larger sampling areas.

6.15.6 Behavior analysis

In the upper graph of Figure 6.31, the number of active cells and the number of bound cells are compared for a simulation with a 200 by 200 cellular automata space and an initial monomer density of 30%. Remember that bound cells must be active cells, too. We can see that these two curves are very much synchronous with each other, with an almost constant difference between their values. An enlarged view for these two curves is shown in the lower graph; the similarity is obvious. The average difference between these two curves is 2480 cells, the number of monomer cells. The same average differences for the other three similar simulations with an initial monomer density of 10%, 20% and 40% are 2475, 2477 and 2473. It is clear that this value is very stable and is a property of the emergent self-replication rule set not related to the other parameters. The

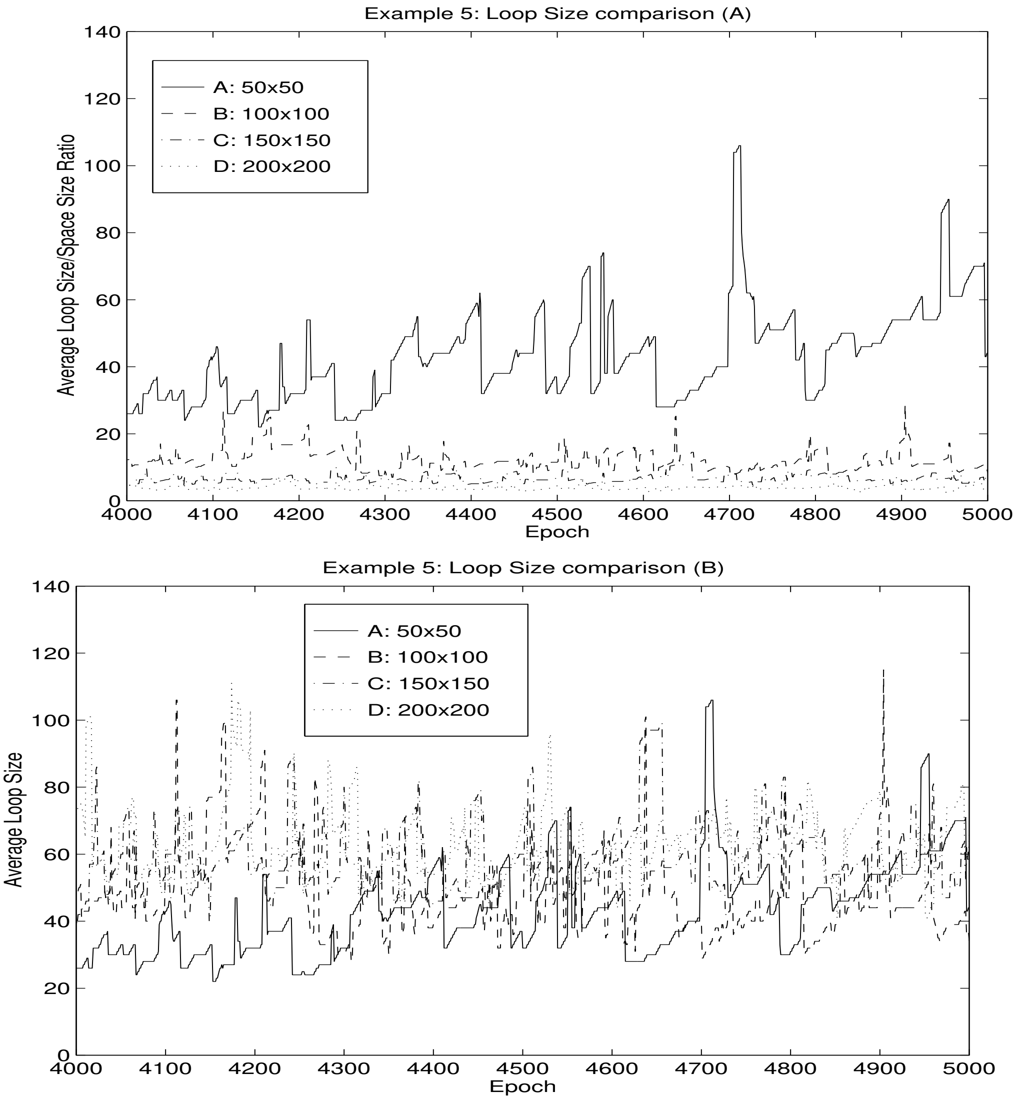


Figure 6.30: Comparison of simulations having different cellular automata space size, part five. In the figure, the average loop size for each simulation are compared. In the upper graph the size is scaled by the cellular automata space size ratio. In the lower graph, it is not scaled. We can see that generally the position of curves are reversed in these two graphs, i.e., the curve for a 200 by 200 space size is the lowest in the upper graph but the highest in the lower graph. This means that the average loop size is dependent upon the cellular automata space size, but the scaling factor is not 1. It must be somewhere between 0 and 1.

average monomer density in the cellular automata space is very stable over time and is about 2480 per 40000 cells.

In order to determine if this constant population of monomers is the natural tendency of the game of life rule which I used to translate the monomers, or if it is the joint property of the self-replicating loop rules and the game of life rules together, some simulations where bound cell were not allowed to be generated were conducted. The result is that none of these simulations can keep a constant monomer population as high as that of 2480 in a 200 by 200 cellular automata space. Actually, all of them cease activities early before reaching 30000 epochs. In conclusion, monomers in the cellular automata space are *co-evolving* with the self-replicating loops. Without either monomers or self-replicating loops, the other kind will not last very long in the cellular automata space.

The behavior of active cells can be seen in the lower graph of Figure 6.31. At the beginning, there are 12000 active cells, or 30% of the total cells, in the space. This number drops rapidly since the game of life rule alone is not capable of holding such a high number of active cells. If no bound cell is formed, the number will never climb back. After just a little while bound cells start forming in the space; they take the active cell curve with them from then on, controlling the shape of the curve.

There is an initial surge of bound cells as can be seen in Figure 6.31. This is due to the initial blooming of small and fast replicating loops. Gradually, the mutual competition between them balances out and lowers the curve.

The changing of growth bit number is shown in Figure 6.32 for a 200 by 200 cells simulation with 30% initial monomer density. Remember that the growth bit stimulates size expansion of the self-replicating loops. They are generated when a loop dies and consumed by loops during their expansion process. The number of growth bits climbs rapidly at the beginning, but gradually levels off at 22500, when consumption and generation of growth bits are balanced. The final growth bit density is balanced at 56%.

When the number of growth bits levels off, the number of loops seems to stabilize too. In Figure 6.33 the long term behavior of the number of loops for the same simulation is shown. After 10000 epochs, there is no significant change of the average number of loops in the cellular automata space. The number settles at 147 loops in a 200 by 200 space. In the beginning of the simulation, there is an overshoot of the number of loops. This is caused by the same reason that the number of bound cells has an overshoot, that smaller loops are blooming at the beginning but slow down when competition pressure raises.

The average loop size varies a lot at first glance at its long term behavior, as shown in the upper graph of Figure 6.34, here the average loop size for the same 200 by 200 cells simulation with 30% initial monomer density is shown for the whole course of the simulation. But if we look closer at the curve, such as in the lower graph, where a detailed portion of the curve is shown, we can find out that actually the average loop size curve is going through a constant cycle of up and down. This behavior is in exact accordance with what we observed in the cellular automata space during the simulation. Recall from Section 6.2 that loops in the cellular automata space have the tendency to grow bigger and bigger, until when there is no more space to grow any further, then the bigger loops will disappear, replaced by smaller loops again. It is this cycling behavior which produces the zigzag shape of the curve.

The average loop size after the cellular automata space has reached a stable condition is about 73 cells for a 200 by 200 cellular automata space size. As noted before, the average loop size is

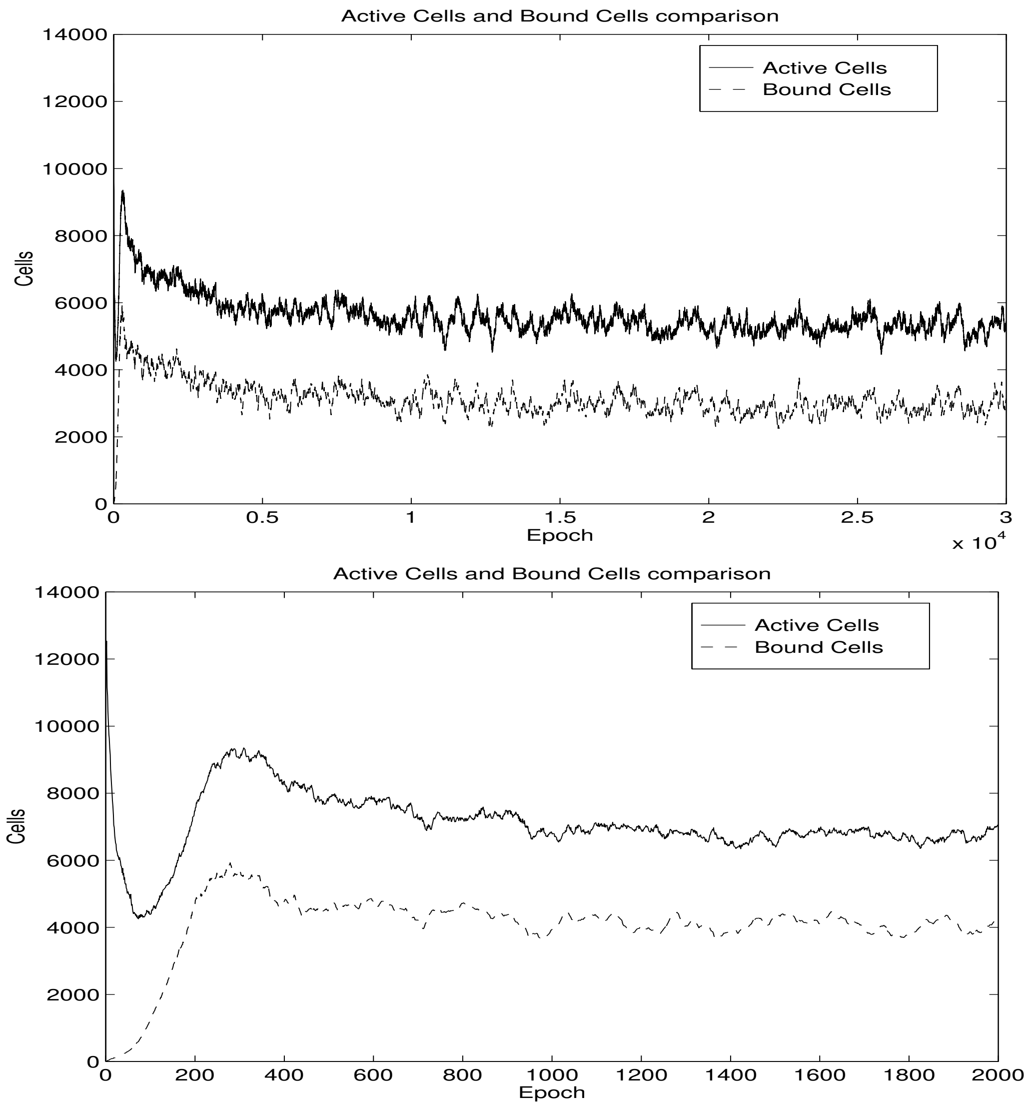


Figure 6.31: The relationship between active cells and bound cells. The number of active cells and active bound cells are plotted in these two graphs for a simulation with 200 by 200 space size and 30% initial monomer density. These two curves are very similar. The lower graph is an expanded view of the first 2000 epochs of the top graph.

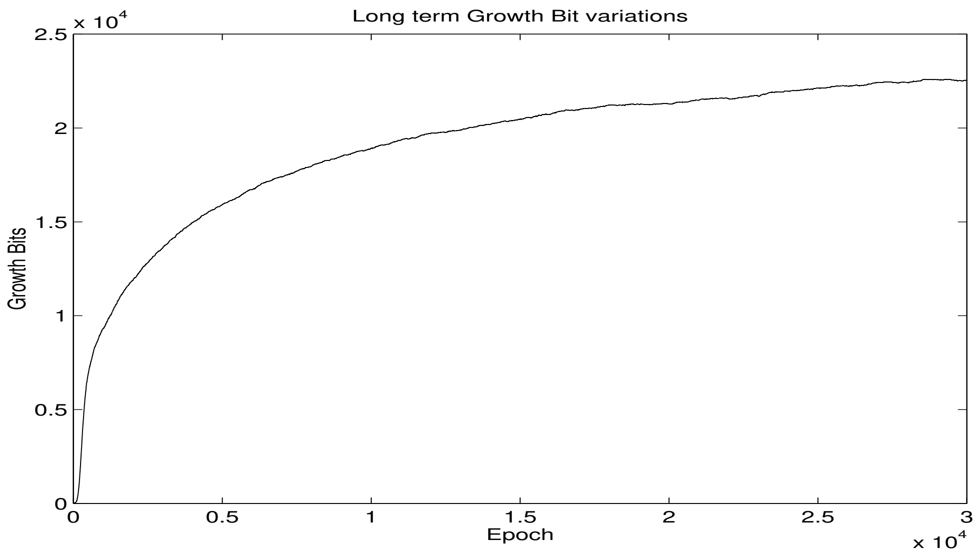


Figure 6.32: The long term behavior of the growth bits. This is for a simulation with 200 by 200 cellular automata space size and 30% initial monomer density.

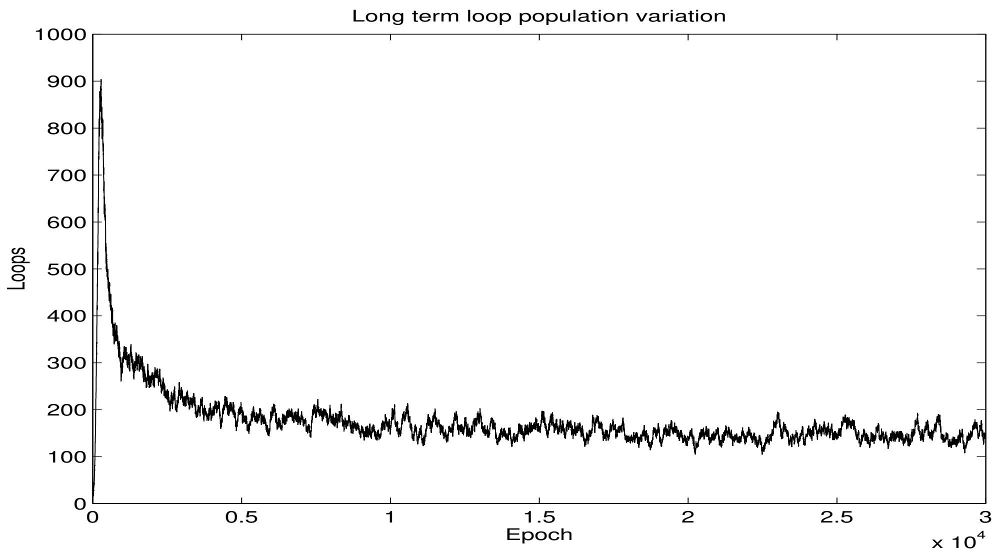


Figure 6.33: The long term behavior of the number of loops.

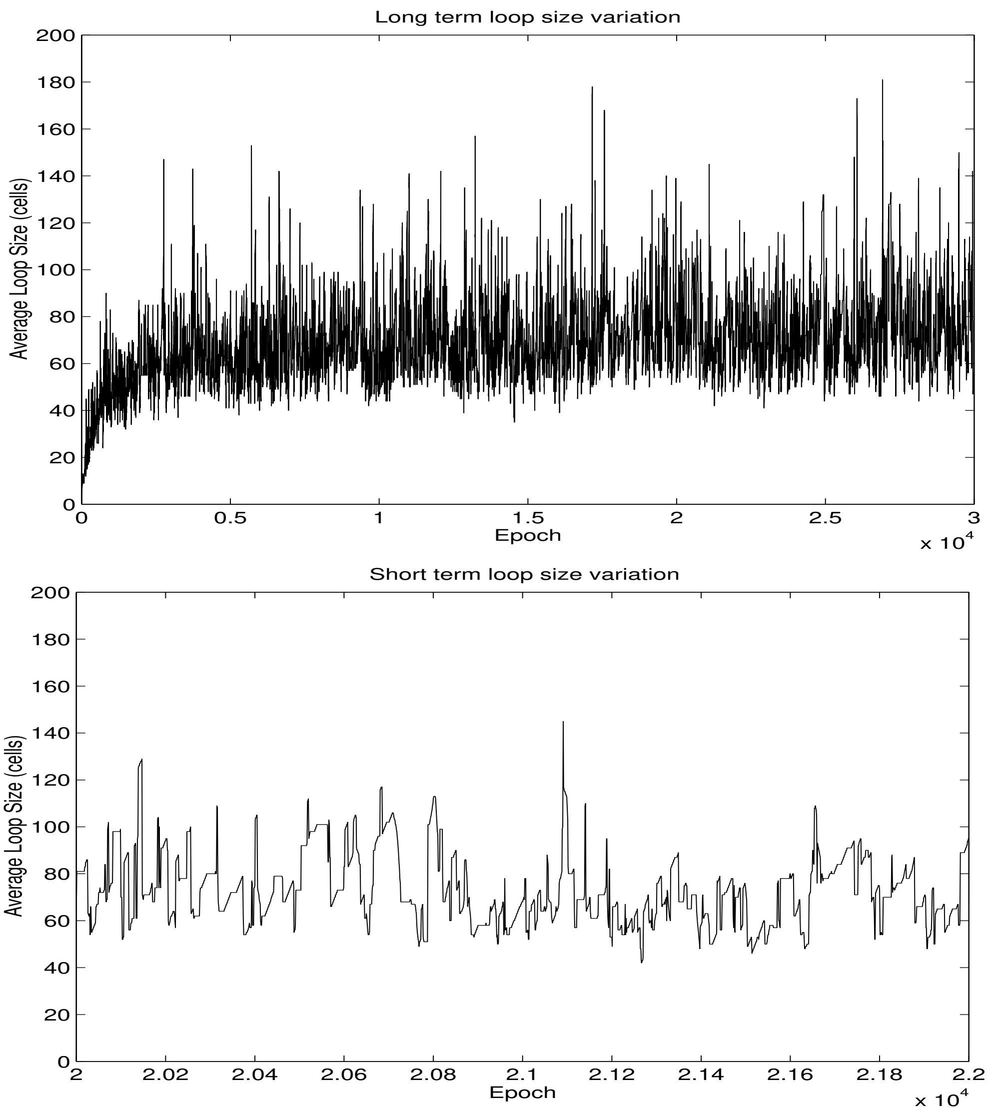


Figure 6.34: The long term behavior of the average loop size. In the upper graph the average loop size of each epoch for the whole course of the simulation is shown. In the lower graph the same data but only for epochs between 20000 and 22000 is shown. It is clear these graphs that the average loop size is constantly cycling within a range of values.

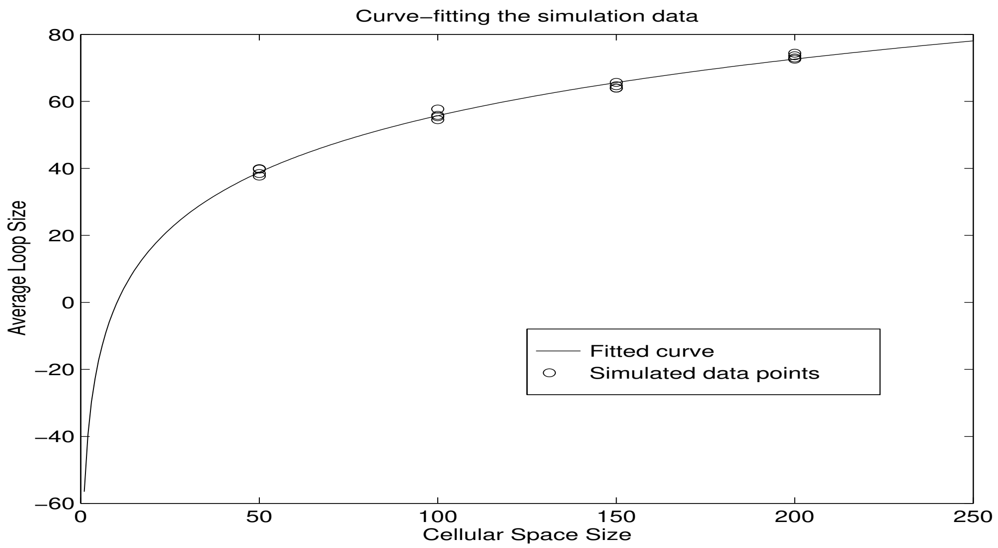


Figure 6.35: The comparison of simulation data and the fitted curve. The average loop size for each simulation are marked by the 'o' marks. The fitted curves are shown with the line.

not linearly scalable with the cellular automata space size. The average loop size for 150 by 150 cellular automata space size is 64 cells, for 100 by 100 cellular automata space it is about 56 cells, and for the smallest 50 by 50 space it is about 39 cells. A two terms curve-fitting with the available simulation data reveals the following logarithmic relationship between the cellular automata space size and the average loop size we can get in such a space:

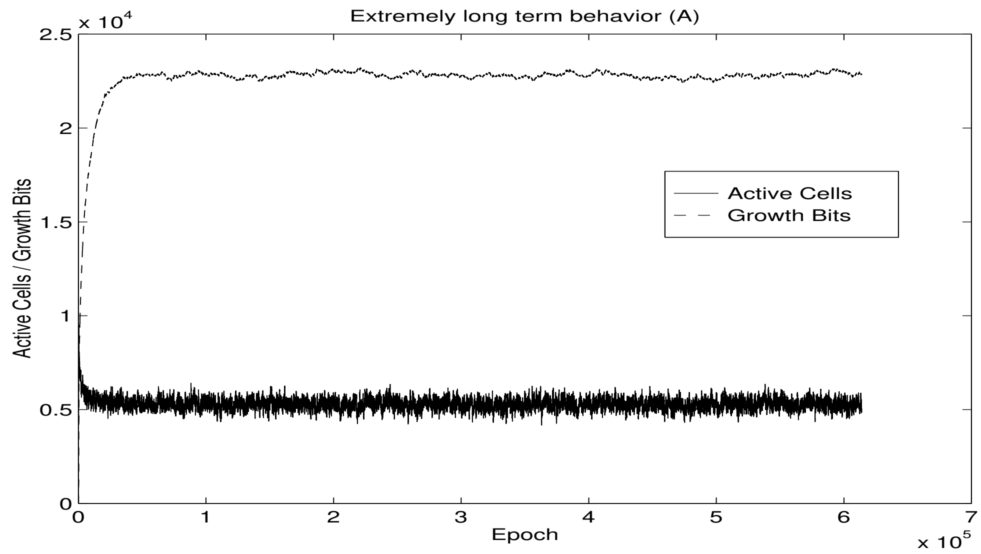
$$\text{loop size} = -56.5291 + 24.3794\log(\text{space size})$$

The fitted curve and the simulation data are compared in Figure 6.35.

6.15.7 Extremely long term behavior

In the previous sections, all simulations were run for 30000 epochs. Although this number is big, it is still possible that some change in the cellular automata space status can happen beyond that range. To determine if such a case is possible, an extremely long simulation was done with a 200 by 200 space size and an initial monomer density of 30%. This simulation was allowed to run for 613920 epochs, about 20 times longer than the normal simulation length. The results show no sign of behavior changing once a stable status has been reached. Some of the data are shown in Figure 6.36.

Since the status of the cellular automata space seems to be very stable in the graphs shown in Figure 6.36, it makes one wonder if the cellular automata space is going through some sort of cycle, i.e., whether the configuration of the cellular automata space is repeating itself. To study this, the accumulated data of the simulation for each epoch were cross-compared with each other. Monomer numbers are not taken into account in the comparison since an exact duplication of the cellular automata monomer configuration at different epochs is very unlikely for a 200 by 200 cellular automata space. Checking for repeated epochs was conducted by comparing only the position and



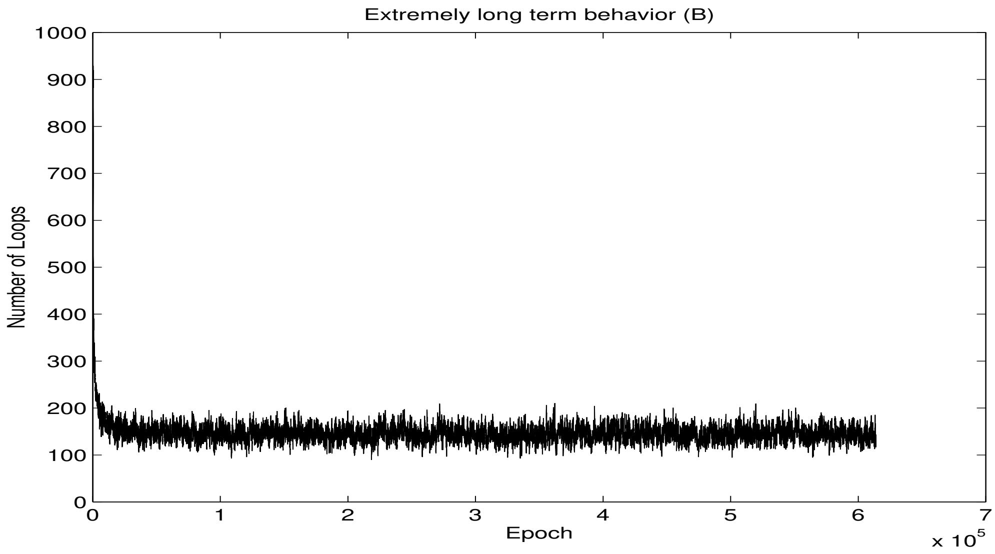


Figure 6.36: Extremely long term behavior of the emergent self-replication rule set. In the upper graph, the number of active cells and the number of growth bits are shown. In the lower graph, the number of loops are shown. All are for a simulation running till 613920 epochs. It is obvious that the cellular automata space is maintaining a dynamic equilibrium status after the initial transient stage. This equilibrium does not change through time.

the number of self-replicating loops in the cellular automata space. Even with this somewhat less stringent comparison criteria, no duplication is found in the comparison.

6.16 Discussion

In this chapter, an emergent self-replicating cellular automata rule set was presented. It allows a randomly initialized cellular automata space to spontaneously generate self-replicating structures. This is the first cellular automata emergent rule set does this, which allows arbitrary size loops to form.

The rule set used here is based on a central design, a design which allows loops of arbitrary size to replicate in the cellular automata space. This is called the *general purpose* self-replication rule set. In addition to that, mechanisms which allow loops to grow in size are supported in the rule set. These features, together with a four cell minimal self-replicating loop, produce a whole family of self-replicating structures in the cellular automata space, all supported by the same rules.

This is the first demonstration that it is possible to produce self-organization behaviors in a random cellular automata space. The generated self-replicating loops replicate by following the instructions stored within their own structure. Those instructions are interpreted by the self-replicating loops to produced their offspring, and are translated to their offspring. By carefully designing the cellular automata rules, or the physics of the cellular automata universe, it is possible that such an entropy decreasing, self-organization behavior can naturally occur, at least in an artificial cellular automata environment. This provides key insights to how an autonomous self-replicating entity can be constructed in the cellular automata space.

Systematic simulations showed that the emergent self-replication rule set provides rich activities in a cellular automata space of varying size, yet the global property of the rule set is generally independent of any particular simulation run and initial parameters. It always leads to the same behaviors when given enough time to stabilize. This kind of "bound to happen" characteristic of the rule set is very encouraging, since it provides a stable foundation for further research.

There are some future possibilities for expanding the emergent self-replication rule set. For one, the self-replicating loops can be allowed to carry not only code for replication control, but also code for evolution. One form of this expansion is studied in the following chapter. Another possible future direction is to support other different structures to replicate, in addition to the loop form.

6.17 The complete rule listing

The actual Trend program used to simulate all examples in this chapter is given here. There are several things which make the program a little different than the individual rules I introduced before. First, for efficiency, I use more *else-if* statements than plain *if* statements in the program since an *else-if* statement will prevent the following codes from being evaluated if its condition is selected, which makes the Trend program more efficient for cellular automata simulations.

Second, rules are organized according to the component types they are dealing with, rather than functions of the rules as was the case in organizing the sections in this chapter. This provides clear and concise program code.

Third, for each component type some preferences and priorities must be decided for each individual rule while ordering them in the program. Generally, special cases are always considered first before normal operations. This organization will not influence the cellular automata behavior in any way.

This Trend program is heavily commented to make it self-explanatory. In various areas of the program the relevant section numbers in this chapter where the associated features are discussed were marked. This is denoted by the § symbol followed by the section number(s).

```
// ****************************************************************
// ****************************************************************
// Emergent Self-Replicating Rules
//                                  written by Hui-Hsien Chou
// ****************************************************************
// ****************************************************************
//
// This Trend rule set defines a CA space which allows arbitrarily
// chosen initial CA configurations to generate self-replicating loops
// over time.
//
// ****************************************************************
// *****************    Default  Rules         ******************
// ****************************************************************
// The default action is to maintain no change if none of the rules
// explicitly specifies a next state value for each field. Therefore,
// the current value is copied over to the next state for each field.

default component=component;
default special=special;
default growth=growth;
default bound=bound;

// ****************************************************************
// ***********  Variable and Function Declarations  ***************
// ****************************************************************
// count: a temporary variable used to store the count of neighbors
//      which meet a certain condition.
// value: a temporary variable used to accumulate component values
//      from the neighbors. Used to determine the signal for a new
//      monomer if birth is occurring.
// y: a nbr variable used for looping through all neighbors.

int count, value; nbr y;

// AbnormalNeighbor is a function which determines if a bound cell may
//      have run into a failure situation where too many or too few
//      neighbors are around it. This usually means that a collision
//      with another loop has occurred. If that's the case a dissolve
//      mark will be generated next (in the main code below).

int AbnormalNeighbor() {                // discussed in §6.10
       count=0;         // clear the counter
                    // count the no. of non-quiescent neighbors.
```

```
    over each other y:
        if (y:component) count++;
            // if zero or more than 5, it's abnormal!
    if (count && count<=5)
        return 0;
    else return 1;
}


// Main program starts here!

// *************************************************************
// ***************      Rules for Monomers!     ****************
// *************************************************************

if (bound==0) {                    // discussed in §6.14

  // A cell is obeying the rules for monomers when its "bound" field
  // is set to zero. In this mode the monomer cell basically obeys
  // a Game-of-Life-like rule set, except that some additional rules
  // to deal with non-monomer actions are included.

  // First the neighbors are scanned, non-quiescent neighbors are
  // counted in variable count, and their component values are
  // accumulated in variable "value". This accumulated value is later
  // used to determine the component value of a new active cell.

  count=0; value=0;                 // reset to zero
  over each other y: {
    if (y:component) {
      count++;                      // count the no. of active neighbor
      value = value + y:component; // and accumulate their values
    }

    // see if a bound cell is a neighbor; if yes, the current
    // cell must be returned to quiescent state and the bound flag
    // marked. This is to make a shield around bound cells (loops)
    // to protect them. The magic value 99 was stored in count for
    // checking below. Note that the only case a bound neighbor
    // cell won't cause a problem to the current cell is when it
    // itself is in the dissolve mode. See the rules for loops for
    // details about the dissolve mode. It is marked by a # mark in
    // the "special" field.

    if (y:bound && y:component && y:special!='#') {
      count = 99;                  // make a special flag in count
      break;                       // then stop scanning, get out
                                   // of the over command.
    }
  } // close of the over command.
```

```
// After the scanning of neighbor cells, various actions are taken
// to determine the next state values for the current cell.

// case 1: if there is a bound cell as a neighbor, the current
// cell should return to quiescent state and its bound bit set.

if (count==99) {
    component='.';
    bound='!';
}

// case 2: if the current cell is just being dissolved from a loop
// (by having a nonzero "special" field), we make one more epoch
// delay so that it won't be killed by its own fellow cells which
// are also dissolving from the same loop.

else if (special)
    special=0;

// case 3: A particular configuration has been designed to be the
// birth initiation configuration. When such a configuration of
// monomers occurs in the CA space, it will form a 2x2 bound loop in the
// next epoch. The configuration is (note that rotation is possible):
//
//                  OO
//                  L>
//
// Since each member of this "Adam" configuration can see each
// other within its own neighbors, we just need three different
// checks for the different components >, L and O to make them do their
// own "changing to bound mode" work. Signal > needs one more step of
// setting the "special" field to * to extrude an arm later, so
// replication can begin.

else rot if (component=='>' && we:component=='L' && nw:component=='O' &&
                  no:component=='O') {
    bound='!'; special='*';
}
else rot if (component=='L' && ea:component=='>' && no:component=='O' &&
                  ne:component=='O')
    bound=1;
else rot if (component=='O' &&
          (so:component=='L' && se:component=='>' && ea:component=='O'||
           so:component=='>' && sw:component=='L' && we:component=='O'))
    bound=1;

// other than above special cases, the current cell will just
// follow a traditional Game of Life rule. If a birth will occur
// the new value of the birth cell will be determined by the
// accumulated value in the "value" variable.
```

```
    else if (count<2 || count>3)                // the death rule
        component='.';
    else if (component=='.' && count==3)        // the birth rule
        component = (value+1)%6+1;

} else {                // closure of the "if (bound==0) ..." statement

// ****************************************************************
// ****************        Rules for Loops!       ******************
// ****************************************************************

    // If a bound cell is in the dissolve mode, it will be dissolved to
    // monomer (i.e., lose bound bit) in the next epoch, no matter
    // what happens in the neighborhood.

    if (special=='#') bound=0;                  // discussed in §6.10

    // If a bound cell sees a dissolving cell in its neighborhood,
    // it itself will be going into that mode too. This dissolving
    // mark was represented by a # in the "special" field of the
    // neighboring cells, and is given the highest priority to
    // consider. Thus when one cell in a loop structure goes into this
    // mode, the whole loop will be taken into that mode in a very
    // short time, and the whole structure dissolves.
    // The only exception is the D component. It is used to separate
    // newly formed loops from their parents. Since this is a
    // separating component, it is considered not part of a loop, so the
    // dissolve mark # won't pass through it. Speaking in another way,
    // the dissolving of parent and child loops is independent. This
    // helps to preserve the parent loop even when the replicating is
    // unsuccessful, or vice versa.

    else rot if (component && component!='D' &&
                    (no:special=='#' || ne:special=='#')) {
        special='#';                            // discussed in §6.10

        // The only time the system will generate the growth
        // stimulation bit on the "growth" field is when a L signal is
        // dissolving. If this line is disabled no growth stimulation
        // bit will be generated, thus no bigger loop will be generated.

        if (component=='L') growth='+';         // discussed in §6.12
    }

    // One of the way the system checks for collision of loops is to
    // check if a cell has too many or too few neighbors (<1 or >5).
    // If these situations happen the cell must be in a wrong
    // structure and must be dissolved.

    else if (component && AbnormalNeighbor())
        special='#';                            // discussed in §6.10
```

```
else {

  // When none of the three special cases above apply, the bound
  // cell is in a healthy loop, and should perform normal loop
  // replicating functions according to its component type.

  // **************** Block O rules ****************
  // Block O is the building block of loops. Normally it will
  // allow the other signals like > or L to pass through it,
  // unless it is in a special clear mode denote by a - mark
  // in the "special" field. In this special mode it will not copy
  // other signals through it, thus cleaning up the signals.
  // This special mode occurs when a hybrid replication
  // situation happens where a smaller loop is generating a
  // bigger loop. Because of the timing difference, an
  // incomplete signal sequence will be copied into the new loop,
  // together with a complete and normal sequence. This
  // incomplete signal sequence must be destroyed.

  if (component=='O') {
    if (special!='-') {              // discussed in §6.9

      // Here the Block O isn't in the clear mode, so normal
      // operations will be done.

      // First check to see if it's in the closing of a hybrid
      // replication by looking at the closing Birth B and all
      // relevant neighbor positions which signify a hybrid
      // replication. If yes, set the special clear mark.

      rot if (no:component=='B' && nw:component=='.' &&
            (we:component=='O' || ea:component=='L'))
        special='-';                 // discussed in §6.9

      // Otherwise, it should pass signals. First check if there
      // is a > signal which wants to pass through. If so, copy
      // it over here.
      // But if there is more than one > signal which wants to
      // pass through, this must be a collision situation, so the
      // special dissolve mode is entered.

      else rot if (we:component=='>') // > signal which wants to
                                      // pass.

        // check for no other signal wants to pass too.
        if (no:component!='>,1' && no:component!='>,2' &&
              ea:component!='>,2' && ea:component!='>,3' &&
              so:component!='>,3' && so:component!='>')
          component='>';  // copy it, discussed in §6.5
        else
```

```
        special='#';    // otherwise, set dissolve mode
                        // discussed in §6.10

    // This is similar, but for 90 degree passing of > in O.
    // Check for collision too.

    else rot if (so:component=='>') // > signal which wants to
                                    // pass .

        // check for no other signal wants to pass too.
        if (we:component!='>,1' && we:component!='>' &&
                no:component!='>,2' && no:component!='>,1' &&
                ea:component!='>,3' && ea:component!='>,2')
            component='>,3';        // copy it, discussed in §6.6
        else
            special='#';    // otherwise, set dissolve mode
                            // discussed in §6.10

    // Extrude by F to complete a new arm. When O is in the tip
    // of a new arm, it will get a > signal when seeing the F
    // signal. This will complete the arm branching step by
    // forming a standard two-cell arm in two steps.
    //                                             O <arm
    //        O               ^                    O
    //      <LOF            LOOO                 OOOO
    //        O     ---->     O       ---->        O
    //        O               O                    O

    else rot if (so:component=='F' && se:component=='.' &&
                no:component=='.' && ea:component=='.'
                && we:component=='.')
        component='>,3';            // discussed in §6.8

    // Check to see if it's in the gap between a parent and a
    // child loop and the replication cycle has just
    // completed, by looking at the closing Birth B or the
    // opposite direction of signal flows on both side, and
    // relevant neighbor positions which signify a closing
    // replication cycle. If yes, change to D to disconnect
    // these two loops.

    else rot if ((nw:component=='>,3' || nw:component=='O') &&
                (ne:component=='B' || ne:component=='>,1')
                && no:component=='.' && ea:component
                && we:component )
        component='D';              // discussed in §6.9
} else

    // if the Block O sees the clear flag -, the flag
    // will be changed to the arm extrude flag when all
    // signals has been cleared (L is the last signal).
```

```
        rot if (we:component=='L')
            special='*';                    // discussed in §6.9
}

// **************** Signal > rules ***************
// Signal > is the extruding signal command. When at the tip
// of a replicating arm, it will cause the adjacent quiescent
// space it points to to change to a Birth B in the next epoch.
// Usually > signals are going in a sequence with a trailing L,
// so it must copy the signal behind it, be it a > or a L, to
// its own current position, unless there is no L or >
// after it, in that case it will change back to O. Note that
// changing back to O doesn't mean its signal disappears.
// Remember that the cell before it will be copying it too
// (be it an O or another > signal), thus signal > is flowing
// in the loops normally, unless something special happens.

else rot if (component=='>') {

    // One special condition is that it may be in the gap
    // between a parent and a child loop when the replication
    // cycle has just completed. By looking at the closing B
    // or the opposite direction of signal flows on both
    // sides, and the relevant neighbor positions which signify
    // a closing replication cycle, it can tell if it's in
    // that position. If yes, it will change to Detach D to
    // disconnect these two loops. This is very similar to the
    // last statement for the Block O above.

    if ((nw:component=='>,3' || nw:component=='O') &&
        (ne:component=='B' || ne:component=='>,1') &&
        no:component=='.' && ea:component && we:component)
        component='D';                      // discussed in §6.9

    // Otherwise, see if L is behind it. If so, it should copy
    // the L signal. One very interesting special case here is
    // that if the "growth" field is set, it will stimulate the
    // growth of signal sequence by one during arm extrusion
    // time. This is achieved by NOT copying the L signal and
    // stay as > for one more cycle. Since > signal is always
    // copied, this will actually insert one more > into the
    // sequence. Yes, the L signal will disappear in the next
    // epoch, but since the "growth" field is still set, that L
    // signal will be recovered in one more epoch. See the
    // rule below.
    // Another special case is when a 2x2 loop has just
    // completed generating a 3x3 loop and is about to extrude
    // its new arm. Since the signals L>> and EF is more than
    // four, the number of cells in a 2x2 loop, there must be
    // some way this L>> sequence can be regained after the
```

```
    // transient E and F signal that extruding a new arm. This
    // is achieved by utilizing the "growth" field. Yes, that
    // way, no matter what's the status of the "growth" field we
    // will always set it to on, so you can't get more >'s in
    // the signal sequence within a 2x2 loop.

    else if (we:component=='L')
        if (nw:component=='E' &&
                (growth || no:component=='>,3'))
            growth='+';                 // discussed in §6.12
        else
            component='L';              // discussed in §6.5

    // With the two special cases considered, all the rest is
    // just simple copying and doesn't need further
    // explanation.

    // copying the L signal, for four cell special case
    else if (no:component=='L')
        component='L';                  // discussed in §6.11

    // copying the > signal
    else if (we:component=='>')
        component='>';                  // discussed in §6.5

    // copying the > signal through the corner
    else if (we:component=='>,1')
        component='>';                  // discussed in §6.6

    // copying the > signal through the corner, in arm tip
    else if (no:component=='>,1')
        component='>';                  // discussed in §6.6

    // Now here is the special case companion rule, see
    // above. If the "growth" field is set on and L is not
    // following >, the > will change back to L again to
    // regain it.
    else if (growth && nw:component=='F') {
        component='L'; growth=0;        // discussed in §6.12
    }

    // otherwise > always changes to O
    else
        component='O';                  // discussed in §6.8
}

// **************** Birth B rules ****************
// Birth B is generated when a quiescent bound cell is
// pointed by a > signal. This is the way how loops grow.
// Functionally B is very similar to Block O in that it just
// copies signals through it. The reason we need the Birth B is
```

```
// that it is easier to decide if a replication arm is closing
// on itself by spotting the B at the periphery.

else if (component=='B') {

    // In the closing of 2x2 loops the B should change to L
    // immediately to preserve the signal sequence.

    rot if (we:component=='L' && no:component=='O')
        component='L';              // discussed in §6.11

    // Otherwise B usually change to the Corner C when seeing L
    else rot if (we:component=='L')
        component='C';              // discussed in §6.6

    // If > is behind B will just copy it. Note that this is
    // one of the two places where a monomer can kill a loop
    // since the > signal is not checked to be bound or not. So if
    // a monomer > signal is facing the Birth B it will
    // influence the normal operation of the loops and kill it.

    else rot if (we:component=='>')
        component='>';              // discussed in §6.5

    // B can't be in a line of bound cells unless it's a
    // collision.

    else rot if (no:component && so:component)
        special='#';                // discussed in §6.10

    // Finally B always changes to O. This occurs when
    // extruding a new arm.

    else
        component='O';              // discussed in §6.8
}

// **************** Signal L rules *****************
// Signal L gives the command of turning the growth direction
// 90 degree counterclockwise at the tip of the replicating arm.
// Signal L will also help generate a new replicating arm
// when closing with the current replication cycle.

else if (component=='L') {

    // L will change to extrude signal E if it sees the special
    // extrude mark on the "special" field. This extrude mark
    // itself was set when seeing Detach D, the disconnect
    // component, in its neighbor.

    if (special=='*')
```

```
        component='E';                  // discussed in §6.8

    // if > catch up with L, the signal sequence must have
    // some problems (possibly due to too many growth stimuli
    // +), and must be destroyed.

    else rot if (we:component=='>')
        special='#';                    // discussed in §6.10

    // copying the E after the L, unless it's already at the
    // corner (thus sw:component=='F', since F always follows E).

    else rot if (so:component=='E' && sw:component!='F'
            && (no:component=='.' || we:component=='.'))
        component='E';                  // discussed in §6.8

    // otherwise L always changes to O
    else component='O';                 // discussed in §6.5
}

// **************** Component . rules *****************
// . might not be called component at all since it's the
// quiescent state for the "component" field. It will be changed
// to a non-quiescent state if seeing either the > or E
// signal, or it may fall back to monomer state if none of its
// neighbor is a bounded non-quiescent cell.
// Note that a monomer can kill a loop by inhibiting the
// generation of the arm when standing beside the potential
// "O" position. This is the second way a monomer can kill a loop.

else if (component=='.') {

    // When seeing signal > pointed to itself, it will change
    // to the Birth B.

    rot if (we:bound && we:component=='>' &&
            (nw:component=='.' || nw:component=='>,1' ||
             nw:component=='L'))
        component='B';                  // discussed in §6.5

    // Or it will extrude the arm at the corner when seeing E.
    // If north is occupied (no matter it is a bound cell or
    // monomer), the extrusion will fail.

    else rot if (so:special==0 && so:component=='E' &&
                    se:component=='.' && no:component=='.')
        component='O';                  // discussed in §6.8

    // Otherwise, see if no bound neighbor is nearby. If so,
    // change back to unbound mode.
```

```
    else {                              // discussed in §6.13
        count=0;
        over each other y:
            if (y:bound && y:component && y:special==0)
                count++;
        if (count==0) bound=0;
    }
}

// **************** Corner C rules *****************
// C is the corner component generated by L signal at the tip of
// a replicating arm. It will turn an incoming > signal 90
// degree counterclockwise, thus changing the direction of
// replication.

else if (component=='C') {

    // First check for fail situation. E or F shouldn't meet C in
    // anyway. If they meet C that means something must have gone
    // wrong so the current loop must be destroyed.

    rot if (no:component=='E' || no:component=='F')
        special='#';                    // discussed in §6.10

    // turn > at the corner, four cell case considered, too.

    else rot if (we:component=='>' || we:component=='>,1')
        component='>,3';                // discussed in §6.6,6.11
}

// **************** Detach D rules *****************
// Detach D is the blocking component. It is generated when a
// replicating arm has fallen back to itself, thus making a new
// child loop. This component can block the dissolve flag # in the
// "special" field to preserve either the child or parent loop
// from the death of each other. This component is erased when
// the two loops have seen it and have generated the arm
// extruding flags.

else if (component=='D') {

    // removing of the blocking cell when seeing "special" field set
    // in the neighbor.

    rot if (ea:special) component='.';  // discussed in §6.9

}

// **************** Signal E rules *****************
// E always followed by F, so it always changes to F. It is
// part of the two signals EF extruding sequence which extrudes
```

```
        // a new arm.

        else if (component=='E') {

            component='F';                    // discussed in §6.8

            // always reset the arm extrude special flag so only one EF
            // sequence is generated.

            special='.';                      // discussed in §6.8
        }

        // **************** Signal F rules *****************
        // Signal F is part of the EF sequence to extrude a new arm.
        // When it sees no new arm generated by its predecessor E, it
        // will set the dissolve flag since extrusion fails.

        else if (component=='F') {

            // extrude testing for 2x2 loop special case
            rot if (no:component=='>,2' && ea:component)

                // copying the > signal or set the dissolve mode
                if (ea:component=='E' || so:component=='O')
                    component='>,1';          // discussed in §6.11
                else
                    special='#';              // discussed in §6.10

            // if arm extruding fails, self-destruction will begin
            else rot if (no:component=='O' &&
                        (ea:component=='O' || ea:component=='L') &&
                        so:component=='.' && we:component=='.')
                special='#';                  // discussed in §6.10

            // otherwise F always change to O
            else component='O';               // discussed in §6.8
        }

        // A special rule for the arm extrude flag. Whenever a cell
        // sees the Detach D and it itself is a corner cell (note: not the
        // Corner C component), the arm extrude flag * will
        // always be set, no matter what component the cell is.

        rot if (component && special=='.' &&
              (ea:component=='D' && no:component && we:component ||
               we:component=='D' && no:component && ea:component))
            special='*';                      // discussed in §6.8
    }
}       // closure of the "} else { ..." statement
```

Chapter 7

Solving SAT Problems with Self-Replicating Loops

Recently there have been suggestions that recombinant DNA techniques can be used to solve some NP-complete problems [Lipton, 1995; Adleman, 1994]. The basic idea is to use different DNA sequences to represent different trial solutions, and to use the separation methods from molecular biology to isolate those DNA sequences which are desired or correct solutions. Since all of the separation operations are done in parallel to all molecules, it is as if we are testing all possible solutions at once. The answer to a problem, if any, can thus be found in a linear number of separation steps. Since the number of DNA molecules is very large even in a small test tube, the problem space this method can explore is astronomically big.

Solving NP-complete problems on a traditional sequential computer normally requires exponential time. The Satisfiability Problem (SAT problem) is one classic example of an NP-complete problem [Hopcroft & Ullman, 1979]. Given a boolean predicate like

$$\mathcal{P} = (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$$

the SAT problem asks "what assignment of boolean values to the binary variables x_1, x_2 and x_3 can satisfy this predicate", i.e., can make this predicate evaluate to True? The predicate $\mathcal{P}$ here is in Conjunctive Normal Form (CNF), where each part of the predicate surrounded by a pair of parentheses is called a *clause*. A SAT problem is usually marked as m-SAT problem if there are m boolean variables in a clause of its predicate. Therefore, the above example is a 2-SAT problem[1].

The SAT problem is NP-complete since, in general, we do not know how to make the assignment to let a predicate be true. Intuitively, we can only test all possible assignments seeking to find if there is any one which will satisfy the predicate. Thus, it takes exponential time to explore the solution space since if there are n boolean variables, there are 2^n possible assignments to those variables. Although there are algorithms which can do better than 2^n, they still require exponential time (e.g. $2^{\sqrt{n}}$).

The so called *DNA computer* technique mentioned above separates DNA sequences representing possible SAT assignments all at once based on the clauses of a given SAT predicate. During each separation cycle all current DNA sequences were tested against one clause of the predicate, and unsatisfying sequences were removed. Therefore, in a limited number of steps which corresponds

[1]Note that without loss of generality, 2-SAT problems are chosen as examples in this chapter due to their simplicity. Although 2-SAT problems can actually be solved in polynomial time and are not truly NP-complete, the method presented in this chapter does not take advantage of that special property and therefore is equally applicable to 3-SAT or higher problems.

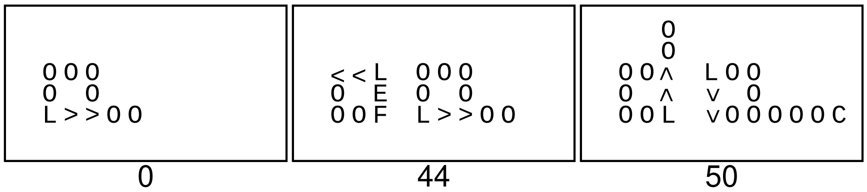


Figure 7.1: A typical self-replicating loop. Starting from epoch 0, the left loop makes a copy of itself in 44 steps, and returns itself to the original state in epoch 50 (rotated 90 degrees counterclockwise). Both the old loop and the new loop are continuing to replicate more loops at epoch 50. It is obvious that many cells are in the block state O and can be used for other purposes other than just being the building blocks of the loop.

to the length of the SAT predicate, the satisfying assignment, if any, can be found in the final test tube.

Although the DNA computer technique is still in its early stages, it does suggest that we may be able to use self-replicating loops to solve the SAT problem, too. Self-replicating loops in a cellular automata space are autonomous entities that function independently and in parallel. The cellular automata model they are based on is without bound theoretically. If we can grow many self-replicating loops, each carrying a different trial boolean assignment for a SAT problem, and we can also somehow impose selection upon these loops so that only those carrying satisfying variable assignments will survive, we can achieve the same phenomenon that a DNA computer can. That is the method of using self-replicating loops to solve the SAT problem described in this chapter.

A typical self-replicating loop like those used in the previous chapter is shown in Figure 7.1. It uses only a quarter of its cells to encode its replication steps, leaving about 3/4 of its cells unused in the Block O state. These unused cells are used to encode bit sequences representing possible satisfying boolean assignments for a SAT problem. A selection mechanism is built into the cellular automata rule set so that only those loops which carry satisfying assignments to a SAT predicate will survive in the cellular automata space through time.

The basic idea is this: we just let the self-replicating loops using their extra cells carry the binary sequences and grow in the cellular automata space, making generations of new loops. The replication process is designed in a way such that each time a new loop is born, its binary bit sequence will be explored for one bit and that this bit will be different when in the child and in the parent loop. Initially, there will be only one loop in the cellular automata space carrying a totally unexplored bit sequence; in the end there will be many loops in the cellular automata space each carrying a different fully explored bit sequence. This is called the *enumeration process*. With this enumeration process, we can get all possible assignments for a given SAT predicate if there is no congestion of loops.

During the course of enumeration, monitoring mechanisms are also built into the cellular automata space to look for any loop which carries unsatisfying assignments. This will be the natural enemy of loops such that those loops which carry unsatisfying binary sequences will not survive. This is called the *selection process*. Since the replication of loops is proceeding in exponential speed (1, 2, 4, 8, ...), we can literally explore all possible SAT solutions in linear time, the same as "the DNA computer", and leave only those loops carrying satisfying boolean assignments in the cellular automata at the end. Put another way, we use self-replication to bring together computing power that increases as the size of the cellular automata space increases.

In the following section, several examples of solving the SAT problem using self-replicating loops are presented first. This is followed by a description of the cellular automata rule set in Section 7.2 and its subsections. Analysis of the rule set and some running results are given in the next two sections. It is shown in this chapter that in addition to self-replication, self-replicating loops can be made to solve problems, too. It is also shown that complex selection mechanisms among self-replicating loops can be incorporated into a cellular automata space. Detailed discussions about these and other contributions of this work are given in the final discussion section.

7.1 Examples

Figure 7.2(a) shows the enumeration result for a self-replicating loop carrying 3 binary bits. In the original loop, unexplored bits are represented by the symbol A. We can see that these A's are replacing some of the block O's in the old self-replicating loop (see Figure 7.1 for a comparison). The original growth signal '>', which has weak rotational symmetry, is also replaced by the symbol G (literally for **G**rowth) in the new loop. The new symbol G is strong rotational symmetry. The reason for this change is because the original '>' symbol for growth no longer needs to be weak rotational symmetry and has been used for other purposes now, which will become clear when we look at the cellular automata rule set which implements all of this in the next section. Explored bits are represented by either digit 0 or 1 in the loops. The binary sequence a loop carries is read off starting right after the L symbol, therefore, the lower left loop in the bottom left figure carries the sequence 001 and topmost loop carries 011. The block O has also been changed to lowercase 'o' typographically in order to avoid confusion with the digit zero. Note that in Figure 7.2(a), only the enumeration process is at work.

Without the selection process, in three generations we get all eight possible boolean assignments for a three variable SAT predicate carried by eight loops in the cellular automata space. Loops will stop growing once they have explored all of their bits (no more A's within themselves). Since the exploration of bits is done one bit at a time for one generation, and explored bits are preserved in the loops and inherited from the parents unmodified, we can be sure that all possible boolean assignments will be found with the enumeration process. Remember that for each exploration step we get a different value in the parent and child loops. For example, starting with the loop carrying totally unexplored binary sequence AAA, in the first generation we will get 0AA in the parent loop and 1AA in the (first) child loop. In the second generation we will get two more new loops carrying 01A and 11A; the two parents (for this 2nd generation) now carry 00A and 10A. In the third and final generation, we get four more new loops 011, 111, 001 and 101. The four parents now carry 010, 110, 000 and 100. Together, we get all eight values for a 3 bit binary sequence.

To remove those loops which do not satisfy a SAT predicate, we spread "*monitors*" throughout the cellular automata space. Each monitor tests for a particular clause of the SAT predicate. If the condition a monitor is looking for is verified, it will "destroy" the loop on top of it. For the predicate example $\mathcal{P}$ we gave before, three classes of monitors, each testing for one of the following conditions, are planted in the cellular automata space:

$$x_1 \wedge \neg x_3$$

$$\neg x_1 \wedge x_2$$

$$\neg x_2 \wedge x_3$$

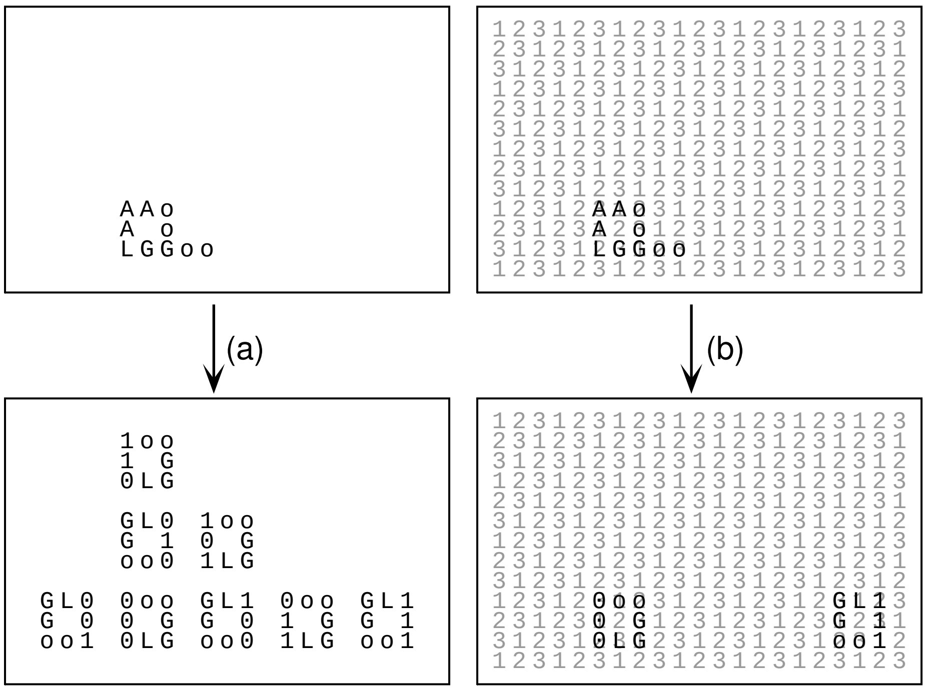


Figure 7.2: The enumeration and selection of satisfying boolean assignments by self-replicating loops. (a) one 3 by 3 self-replicating loop can carry 3 binary bits (location marked by AAA) and enumerate all eight possible boolean assignments in three generations. (b) When "monitors" are introduced into the cellular automata space, the selection process kicks in. In three generations only those loops carrying satisfying assignments to the boolean predicate $\mathcal{P}$ will be immune from elimination by monitors and will survive in the cellular automata space. Monitors are represented by light gray digits in part (b). There are three different kinds of monitors, each standing for one clause of the original predicate $\mathcal{P}$.

These conditions are based upon the clauses of the specific predicate $\mathcal{P}$, with their expression negated, because if any one clause of predicate $\mathcal{P}$ is **not** satisfied, the whole predicate will not be satisfied, thus the negation. A monitor will destroy the loop if its corresponding clause is found to be unsatisfied by the binary sequence carried by the loop on top of it. This detection process is done in linear time since essentially each monitor is just a finite automata machine and the binary sequence passing through it can be seen as the regular expression for recognition. With enough monitors in the cellular automata space, they can effectively remove all unsatisfying solutions. Remember that all clauses of a SAT predicate must be satisfied for the whole predicate to be satisfied and failing to satisfy **any one** clause can prove to be fatal. Note that the size of the self-replicating loop must always be big enough to cover different monitors at least once, otherwise some clauses will not be checked against the loop.

Figure 7.2(b) displays the result of replication starting from the same original loop as used in Figure 7.2(a) but with the selection process turned on this time. Monitors are planted throughout the cellular automata space according to the three conditions set above. They are shown in light gray color in the background of the figure. Since it does not hurt to have more monitors than less in the cellular automata space, it was decided to fill all cellular automata cells with monitors. We can see that there are three types of monitors designated by symbol 1, 2 and 3 because there are three conditions to test.

The symbol of a monitor tells it which condition it should test. Like all the other cellular automata activities, the testing rules for each condition are defined in the cellular automata rule set. Basically, a monitor keeps track of the boolean bit sequence flowing through its position, sets flags and destroys loops if the cellular automata rule condition is found. The monitor rule set is working independently of the self-replication and enumeration rule set for the self-replicating loops. But if an unsatisfying loop is found, the monitor will interfere with the self-replicating rule set so that the unfit loop can be destroyed.

In three generations we have only two loops left in the cellular automata space instead of the original eight. These two loops carry exactly the only two satisfying boolean assignments for the original SAT predicate $\mathcal{P}$, which are 000 and 111.

We display more intermediate steps for the enumeration and selection process starting with the 3 by 3 loop carrying three unexplored binary bits in Figure 7.3. Since monitors do not move once planted, for the sake of clarity they are not shown in this figure. But they are still there, doing their jobs, exactly as shown in part (b) of Figure 7.2.

It can be seen in this figure that part of the family tree is killed without even being generated, i.e., the topmost loop in Figure 7.2(a) has never been generated because its parent loop (the one below it in the same figure), which still carries unexplored binary bits, is killed early in the replication cycle. Since it has been found (by the monitors) that the partially explored bits 01A in this parent loop do not satisfy one of the clauses, there is no need to explore further since all its descendents will carry the same binary bits.

Another example solving a six variable SAT problem using 4 by 4 loops is shown in Figure 7.4. This is for satisfying the six variable predicate

$$\mathcal{Q} = (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_6)$$

Self-replicating loops are able to find the only two satisfying boolean assignments out of a total of 64 possible assignments in 394 iteration steps, or about six generations. One replicating generation for a 4 by 4 loop takes 65 cellular automata iteration steps, which involves four cycles of the replication signal sequence in the parent loop to build the child loop plus one more cycle to extrude the new arm. Detailed analysis of replication steps needed for loops of different sizes is provided in Section 7.3.

We can see from this and the three variable SAT example above that generally it takes n generations to solve an n variable SAT problem. It is understandable why this is the case since for each generation one bit is explored and there are n bits in an n variable SAT predicate. Again, in this example the monitors are not shown in the figure for the sake of clarity but they are everywhere in the cellular automata space trying to find their victims. They are like a classical automata machine recognizing computer-theoretical regular expressions represented by the boolean sequences carried by the self-replicating loops. They use different flags to represent different stages of the recognition process and will destroy the loop on top of them if the recognition is completed, i.e., an unsatisfying loop has been found. There are six different kinds of monitors in this example since there are six clauses in the predicate $\mathcal{Q}$. Starting from the totally unexplored bit sequence AAAAAA, the only two satisfying boolean assignments of this particular 6 variable SAT predicate is 000100 and 111100 (remember bit sequences are read off right after the signal L, and correspond to boolean assignments to the variables of a predicate).

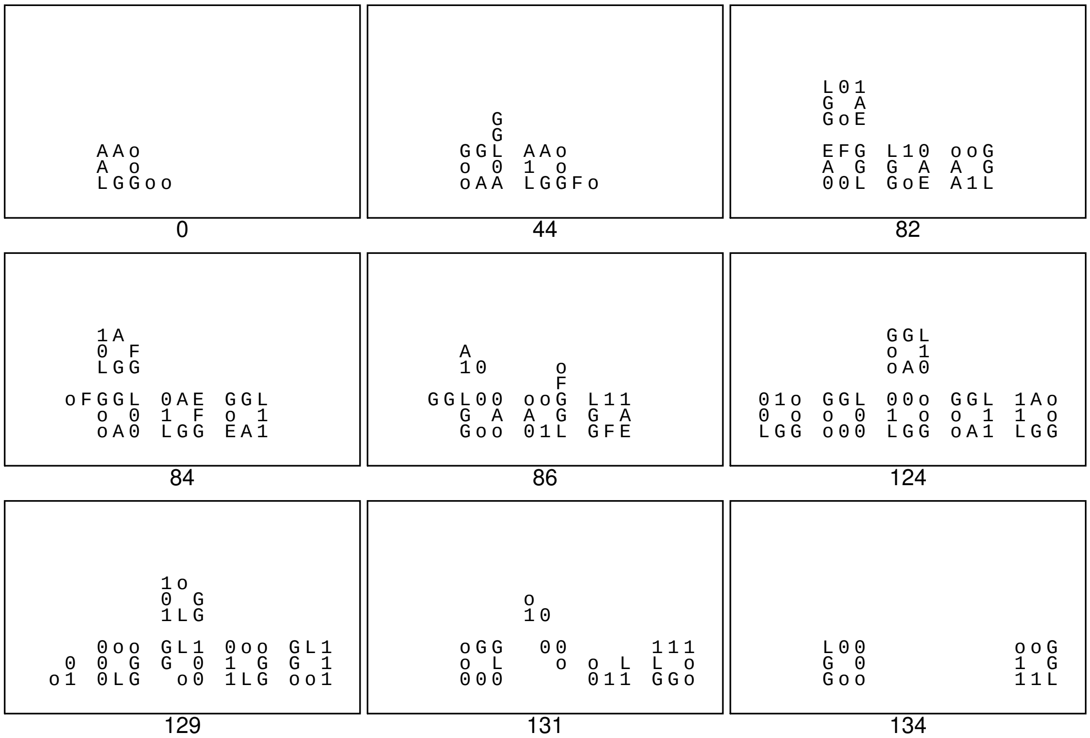


Figure 7.3: Progressive stages in the enumeration and selection of 3 by 3 loops. In epoch 0 the initial loop is placed on the cellular automata space, which carries unexplored binary bits represented as AAA. Monitors checking for the three clauses of the predicate $\mathcal{P}$ are also planted in the cellular automata space just like in Figure 7.2(b), but are not shown in this figure so we can easily read the data in the loops. In epoch 44 the first replication cycle has completed and we get two loops in the cellular automata space. The first binary bit has been expanded into 0 and 1 in the two loops by the enumeration process. In epoch 82 the second replication cycle has completed and we have four loops in the cellular automata space. But starting in epoch 84 the top loop is being destroyed and erased (look at the missing corner cell of the loop in epoch 84). Apparently its bit sequence does not satisfy one of the clauses of predicate $\mathcal{P}$ ('01A' does not satisfy $x_1 \vee \neg x_2$, the second clause), therefore it is being erased by the monitor under its upper-right corner. In epoch 86 the erasing process continues while the other loops start their next replication cycle. In epoch 124 the third (also the last) replication stage is completed and we have six loops in the cellular automata space, but four of them do not survive the monitors very long since they do not satisfy some clauses and are erased (epochs 129 and 131). Finally, we are left with two satisfying assignments 000 and 111 at epoch 134.

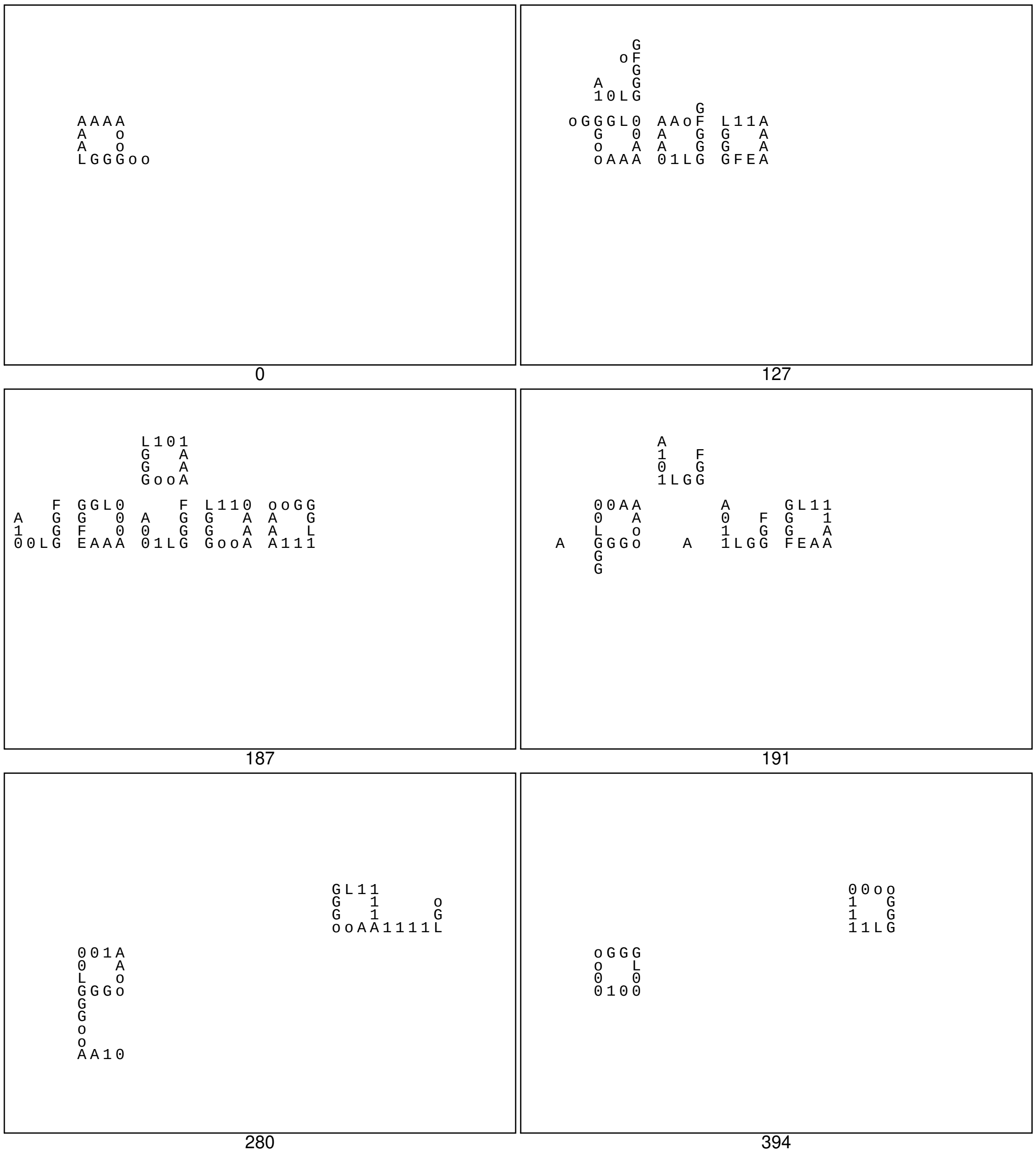
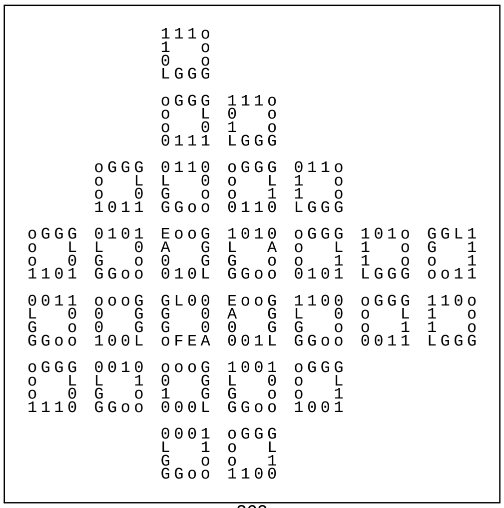


Figure 7.4: Solving a 6 variable SAT problem using 4 by 4 loops. The predicate to satisfy is $\mathcal{P} = (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_6)$ The original loop carrying six unexplored bits AAAAAA is placed in the cellular automata space at epoch 0. New loops are generated and killed by monitors (not shown in the figure for clarity) if they carry unsatisfying bit sequences. There are only two satisfying boolean assignments out of a total of 64 possible assignments for the predicate $\mathcal{Q}$, 000100 and 111100, which are found in epoch 394, about six generations of the 4 by 4 loop.



368

Figure 7.5: Crowding of self-replicating loops while finding satisfying boolean assignments. Self-replicating loops will be crowded together if the speed of removing unsatisfying loops cannot keep up with the speed of self-replication. In this example, an extreme case is shown where all loops are immune to monitors since the boolean assignments they carry can all satisfy the predicate $\mathcal{R}$, and therefore none of the loops are removable. In five generations some loops in the congested region run out of space to grow and cannot find all possible boolean assignments since they still have unexplored bit A's but have no space to expand into.

Self-replicating loops may not always find all satisfying boolean assignments for a given predicate, but they will find at least one if there are some satisfying assignments. The DNA computer cannot find all satisfying boolean assignments for a given predicate either, since its recombinant DNA separation method is not 100% precise. Both the self-replicating loop and the DNA computer can tell us if a boolean predicate is satisfiable at the least but they may not find all satisfying cases. The SAT problem asks if a predicate is satisfiable by at least **one** boolean assignment; finding all satisfying assignments is usually not needed, especially when there are many satisfying assignments.

For example, for the contrived six variable predicate

$$\mathcal{R} = (x_1 \vee \neg x_1) \wedge (x_2 \vee \neg x_2) \wedge (x_3 \vee \neg x_3) \wedge (x_4 \vee \neg x_4) \wedge (x_5 \vee \neg x_5) \wedge (x_6 \vee \neg x_6)$$

which is satisfiable by any boolean assignment, self-replicating loops will never die once born since all of them are immune to the six (non-destructive) monitors representing the six trivial clauses. Since loops will not be erased from the cellular automata space, they will crowd each other in the cellular automata space in a short time. Therefore, they cannot explore all possible boolean assignments, as shown in Figure 7.5. Nonetheless, they do tell us that this predicate $\mathcal{R}$ is satisfiable in numerous ways and thus answer the SAT problem: is the predicate satisfiable?

A 100% satisfiability in the case of predicate $\mathcal{R}$ may be a little bit extreme, since in that case we do not need to "solve" the trivial SAT problem at all. But a high satisfiability of a given boolean predicate does slow down the process of finding its satisfying assignments. Self-replicating loops can take a much longer time to search for possible boolean assignments, even if they can find all satisfying assignments eventually. Loops in the congested region cannot grow until new space are provided around them, which could take a very long time. For example, if 50% of all possible boolean assignments of a 6 variable SAT problem are satisfying, the self-replicating loop takes 787 iteration steps, or about 12 generations, to reach all these satisfying assignments as shown in Figure 7.6(a). At 25% satisfying assignments, the cellular automata space is less congested, but the self-replicating loop still takes 695 iteration steps, or about 11 generations to reach all satisfying assignments, as shown in Figure 7.6(b). The reason is that the cellular automata space for the 25% case is still as congested as in the 50% case until very late in the simulation, when most of the unsatisfying loops start being removed.

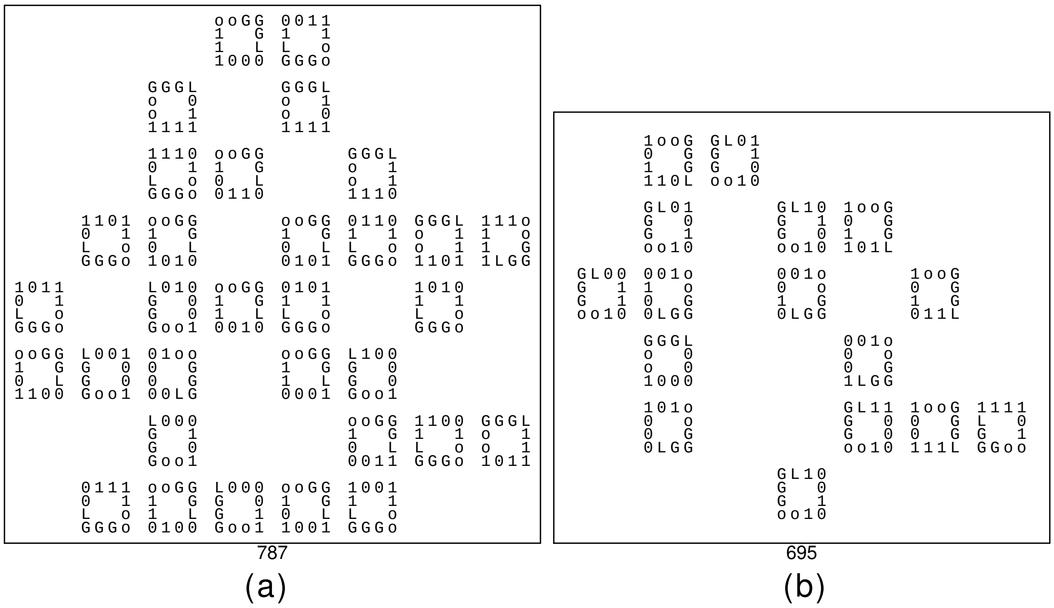


Figure 7.6: Slow exploration of SAT assignments. (a) If 50% of all possible boolean assignments for a 6 variable SAT problem are satisfying, it takes self-replicating loops 787 iterations to reach all these assignments. (b) It still takes 695 iterations for self-replicating loops to reach all 25% satisfying boolean assignments for another 6 variable SAT problem.

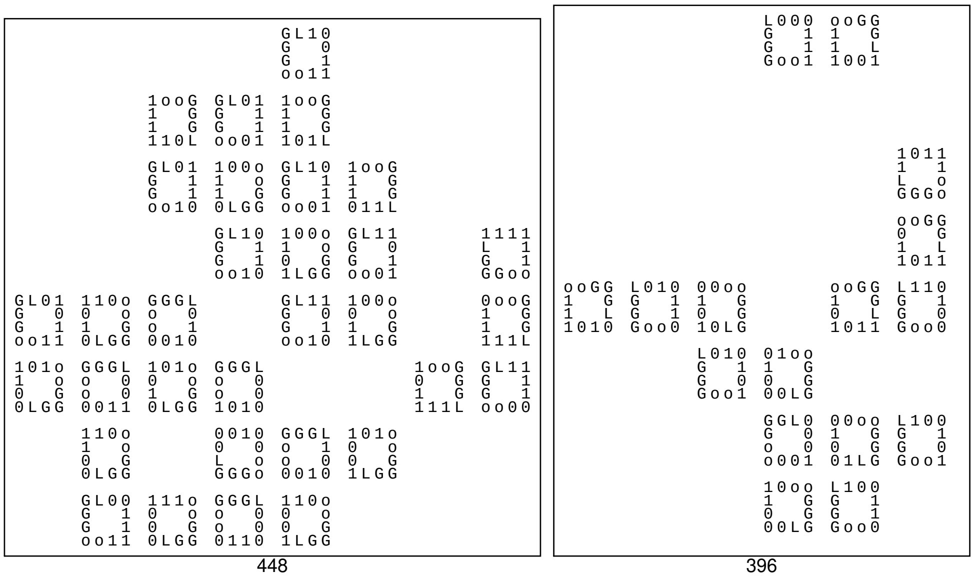
The time for removing of unsatisfying loops plays a major role in the efficiency of finding satisfying boolean assignments for a given SAT problem. Note that to determine whether a predicate is satisfiable or not is generally easier than finding **all** satisfying boolean assignments, since we need to find **only one** satisfying boolean assignment to answer that question.

For example, at the same satisfying assignment ratio as in Figure 7.6(a) and (b), if some unsatisfying loops can be decided earlier by the monitors during the replication process (at generations 3 and 4), it takes only about 7 generations to reach all satisfying assignments shown in Figure 7.7(a) for 50% satisfiability, and it takes only about 6 generations to reach all satisfying assignments shown in Figure 7.7(b) for 25% satisfiability. In the latter case we have reached the theoretical minimum (fastest) number of steps to find the satisfying assignments. The fact that SAT problem characteristics (like the time for removing unsatisfying loops, etc.) can influence the efficiency of finding satisfying boolean assignments by self-replicating loops will be discussed further in Section 7.4.

Note that all examples presented in this chapter are for solving 2-SAT problems, i.e., their clauses have two boolean variables. This is for the purpose of simplicity. The cellular automata rule set can be easily modified to solve SAT problems with longer clauses. There is no inherent limitation on the clause length for using this method to solve a SAT problem.

7.2 A behavioral look at the cellular automata rules

We start to look at how self-replicating loops can be made to solve the SAT problem in this section. Since the structure and function of the cellular automata rule set for solving SAT problems is very similar to the emergent self-replicating rule set, only a behavioral look at the working of the



(a) (b)

Figure 7.7: Fast exploration of SAT assignments. (a) self-replicating loops take 448 iterations to find all satisfying boolean assignments for a 6 variable SAT problem with 50% satisfying ratio if trimming of unsatisfying loops occurred earlier at generation 3 and 4 instead of at the last (6th) generation. (b) Similar results for a 6 variable SAT problem with 25% satisfying boolean assignments. It takes self-replicating loops only 396 iterations to find them all, which is approaching the theoretical optimum speed of this method.

cellular automata rule set is provided. We leave the detail rule set listing at the end of this chapter for reference.

Again, a split of the cellular automata cell into fields is used to simplify rule set programming. First, let us look at the data fields used by the rule set. A brief description is given of each data field and its state values. Their usage will become clear when we see the inner working of the cellular automata rule set in the following subsections. The SAT rule set uses the Moore neighborhood, which is the same as the emergent self-replicating rule set of the last chapter.

7.2.1 Fields

The cellular automata rule set to solve the SAT problem is based on six functionally divided fields. A graphical depiction of the data fields used in the SAT rule set is shown in Figure 7.8. Valid states for each field are also listed.

The fields are:

1. code(4 bits): This is the fundamental field which carries instructions for self-replicating control and also the SAT bit assignment sequences. Valid values are the following:

   . This is the quiescent state: it means nothing is there. Normally when displaying the cellular automata space it is replaced by white spaces. The '.' symbol is only used in the cellular automata rules to represent the quiescent state.

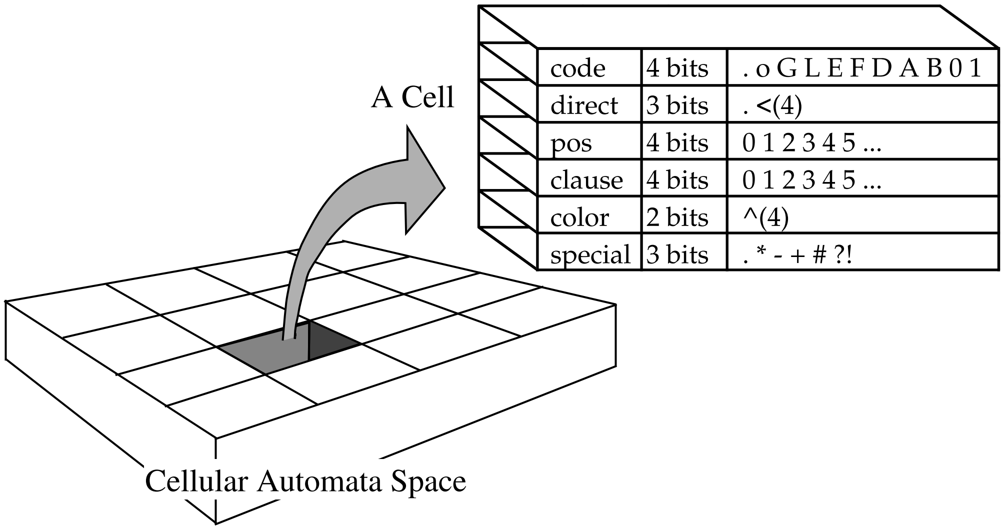


Figure 7.8: The data fields used in solving the SAT problem. Listed are the bit depths and valid state symbols in each field. Weak rotational symmetry symbols are denoted by "(4)" after them, indicating that they represent four states. Quiescent states are represented by a period '.' in the rule set, but are not displayed when showing the cellular automata space configuration.

o This is the building block of the loop. It serves as a place holder to keep the integrity of the loop and conceptually allows other codes to pass through it. It was represented by the capital letter 'O' in the rule set we discussed in the previous chapter. It is now changed to lowercase 'o' to avoid confusion with the number '0', which is also a valid state of this data field.

G This is the extrude signal, which directs the expansion of a cellular automata signal pathway into the quiescent space. It was represented by the symbol '>' in previous self-replicating rule sets that had weak rotational symmetry. It is now changed to the letter 'G' and to have strong rotational symmetry (i.e., represented by only one state value). With the help of the direc field to provide directional discriminations now (see below), weak rotational symmetry is no longer needed for this signal. The introduction of a dedicated direc field to provide directional discrimination is necessary because there is no specific signal order in the loop anymore and it also simplifies rule set design.

L This is the turning signal. It changes the expansion direction of a signal pathway by 90 degrees counterclockwise, as before.

E F This is the pair of signal sequences to direct the breaking out of a new arm for replication.

D This is the detach component which separates the parent and child loops during the replication process.

Those above are signals for replication control. The following four more signals have no effect on the replication process but are added for enumerating and representing SAT binary bit assignments:

A An unexplored SAT binary bit. When explored during a replication cycle it will always become 0 in the parent loop and 1 in the child loop.

B Also an unexplored SAT binary bit. It is seldom used in our examples of this chapter. When explored during a replication cycle, it will always become 1 in the parent loop and 0 in the child loop. Its function is a direct opposite of signal A. When to use it will be discussed in the section about efficiency (Section 7.4).

0 The explored SAT binary bit 0. It will not change further except when it is carried around in a self-replicating loop. It represents an assignment of boolean value zero to a corresponding boolean variable of the SAT predicate.

1 The explored SAT binary bit 1. It will remain in this state except being carried around in a self-replicating loop. It represents an assignment of boolean value one to a corresponding boolean variable of the SAT predicate.

We actually need only signal A or B to represent unexplored SAT bits, not both. But having both of them has the advantage of changing the exploration pattern and may help the efficiency issue (see Section 7.4).

2. direc(3 bits): This is a new field which defines the signal flow pathway. One different design feature of the SAT rule set when compared to the self-replicating rule set in the last chapter is that the signal flow direction is no longer implicitly defined by the signals themselves. For example, where "signal 'L' always follows signal '>"', when we see a signal sequence 'L>', we know it is going to the right, and when we see a signal sequence '<L', we know it is going to the left. There are definite patterns of signal sequences in the loop (always a certain number of '>' signals followed by the signal 'L'), and the cellular automata rule set can take advantage of that knowledge and determine the flowing direction unambiguously.

   But in the SAT rule set a loop can now carry an arbitrary SAT bit sequence and there is no easy way to tell which way the signal sequence 010101 or 001001 will flow, for example. Although one might find some other solution, such as making all bit states have weak rotational symmetry to gain a sense of orientation for the SAT sequence, these methods are at best awkward and would waste useful cellular automata states. A more elegant solution is to use a "direc" data field to explicitly point out the signal pathway. The direction the direc data field points will be the direction **any** signal sequence on top of it will flow toward. We can imagine the direc field as a conveyor belt in a factory assembly line which carries everything with it. What we are doing here is to factor the weak symmetry information out into a separate field direc that **all** symbols can reference. Valid state values for direc are the quiescent state plus the four weak rotational state values of the signal >, which always point at where signals flow.

3. pos(4 bits): This is the data field which records what corresponding boolean variable in a SAT predicate is represented by a binary bit in a SAT sequence carried in a loop. We need this field in order to tell the order of bits and (for the monitor) to check for unsatisfying bit sequences. Values in this field are attached to bits in the code field and flow with them. Valid values for this field are numbers 0, 1, 2, ... etc., each standing for boolean variable one, boolean variable two, ... etc. in the SAT predicate.

4. clause(4 bits): This is the "monitor" field which encodes what particular SAT predicate clause the monitor in a cell should be checking for. A cell with a particular non-quiescent clause value will look for any bit sequence passing atop it which represents an unsatisfying boolean

assignments with regard to the clause it is checking for. Cellular automata rules are defined to control this checking process. If the monitor can find such an unsatisfying sequence it will destroy the loop by setting a special flag and kill the loop carrying that sequence on top of it. Valid values this field are numbers 0, 1, 2, ..., etc. which stand for clause one, clause two, etc. in the SAT predicate.

5. color(2 bits): This is a tag field to mark the expansion direction for each loop. It is used to detect the collision of the expansion arms of two self-replicating loops. Since two loops expanding at the same direction (thus carrying the same color) can never collide, if a collision of loops occurs it can be detected by different color's of the arms. Valid values for this field are just the four weak rotational values of the signal ˆ. This field is different to the direc field in that its values are always the same within a loop, but such is not the case for the direc field, which can have four different values in the four different sides of a loop.

6. special(3 bits): This is the field which denotes occasional special situations in the cellular automata space. There are the following special situations:

   . No special situation.

   * A branching signal sequence (EF) will be generated.

   \- Unexplored binary bit A will change to bit 1 and unexplored binary bit B will change to bit 0 when seeing this special flag.

   \+ Unexplored binary bit A will change to bit 0 and unexplored binary bit B will change to bit 1 when seeing this special flag.

   \# The destruction flag. Any loop cell touching this special flag will be erased in the next iteration. It is set when an unsatisfying loop is found by a monitor. It will kill an unsatisfying loop in a very short time.

   ? A partial unsatisfying flag. When the first boolean variable of a clause is not satisfied this flag will be set by the checking monitor. It signals another state of the monitor automata. The monitor in a cell with this special flag on will be checking against the second variable of the clause it is assigned to check for (by the clause field), instead of the first variable as it normally will check. If the second variable of the clause is also found to be not satisfied by the current binary sequence on top of it, according to the cellular automata rules, the monitor will set the destruction flag #, to destroy the loop.

   ! The collision flag. It is set when the expanding arm of a loop detects that it is colliding with other loops or their arms. Any active loop part touching this special flag will be erased in the next iteration unless it is a branching point where more than one signal pathway is branching out from the same cell. It will effectively remove an old replicating arm of a loop in a very short time and set to start a new arm at another direction.

At the beginning of a simulation, one loop is put into the cellular automata space. The code field of the cellular automata space is initialized to reflect a normal self-replicating loop configuration. Some unexplored A bits to represent the boolean assignment sequence is placed in the loop body, too. The direc field is initialized to control the signal flow direction for the loop. The pos cells under those A bits are set to represent their associated variable numbers of the SAT predicate. An arbitrary cell in the clause field is set to a nonzero value, usually one, to represent a *seed* monitor

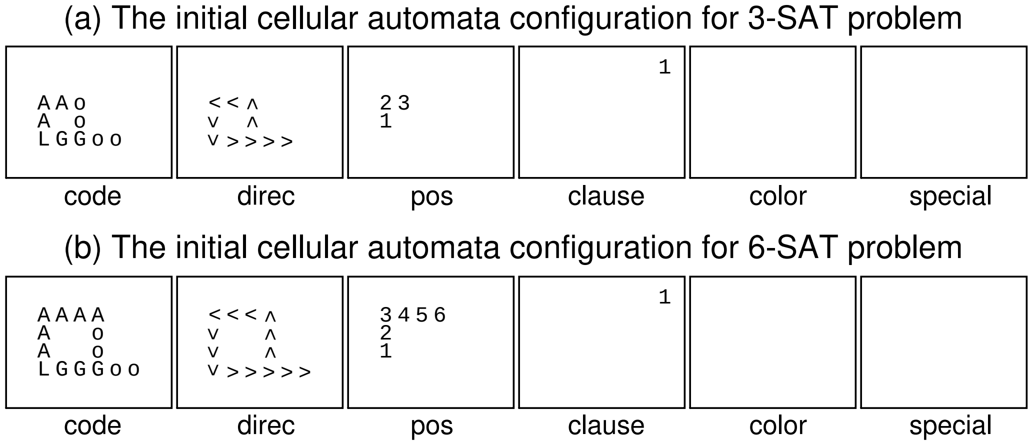


Figure 7.9: Examples of the initial cellular automata configuration. (a) The initial cellular automata fields for solving the 3 variable SAT problem as given in Section 7.1. (b) The initial cellular automata fields for solving the 6 variable SAT problem given in Section 7.1, too.

in the clause field. After the simulation begins, the cellular automata rule set will evenly spread different monitor values over the whole clause field starting from that seed monitor. Without this seed monitor the spreading of monitors will not be carried out and the clause field will remain quiescent. This seed monitor can be seen as the switch to turn on the monitor checking mechanism of the cellular automata rule set. The color field and the special field is initialized to the quiescent state zero. Two typical examples of the initial cellular automata space configuration for solving the 3 variable SAT and the 6 variable SAT problem mentioned Section refse:examples are given in Figure 7.9, where all initial field contents are displayed in detail.

This completes our discussion about fields. Although there are six data fields, only the code field is displayed in the examples of previous sections (except Figure 7.2) when the content of the cellular automata space was presented. This is because the major functionality of the SAT rule set is controlled by the code field. In the following subsections, our discussion will still be oriented around the code field. When no explicit context is given, any symbol described will be associated with the code field. Some other fields will also be displayed alongside the code field when we consider the inner working of the SAT rule set and when it is necessary to explain the functions of the other fields together with the code field.

The SAT cellular automata rule set has two independent functionalities which have been introduced before: the enumeration process by self-replication loops to generate all possible boolean assignments to a SAT predicate, and the selection process by monitors to select only those loops which carry satisfying assignments to survive at the end. The enumeration process is bundled within a normal self-replication process of the self-replicating loop by providing more rules to change an unexplored bit A in the original loop into bit 0 in the parent and bit 1 in the child loop after the replication. The selection process is a new mechanism in this rule set which is basically modelled after the usual regular expression checking automata machine, which has its own states to represent different stages of the checking process. The binary sequence carried by a loop is the string this automata is checking against. We will first look at the enumeration process in the following several subsections. The monitor selection process will be discussed in the last subsection.

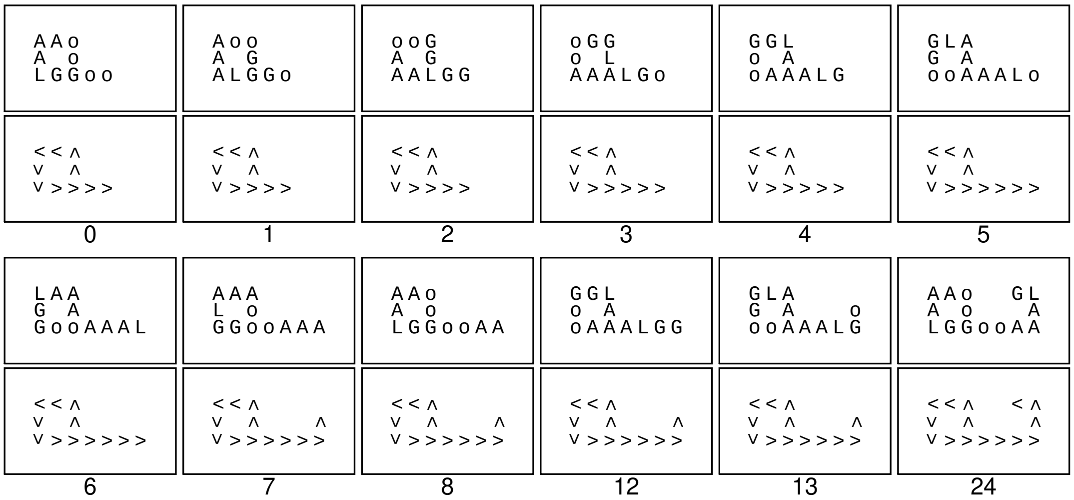


Figure 7.10: The flow, extending and turning of signal sequence. The code field (top) is displayed with its associated direc field (bottom) for a number of epochs. The direc field defines the signal flow direction. Note that these two fields actually overlay each other at each cell; they are displayed separately only for ease of reading.

7.2.2 Signal flow, extending and turning

The signal flow pattern in the SAT rule set is very similar to that of the emergent self-replicating rule set. The only difference is that the direction of signal flow is now explicitly defined by the direc data field. Look at the code field and its associated direc field content as shown in Figure 7.10. Each cell with an active direc field will copy the signal value from the neighbor sitting in the back of its direc field pointer during each iteration. Therefore, the signal flow direction is the exact direction to which the direc field points.

When a signal G reaches the end of the signal path it will extend into the quiescent space and produce an ‘o’ there which will allow other signals to continue passing through it; signal G will also set a new direc pointer there to point to its extension direction, as shown in epochs 2 and 3 in Figure 7.10.

The turning of the extension direction is made by the signal L. When reaching the end of the signal path, it will set a new direc field pointer at its left neighbor cell pointing at the new extension direction. Note that the code field does not get extended by the L signal; only the direc field is influenced by the L signal. This is shown in epochs 6 and 7 of Figure 7.10. When a signal G reaches the end again in epoch 12, it will extend toward the new direction set forth by the L signal, making a new ‘o’ at the code field, as seen in epoch 13. Continuing with this working pattern, we will get a replication arm turning counterclockwise as seen in epoch 24.

Only signals G and L will change the quiescent space into active cells. The other signals are ignored when reaching the end of the signal path. These other signals like A, 0 or 1 are used to carry the SAT binary bits and do not play roles in the replication control, as seen in epochs 7 and 8 of Figure 7.10, where some A’s disappear at the end of the extending path without making any change to the quiescent neighbors.

7.2.3 Self-replication, detachment and new arm extrusion

When the replicating arm finally turns on itself, it will trigger the formation of the detach signal D, which will in turn close the new signal path, form a new loop and finally separate the two loops.

When signal D is generated it triggers the setting of two special flags in its neighbors. Each of the flag belongs to each of the loops. These two special flags have the effect of enumerating the SAT bit sequences and generating new replicating arms. The details about how enumeration is done will be discussed in the following subsection. We look at how the new replicating arm is formed in this subsection.

Consider Figure 7.11. In epoch 33 the arm is closing, which triggers the formation of detach signal D in epoch 34. Signal D then triggers the setting of two special field flags in epoch 35. Note that signal D also modifies the direction of the direc pointer in the new loop to complete the loop in epoch 35. When seeing the appearing of special flags in its neighbors signal D will disappear in epoch 36 together with its underlying direc pointer. This completely separates the two loops.

The two special flags in corners of the two loops will trigger the formation of the signal sequence EF as seen in epochs 38, 39, 40 and 41. The EF signal sequence will form a new replication arm in the following corner as seen in epochs 41, 42 and 44 of Figure 7.11.

7.2.4 Enumerating the SAT sequences

As seen in Figure 7.11, in epoch 35, two '+' flags are formed beside signal D. The left '+' flag immediately causes the code signal A atop it to be changed to binary bit 0 in epoch 36 while itself changes to the '*' flag for generating an arm extrusion sequence next. In the meantime, the signal L atop the right '+' flag changes it to a '−' flag from epoch 36 to 37. The '−' special flag then causes the signal A atop it to change to binary bit 1 in epoch 38 while itself changing to the '*' flag for generating arm extrusion sequence next. In epoch 39 the left '*' special flag disappears since it has generated the EF sequence. The right '*' flag will also disappear in epoch 41 when its EF sequence has also been generated.

The function of the '+' special flag is to make the code signal A change to binary bit 0, and the function of the '−' special flag is to make the same signal A change to binary bit 1. Because of the timing difference in the parent and child loop, the special flag '+' in the child loop will be changed to flag '−' before it influences the following signal A. Therefore, the explored SAT sequences will be different in the parent and child loop. **This is how the SAT rule set enumerates the SAT binary bit sequences.**

Explored binary bits 0 or 1 will stay unchanged and get copied into all further descendent loops, while unexplored signal A's will be gradually reduced to either binary bit 0 or 1, depending on whether they are in the parent or child loop. Starting with only one loop with all unexplored signal A's in the cellular automata space, the replication/enumeration process will eventually produce all possible SAT binary bit sequences with the same number of bits as the original loop. The order in which these bit sequences are generated is dependent on how loops get replicated and normally starts with the bit closest to the signal L. These are representations of all possible binary bit assignments to the variables of a predicate. An example has been shown in Figure 7.2(a) where an original loop with three A's sequence generates all eight possible SAT binary bit assignments in the end. When the enumeration sequence ends, all A's have been explored. Loops will no longer replicate but signals within them will still be cycling around.

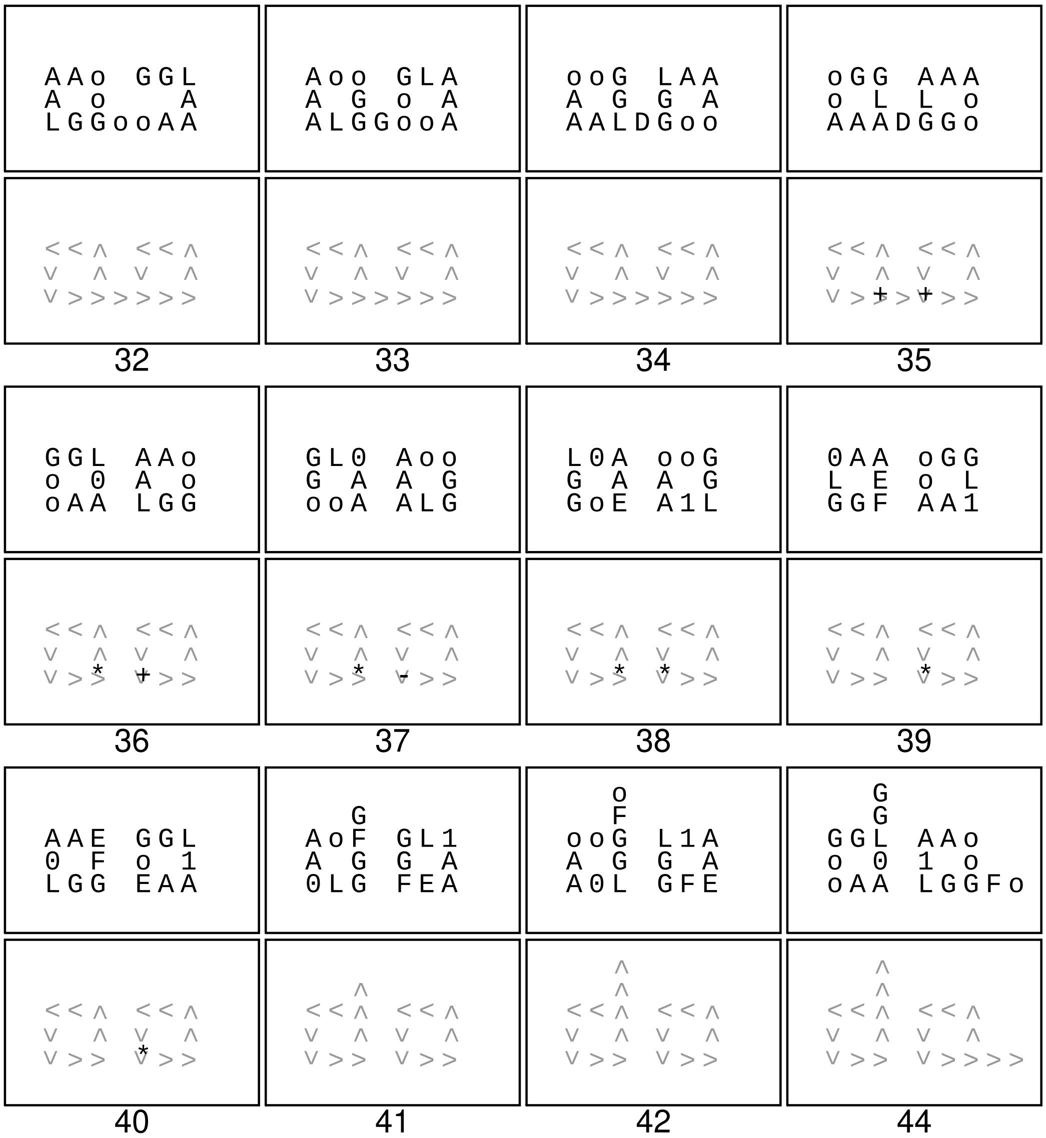


Figure 7.11: The separation of loops and formation of replicating arms. As in Figure 7.10, the direc field is displayed in a separate figure below the code field figure to facilitate reading. It is shown in a light gray color. The special field is shown overlaying the direc field in a darker color.

7.2.5 Collision detection, recovery and preservation of loops

It is possible for two loops to compete for space and collide with each other in the cellular automata space. The SAT rule set makes sure that all collisions are resolved peacefully and no information is lost because of the collision, i.e., the bit sequences will all be generated by the loops despite the collision.

A typical collision situation is shown in Figure 7.12. The lower left and the right loop both try to grow into the middle space between them. The lower left loop is several steps ahead of the right loop, which is just about to extrude its new replicating arm. In epoch 4 the signal L of the lower left loop reaches the tip of its arm. Normally, signal L will change the replication direction to point up but in this particular situation it will cause the special flag '`!`' to be set in its upper neighbor in epoch 5. The reason is that the neighbor there has sensed the lower left loop's intention to grow into its position but has also found out that there is no more space to grow beyond it (since its upper neighbor is occupied by the right loop now). So it sets the '`!`' collision flag. This collision flag will flow backward along the signal path as seen in epochs 6, 7, and 9, killing everything on its way back to the main loop until it reaches the corner of the left loop in epoch 10, where it will then change to an extrusion flag '*'. Remember that the '*' extrusion flag has the function of generating a new replication arm **in the next corner**. Together these two special flags '`!`' and '*' achieve the effect of retracting an old replicating arm and generating a new one.

The replication of the right loop is uninfluenced since when it reaches the edge of the lower left loop in epoch 9, the replication arm of the lower left loop has already moved out of its way, so the right loop still has room to grow and does not have to retract its arm.

A more complicated matter here is that the upper region of the lower left loop has also been taken by another loop already there. While the special '*' flag left in the corner does successfully generate an arm extrusion sequence EF in epoch 13, this EF sequence fails to make a new arm since there is no space left in the upper part of the loop, as seen in epoch 15.

But by the same mechanism, the failed EF sequence leaves another special '*' flag in the new corner as seen in epoch 16. This new special '*' flag generates a new EF sequence in epoch 23 which finally makes a successful new arm in epoch 26.

The detection of collisions needs further explanation. Look back at epoch 4 of Figure 7.12 and compare the situation with epoch 32 of Figure 7.11. In the former case, two loops are colliding with each other and one arm needs to be retracted. In the latter case, it is the closing of the replicating arm for the **same** loop and should be allowed to continue. Given that only local information is available to the colliding central cell, how can it tell one way from the other?

To resolve the problem the color field is used. When replicating toward different directions, loops will possess different colors. This is then used to judge the situation when a collision occurs. If two colliding paths possess the same color, they belong to the same loop and this is the closing of a replicating arm. If these two paths have different colors, we know that a collision has occurred between two different loops. In the example of Figure 7.12, the lower left loop is growing toward the right and the right loop is growing toward the left, so they have different colors, even though the color value is not shown in this figure for the sake of clarity.

7.2.6 SAT clause checking, unsatisfying loops detection and deletion

All of the previous description is about the replication of loops and the enumeration of SAT bit sequences. In this final subsection we consider how monitors work in the cellular automata space

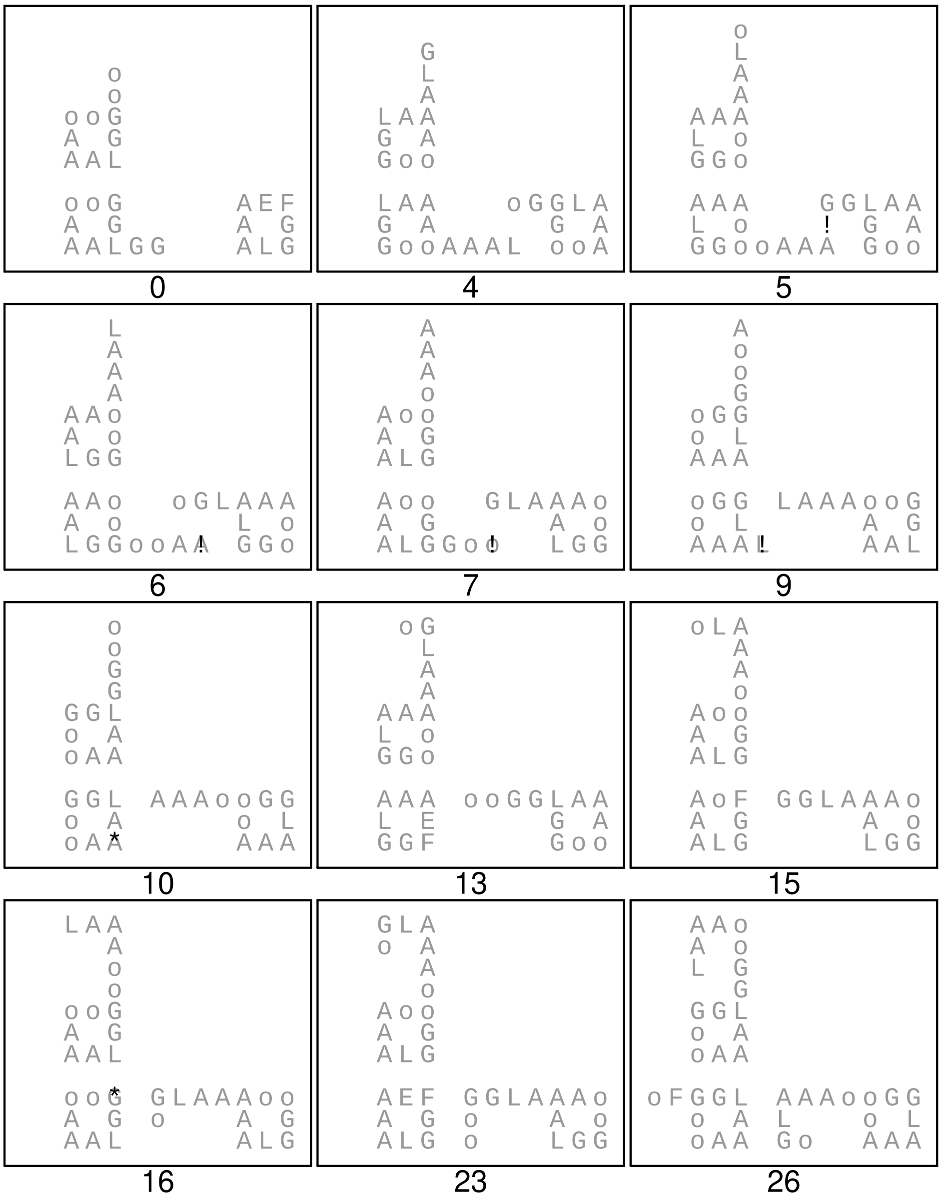


Figure 7.12: The collision of replicating loops. The code field is shown in the background light gray color and the special field is shown in the foreground black color. Note in epoch 5 a special '!' flag was generated in between the two lower loops. It is moving backward toward the lower left loop in epochs 6, 7, and 9. In epoch 10 it changes to a special '*' flag at the lower right corner of the left loop. The '*' special flag induces a new EF sequence in the following epochs. Unfortunately, the EF sequence still fails to generate a new arm at epoch 15 and 16 since the upper region of the left loop is also occupied by another loop above it. Again, In epoch 16 the failed extrusion signal sequence EF leaves a special '*' flag in the upper right corner, which will cause new arm extrusion EF sequence to be generated later. Finally this EF sequence works and a new arm is produced at the left side of the loop in epoch 26.

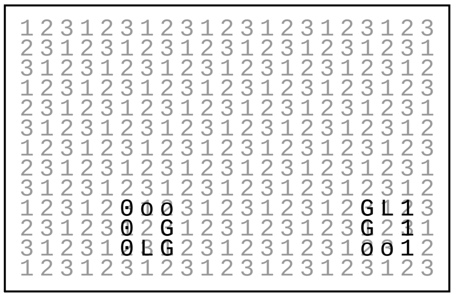


Figure 7.13: Monitors in the clause field. After monitor distribution is done, the clause field will be full of different monitor values which are distributed in such a way that all loops will be touching all different monitors at least once. Shown in the background light gray color is the clause field content. In the foreground there are two loops exactly as shown in Figure 7.2(b). We can see that these two loops touch all three different monitors many times in various cells.

to erase those loops carrying unsatisfying bit assignments for a SAT predicate.

The clause field determines which SAT clause the monitor in each cell of the cellular automata space is checking for. The clause field is independent of all of the other fields used in the SAT rule set since monitors work independently of the self-replicating loops, and have their own governing cellular automata rules. Monitors distribute themselves throughout the clause field at early stages of the cellular automata simulation. Once distributed, the particular clause each of them checks for is fixed and will not change over time. The distribution of monitors is arbitrary as long as we can be sure that each loop will touch all different monitors **at least once** on its path, so that all different monitors will have a chance to check the bit sequence of the loop. Remember that we have different monitors to check for different clauses of a SAT predicate. This requirement is to make sure that all clauses of the SAT predicate will be checked for with all loops.

One arbitrary monitor is initialized by putting it in an arbitrary cell in the cellular automata space at the beginning of the simulation. The following cellular automata rules distribute the *seed* monitor throughout the cellular automata space, but with different clause values at different cells.

```
if (clause==0)
    if (no:clause)
        clause=no:clause%noclauses+1;
    else if (we:clause)
        clause=we:clause%noclauses+1;
```

These rules are simple. If there is no active monitor in a cell, thus clause==0, then the monitor value from either its north or west neighbor is referenced, if any of them exists. This value is modified modulo the total number of clauses (noclauses), and then increased by one. The calculated value is stored in the cell. This rule set guarantees that any monitor in a cell differs to all its neighbors at most modulo one (1 and 3 have a modulo distance of one for a three clauses predicate), as can be seen in Figure 7.13.

A typical clause field content after monitor distribution has been done is shown in Figure 7.13, which is the same as Figure 7.2(b). The clause field contains three different monitors numbered 1, 2 and 3 for the three clauses of predicate $\mathcal{P}$ that we have discussed in the beginning of this chapter. We can see that the distribution of the clause values (the monitors) is even and each loop touches all three monitors at least once.

The clause field makes up the "monitors" we mentioned in Section 7.1. Each monitor will keep checking the SAT binary sequence passing through it for unsatisfying sequences, according to the following cellular automata rules. If an unsatisfying sequence (and the loop which carries it) is found by a monitor, that monitor will set the special field to flag '#', to destroy the loop. This will kill the whole loop carrying the sequence. The process of checking and deletion of loops done by monitors works independently from the other functions of the SAT rule set such as self-replication and the SAT bit sequence enumeration of loops.

First, the clauses of a SAT predicate is represented by four arrays within the cellular automata rule set, such as the following:

```
int noclauses = 3;
int pos1[] = { 0, 1, 1, 2};
int code1[] = { 0, '1:code', '0:code', '0:code'};
int pos2[] = { 0, 3, 2, 3};
int code2[] = { 0, '0:code', '1:code', '1:code'};
```

The arrays above encode the 3 variable SAT problem we have seen in Section 7.1. Remember that there are three conditions to check for this problem, thus `noclauses=3`. These conditions are reproduced below for ease of reference. The first element, with index value 0, of all arrays is always set to zero and is not used. Condition one is encoded by the second element (which has index value 1) of the four arrays. In `pos1[]`, its index 1 element has the value 1, which denotes that the first boolean variable in condition one is variable 1. The index 1 element of `code1[]` is `'1:code'`, which means that this first variable (variable 1) must equal symbol '1', in the `code` field. The index 1 element of `pos2[]` is 3 and the same element of `code2[]` is `'0:code'`, together they mean that the second boolean variable of condition one (i.e. variable 3) must equal '0' in the `code` field. These four index 1 elements of the four arrays precisely encodes condition one of the following:

$$x_1 \wedge \neg x_3$$

$$\neg x_1 \wedge x_2$$

$$\neg x_2 \wedge x_3$$

Similarly, as read off from index 2 elements of these arrays, the first variable of condition two is variable 1, which must equal '0', and the second variable of condition two is variable 2, which must equal '1', etc. All clauses of a predicate is encoded in this fashion. The value of `noclauses` and the size of the arrays are adjusted with respect to the actual predicate length.

If a monitor is set to check for condition 1, it will reference index 1 elements of all four arrays. If a monitor is set to check for condition 2, it will reference index 2 elements of all four arrays, etc. The rules for monitor checking are listed below. These are independent of the SAT predicate being checked:

```
if (special=='.'  && pos==pos1[clause] && code==code1[clause])
     special='?';
else if (special=='?'  && pos==pos2[clause])
     if (code==code2[clause])
          special='#';
     else
          special='.';
```

The first *if* statement checks for the first variable of the condition denoted by the `clause` field value, which is used as an index into the four encoding arrays. If the variable and its value represented in the `pos` and `code` field match the condition values stored in arrays `pos1[clause]` and `code1[clause]`, the alarm flag '?' is set in the current cell, which announces checking for the second variable of the same condition. The second *if* statement in above rules works similarly to the first *if* statement. If the variable and its value are also found to match the condition, the destruction flag '#' is set in the current cell, which will end up destroying the whole loop carrying the sequence. Otherwise, the `special` field is cleared, and checking resumes with the first variable of the condition again.

Since the function of monitors is dependent on the particular SAT boolean predicate they are checking for and is built into the cellular automata rule set, different rule sets will have to be used for different predicates. But this requires just a change of the condition arrays used by the monitors and is not a difficult task. The rules for checking do not have to be modified.

A more detailed example about how monitors work is shown in Figure 7.14. For the sake of clarity, the monitor distribution rule set is disabled for this example. Only one monitor is planted into the cellular automata space, which is located at the upper right corner cell of the top loop. The monitor is set to check for condition 2 of the predicate $\mathcal{P}$, as seen by the number 2 in the `clause` field. No other cell in the cellular automata space has monitors in it for this example. If this No. 2 condition

$$\neg x_1 \wedge x_2$$

is found to be true by the only monitor in this space, then clause 2 of the predicate $\mathcal{P}$ is determined to be unsatisfied by the loop occupying that cell and destruction will begin.

The bottom loop in Figure 7.14 is a parent loop. In epoch 77 the detach signal D causes the two `special` flags '`+`' to be set in both loops. The '`+`' flag in the child loop (top one) is later changed to the '`-`' flag by the L signal in epoch 79, which in turn causes the following signal A to be changed to binary bit 1 in epoch 81. Since the first variable (x_1) has been verified to be '0' by the monitor (`pos==pos1[2] && code==code1[2]`), at epoch 81 the monitor in the upper right corner cell causes the `special` flag '`?`' to be set in its position in epoch 82, which marks checking for the second variable of the condition. Remember that this is the effect of the first *if* statement in the rules above. Continuing in epoch 82 the second bit (which has just been converted from signal A in epoch 81) is just passing over the monitor embedded cell. Since it is the second bit with the value '1', the second part of the condition has also been verified by the monitor (`pos==pos2[2] && code==code2[2]`), which fulfills the No. 2 condition above and proves that clause 2 is not satisfied by the current SAT sequence 01A carried by the loop sitting on top of the monitor. Therefore, in epoch 83 the `special` destruction flag '`#`' is set by the monitor which quickly destroys the whole loop in epoch 88. Note that the third bit has not even been explored yet in this loop.

Recall that the monitor knows the bit 0 on top of it in epoch 81 is assigned to variable x_1 of the predicate $\mathcal{P}$ and the bit 1 on top of it in epoch 82 is assigned to variable x_2 of the same predicate by looking at the `pos` data field, which encodes the variable associated with a bit in the `code` field. The `pos` field content is always translated with bits in the `code` field so that any monitor touching those bits will know which boolean variables they are associated with in the original SAT predicate.

In actual simulations, there are monitors all over the cellular automata space in every cell. As long as a loop is touching all different monitors at least once, which is the case by the current

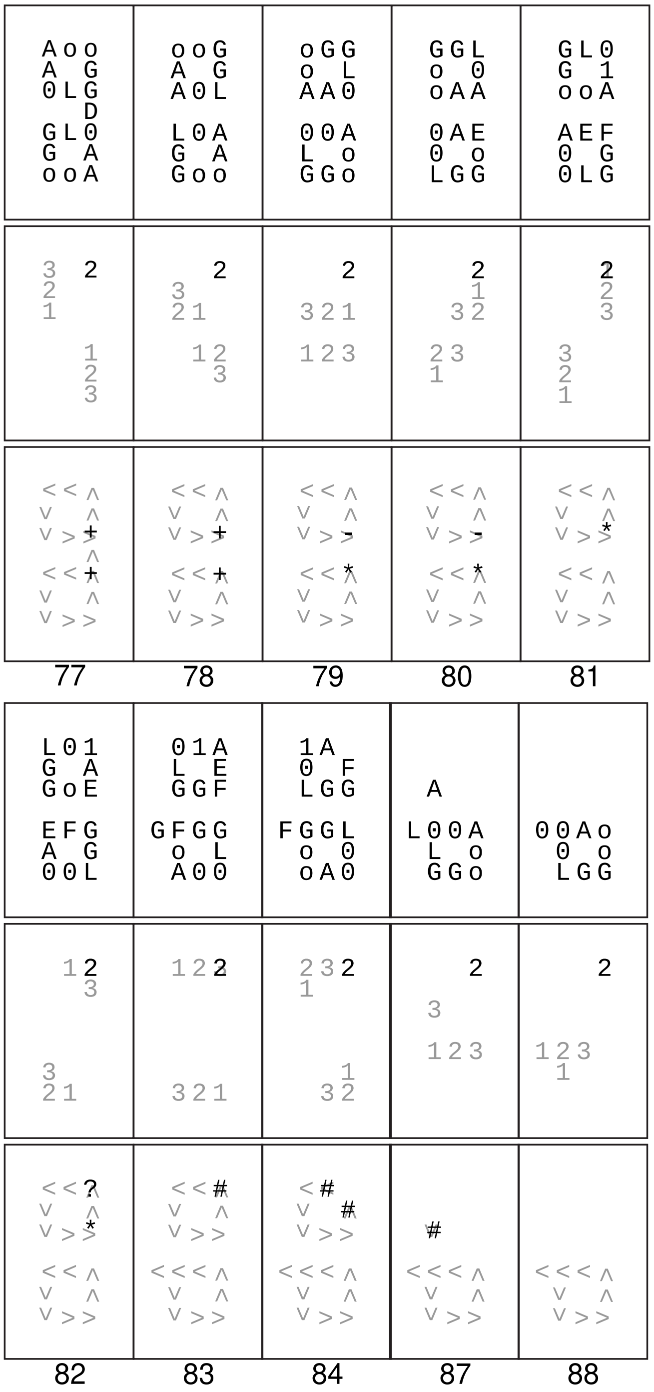


Figure 7.14: Checking for unsatisfying loops and deletion of them by the monitor. The upper row shows the code field, the middle row shows the clause field with the pos field in the background gray color, and the lower row shows the special field with the direc field in the background color for reference. The only monitor is in the upper right corner cell of the top loop which is set to check for clause 2 of the predicate $\mathcal{P}$ we discussed before.

monitor distribution rules, it is guaranteed that unsatisfying loops will all be removed by monitors eventually.

7.3 Analysis

Each cell of the CA space can accommodate one instruction, or code field value. For a self-replicating loop which has n cells on one side (an n by n loop) it needs n code values for its own replication control, plus two code values for the arm extrusion control sequence. The rest of its cells can be used to carry the SAT bit assignments. Therefore, an n by n loop can carry the following number of SAT bits:

$$4 \times (n-1) - (n+2) = 3n - 6$$

For example, for the 3 by 3 loop in Figure 7.14, the loop can carry $3 \times 3 - 6 = 3$ SAT bits.

On the other hand, if we are trying to find the smallest n by n loop which can solve an x-bit SAT problem, we can form the inequality $x \leq 3n - 6$, which gives

$$n \geq \frac{x+6}{3}$$

or

$$n = \left\lceil \frac{x+6}{3} \right\rceil$$

that is the smallest loop size capable of solving an x-bit SAT problem.

The number of iteration steps needed for the replication of an n by n parent loop is

$$5 \times 4 \times (n-1) + (n-1) = 21(n-1)$$

which is calculated based on the fact that an n by n loop has $4(n-1)$ cells and it takes 4 cycles of the signal sequence in the parent loop to replicate the child loop. Plus, it takes one more cycle to extrude the new arm and $n-1$ more steps to move the starting signal G to the new arm position. The child loop is always two steps behind the parent loop, so it takes $21(n-1)+2$ steps to complete.

During each replication generation, one SAT bit is explored. For an x-bit SAT problem, x generations are needed to explore all possible bit assignments. To calculate the cellular automata world size required to solve an x-bit SAT problem, we notice that the maximum expansion along one direction in the CA space for x generations is $x(n+1)$, which is the width for each loop plus one boundary cell, timed by generations. Therefore, the maximum size needed to solve an x-bit SAT problem along one dimension of the cellular automata space is

$$2x(n+1) + n$$

where the extra n is the original loop width. The maximum world size of a dimension is independent of the characteristic of the SAT problem being solved. It is dependent only on the SAT bit number, x.

The estimated number of iterations of the cellular automata universe needed to determine whether a SAT predicate is satisfiable or not is calculated by multiplying the number of generations by the number of replicating steps per generation. The number of replicating steps for a child loop is used in the calculation, which gives

$$21x(n-1) + 2x$$

loop size n $\mathcal{O}(n)$	total cells $4(n\text{-}1)$ $\mathcal{O}(n)$	maximum SAT bits x=$3n$-6 $\mathcal{O}(n)$	rep. steps (parent) $21(n\text{-}1)$ $\mathcal{O}(n)$	rep. steps (child) $21(n\text{-}1)+2$ $\mathcal{O}(n)$	maximum CA width $2x(n+1)+n$ $\mathcal{O}(n^2)$	estimated iterations $21x(n\text{-}1)+2x$ $\mathcal{O}(n^2)$
3	8	3	42	44	27	132
4	12	6	63	65	64	390
5	16	9	84	86	113	774
6	20	12	105	107	174	1284
7	24	15	126	128	247	1920
8	28	18	147	149	332	2682
9	32	21	168	170	429	3570
10	36	24	189	191	538	4584
11	40	27	210	212	659	5724
12	44	30	231	233	792	6990
13	48	33	252	254	937	8382
14	52	36	273	275	1094	9900
15	56	39	294	296	1263	11544
16	60	42	315	317	1444	13314
17	64	45	336	338	1637	15210
18	68	48	357	359	1842	17232
19	72	51	378	380	2059	19380
20	76	54	399	401	2288	21654
21	80	57	420	422	2529	24054
22	84	60	441	443	2782	26580

Table 7.1: Mathematical property for some self-replicating loops

In Table 7.1, the cell numbers, maximum SAT bits, replication steps, maximum world size along one dimension and estimated iteration steps are listed for some different loops. The order of magnitude, or complexity, is also given for each term.

Normally we just need to find **one** satisfying boolean assignment to determine if a predicate is satisfiable or not. That usually requires only the estimated number of iteration steps stated above. But in some cases, the actual iteration steps needed can be critically dependent on the characteristics of the SAT problem being solved. This is especially so when we try to find **all** satisfying boolean assignments to a predicate. An example was given in Figure 7.5 at page 167 where solutions were so abundant that loops crowded each other and could not explore all boolean assignment cases. In such a situation, the self-replicating loop cannot explore all of the satisfying assignments to a predicate and therefore cannot find all satisfying ones.

7.4 Efficiency issues

In the examples of Section 7.1, we can see that the characteristics of a particular SAT problem can dramatically influence the efficiency of finding the satisfying boolean assignments using self-replicating loops. Actually, self-replicating loops may not find all satisfying cases if the cellular automata space gets too crowded. The last column of Table 7.1 lists the estimated number of iteration steps needed to determine if a given SAT predicate is satisfiable or not, but this may not be the exact steps to find the satisfying boolean assignments for the predicate. If we assume that loops in the cellular automata space will never get too crowded to prevent continuing self-replication, then that number will be the exact number of steps to find the satisfying cases. The question now is whether this assumption that loops will never get too crowded is reasonable or that the cellular automata space is usually crowded when solving SAT problems.

First, we need to determine how the satisfiability checking process of the method influence the cellular automata space occupancy density. If loops will never be killed after being born, in five generations loops in the central region of the population will start to have problems continuing replication, despite still carrying unexplored code A's. This has been demonstrated in Figure 7.5. It will prevent the method from finding all satisfying assignments. On the contrary, if during each generation only one loop is kept (the other one being killed, be it the parent or the child loop), the self-replicating loop can finish searching in exactly x generations for an x-bit SAT problem. This is the theoretical best case the method can achieve since it takes at least x generations to explore all x variable assignments in an x variable SAT predicate[2].

The monitor selection process is the key factor here. To have a quantitative understanding of how the selection process influences the efficiency of the method in finding all satisfying boolean assignments, a series of simulations were run, each with a controlled monitor selection behavior. In order to measure the progress of loops toward finding all satisfying assignments during each iteration, the *generation index* of a loop is defined as the number of unexplored code A's within that loop deducted from the number x for an x-bit SAT problem. Therefore, the starting loop, with x unexplored bit A's, is at generation $x - x = 0$. The final, totally explored loop which no longer replicates, is at generation $x - 0 = x$. The reason of using the generation index instead of directly using the number of explored bits is to avoid decreasing the value through time. The progress of the whole cellular automata space configuration toward finding all satisfying boolean assignments is then defined as the average value of the generation index of all loops in the space.

To facilitate comparison, iteration steps are also calibrated with generations. The current iteration number is divided by the number of steps for one full replication generation of a loop to obtain a calibrated generation index of the time step.

In Figure 7.15, the progress curves of four different selection schemes are shown, each is controlled by a custom tailored SAT predicate of the problem being solved. All cases are based on solving 6-bit SAT problems using 4 by 4 loops:

- Curve A represents the theoretical best case where for each generation only one loop is kept, so it reaches all satisfying assignments in exactly 6 iteration generations.
- Curve B represents the worse case where loops are never killed by the monitor, so overcrowding prevents finding all satisfying assignments for the predicate. It never reaches an average generation index of 6.
- Curve C represents a selection scheme where 50% SAT sequences are satisfying but selection occurs only at the final generation (the exploration of the last A bit). It is very slow to find all satisfying assignments in this case since loops in the central region are trapped while waiting

[2]Assuming, of course, that the SAT problem is actually satisfiable; it takes even shorter time to determine a SAT problem is not satisfiable if there is no loop in the space after some early generations.

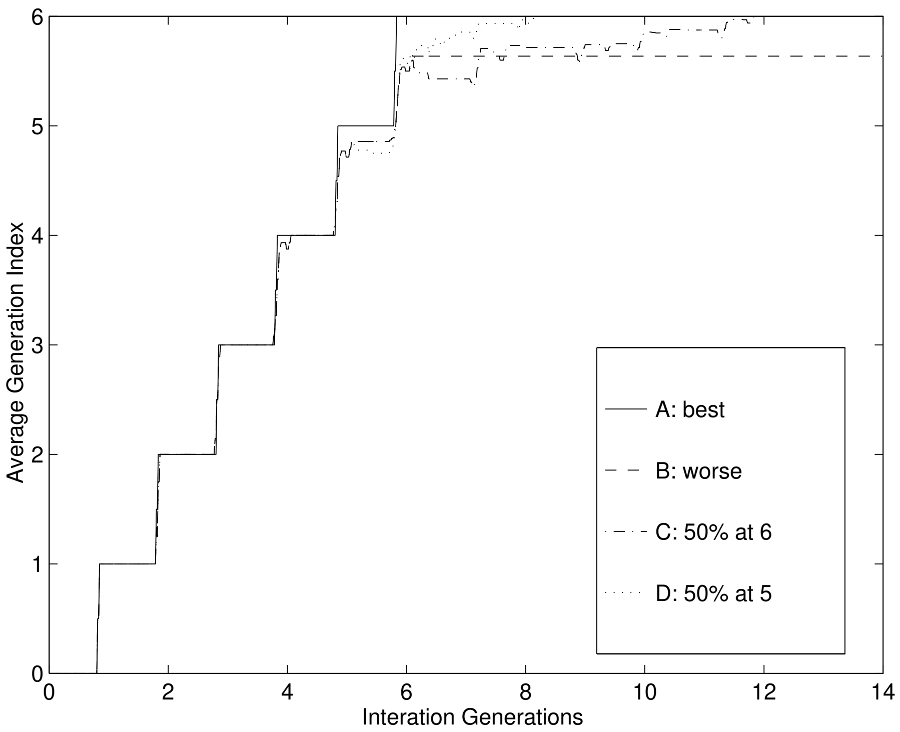


Figure 7.15: The progress speed for different selection schemes. Each selection scheme is controlled by a specially designed SAT predicate of the problem being solved. The vertical axis shows the average generation index of the cellular automata space, which reveals the progress of the cellular automata space toward finding **all** satisfying boolean assignments for a SAT problem. For 6 variable SAT problems of this example, the generation index is always between 0 and 6. The horizontal axis shows the cellular automata iteration steps taken, calibrated by the iteration steps for one full replication cycle of the loops.

for peripheral loops to explore. Outer loops must be erased first to leave space for inner loops to expand. This is a time consuming process and it takes a much longer time to finish (12 iteration generations).

- Curve D represents the same 50% satisfying ratio of all possible boolean assignments but the removal of unsatisfying loops occurs at the fifth bit A (i.e., one generation earlier than Curve C). We can see that the time it takes to find all satisfying assignments is faster than Curve C, at generation 8.

From Figure 7.15 it seems to suggest that the earlier the selection occurs, the faster the method will be in finding all satisfying SAT sequences. To examine this belief two additional 50% curves with even earlier selection stages are plotted in Figure 7.16, together with the two original 50% curves of Figure 7.15. The critical region is zoomed in to facilitate comparison in this new figure. It can be seen that the suggestion above is true, that earlier selection at bit 4 does run faster than selections at bit 5 or 6, but when the selection goes too early in the case of curve D, the overcrowding effect will kick back which prevents the method from finding all satisfiable answers. This is similar to the worse case of Figure 7.15.

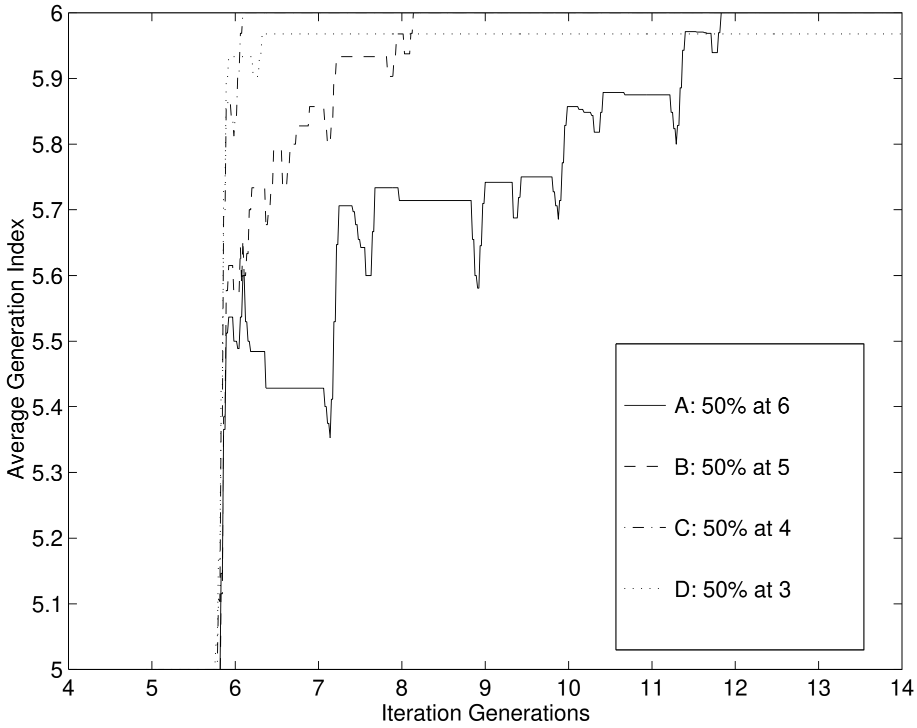


Figure 7.16: The progress speed for 50% satisfying assignments at different selection stages. Again, the satisfying assignment ratios and the selection stages are controlled by custom tailored SAT predicates. The vertical axis shows the progress of the cellular automata space toward finding all satisfying assignments for a particular SAT problem, using the average generation index as the unit. The horizontal axis shows the iteration steps taken, which is calibrated by the steps for one full replication cycle of the loops. To facilitate comparison, only the critical region are shown in this Figure.

We come to wonder if the satisfying ratio will influence the speed of finding all satisfying assignments, too. Figure 7.17 displays solving similar 6-bit SAT problems using 4 by 4 loops, but at different satisfying ratios and selection stages. In this figure, the critical part is also zoomed in on.

- Curve A represents a 50% satisfying ratio with selection occurring at bit 6.
- Curve B represents a 25% satisfying ratio with selection occurring at bit 5 and 6. (We need two distinguishing bits to trim the satisfying ratio down to 25%).
- Curve C represents a 50% satisfying ratio with selection occurring at bit 4.
- Curve D represents a 25% satisfying ratio with selection occurring at bit 3 and 4.

It is obvious from Figure 7.17 that the less the ratio of satisfying assignments, the faster the finding of all satisfying assignments will be at comparable selection stages, but the speed up can never go beyond the theoretical best case. Intuitively, the reason for this behavior is because a smaller

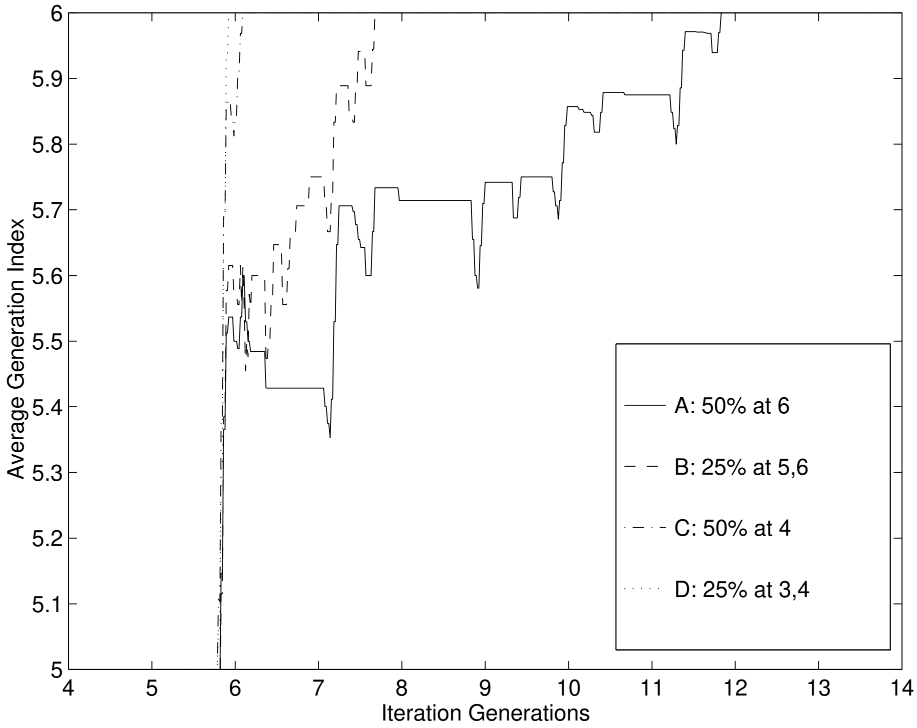


Figure 7.17: Progress curves for different ratios of satisfying assignments and selection schemes. The same coordinate system as in the previous two figures is used here.

satisfying ratio will allow more loops to be killed by the monitor and can decrease the crowdedness of the cellular automata space. This will facilitate loop growth and exploration, thus speeding up the process. Again, the crowdedness of the cellular automata space should be the controlling factor of the efficiency of the method in finding all satisfying SAT sequences.

To better understand how the crowdedness of the cellular automata space correlates with the speed of the method, we must have a quantitative measurement of the crowdedness. The *crowding factor* for a loop is defined to be four minus the number of directions the loop can still grow. For a loop alone in the cellular automata space, its crowding factor is $4 - 4 = 0$, i.e., it is not crowded at all. For a loop which is fully surrounded by other loops, its crowding factor becomes $4 - 0 = 4$, which is also the maximum value of the crowding factor. The average value of crowding factors of all loops in the cellular automata space during each iteration is taken as the crowdedness of the space as a whole.

With the new measurement of the crowdedness of the space, the corresponding crowdedness curves for solving the same four 6 variable SAT problems of Figure 7.15 are shown in Figure 7.18, these are for the best case, worse case, 50% selection at the 6th bit, and 50% selection at the 5th bit, respectively.

- Curve A for the best case has a pulse-like pattern since only one loop is kept during each generation. Whenever a replication cycle is completed one of the two loops in the cellular automata space is identified by the monitors as unsatisfying and is removed, so the average

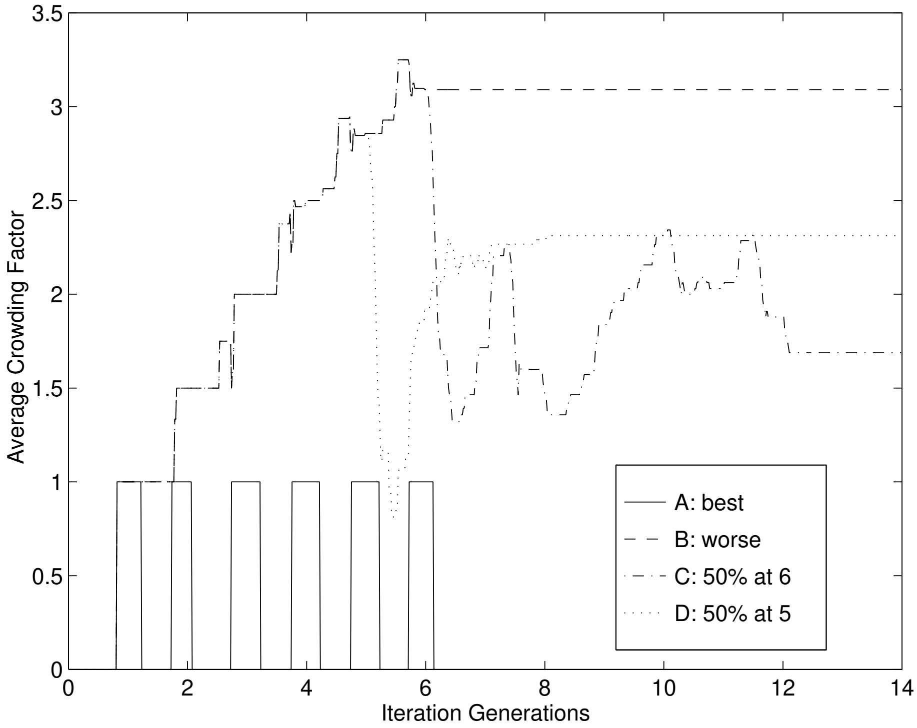


Figure 7.18: The crowdedness curve for different selection schemes. These selection schemes are controlled via four manually designed predicates for the four 6 variable SAT problems. Vertical axis shows the average crowdedness of the cellular automata space during the simulation. Horizontal axis is still the iteration steps calibrated to generations.

crowding factor of the space is jumping between 0 and 1.

- Curve B for the worse case climbs rapidly to a crowding factor around 3 and then stays there. This high crowdedness prevents further exploration. The whole cellular automata space is at a standstill; nothing ever changes afterward.

- Curve C represents the crowdedness for a case with a 50% ratio of satisfying assignments and that selection occurs only at the last (6th) bit. We can see that initially the curve shoots up rapidly like curve B does until it is almost impossible to continue due to overcrowding, then a sudden drop of the crowdedness occurs due to the selection process done by monitors for bit 6. It then fluctuates between the crowding factor of 1.25 and 2.3 for a long time. This is the time when inner loops get the chance to expand, but slowly.

- Curve D, for a case with 50% ratio of satisfying assignments and selection at the 5th bit, behaves similarly to curve C. But its drop of crowdedness occurs one generation earlier than curve C. Since selection occurs when the crowdedness value is lower, the drop of the crowdedness curve is deeper, too. Before the curve is able to climb back to a higher value all satisfying assignments have been found, therefore the curve does not fluctuate.

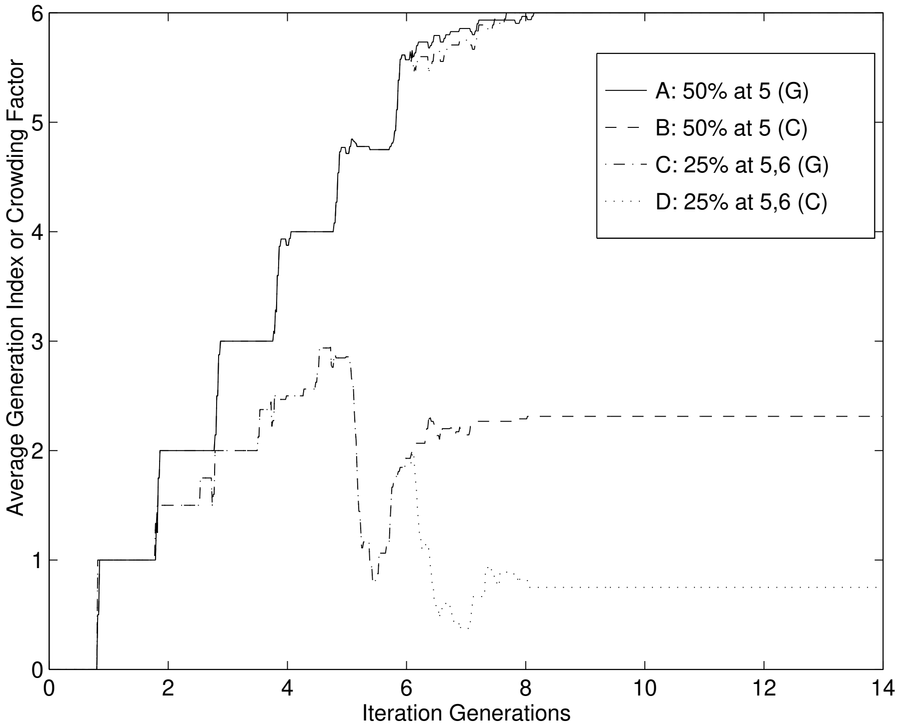


Figure 7.19: The crowdedness curve and the progress curve for two different 6 variable SAT problems. One with a 50% ratio of satisfying assignments, the other with a 25% ratio. Vertical axis shows either the average generation index (for Curve A and C), or the average crowding factor (for Curve B and D), of the cellular automata space. Horizontal axis shows the iteration steps calibrated to generations.

For more comparisons the crowdedness curve and the progress curve are plotted together for two additional cases. One has a 50% satisfying assignment ratio with selection at the 5th bit, and the other has a 25% assignment ratio with selections at the 5th and 6th bit. See Figure 7.19.

It is obvious that the 25% curve and the 50% curve behave similarly in the beginning and the first drop of crowdedness since they both have their first selection occurring at bit 5. But the 25% curve has one more major drop of the crowding factor, which helps to ease off the congestion of the cellular automata space, therefore also helps with the speed. As such, the 25% case is able to finish earlier accordingly.

Our last example is about the long term behavior of the method. In this example, two 9-bit SAT problems are solved using 5 by 5 loops. The satisfying assignment ratio is controlled to be 1/8, with selections occurring at bit 7, 8, 9 for the first problem, and bit 3, 6, 9 for the second problem. The progress curve and crowdedness curve for the two cases are summarized in Figure 7.20. We can clearly see that early selections in the second case can greatly help in lowering the cellular automata space crowdedness and can therefore make self-replicating loops proceed faster in finding all satisfying assignments.

Based on all these observations we can draw the following conclusions. To efficiently find all satisfying assignments for a SAT problem, the cellular automata space cannot become **too** crowded. The average crowding factor must be kept under 2 to avoid slowdown of the search process. If the

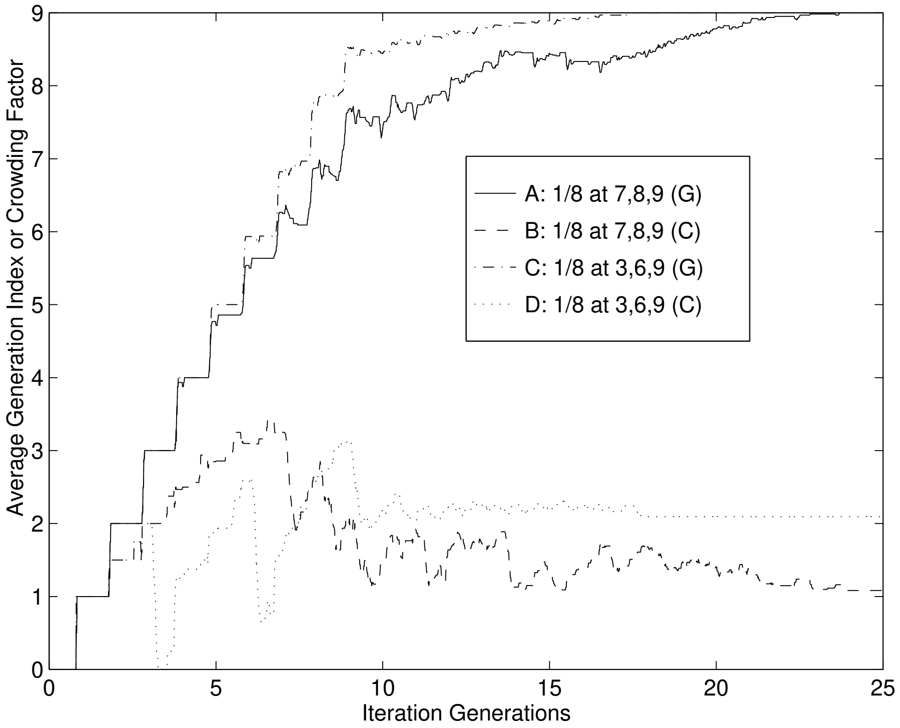


Figure 7.20: Behaviors of solving two 9 variable SAT problems with different selection schemes. The same coordinates of the previous figure are used in this example. Curve A and C show the average generation index of the cellular automata space; Curve B and D show the average crowdedness.

average crowding factor reaches 3 it can halt the whole exploration process, making finding of **all** satisfying assignments impossible. Since the average crowding factor is climbing rapidly during each generation, it must be brought down at least once in three generations if it has to be below 2. Since the satisfying assignment ratio for a SAT problem is closely related to how many times selections are made by the monitors, for an x-bit SAT problem this roughly amounts to a $\frac{1}{2^{x/3}}$ assignment ratio. This can be seen as the upper bound on satisfying assignment ratio for any particular SAT problem if we hope to efficiently solve the problem by self-replicating loops.

In Table 7.2 the corresponding satisfying ratio bounds for same entries of Table 7.1 are listed. The last question is whether these bounds are reasonable for the SAT problem. Of course, it is harder to find **all** satisfying boolean sequences for a particular SAT predicate than to find just a few. If we only want to determine whether a SAT predicate is satisfiable, this method can tell us the answer in fewer number of iteration steps and within a predetermined cellular automata space size, regardless of whether this method can find all satisfying assignments eventually. The reason for this conclusion is the following. For the best case of SAT problems, where there is only one loop left in the cellular automata space during each generation: if there is a loop left in the cellular automata space at the end, we know that the SAT problem being solved is satisfiable. For the worse case of SAT problems where all loops are crowding together in the cellular automata space which prevents the exploration of all satisfying assignments, we still have at least those peripheral loops to prove that the SAT problem is satisfiable. Note that eventually, loops in the periphery will

loop size n	total cells $4(n-1)$	maximum SAT bits $x = 3n - 6$	solvable max. satisfying ratio $1/(2^{x/3})$
3	8	3	50.000000%
4	12	6	25.000000%
5	16	9	12.500000%
6	20	12	6.250000%
7	24	15	3.125000%
8	28	18	1.562500%
9	32	21	0.781250%
10	36	24	0.390625%
11	40	27	0.195313%
12	44	30	0.097656%
13	48	33	0.048828%
14	52	36	0.024414%
15	56	39	0.012207%
16	60	42	0.006104%
17	64	45	0.003052%
18	68	48	0.001526%
19	72	51	0.000763%
20	76	54	0.000381%
21	80	57	0.000191%
22	84	60	0.000095%

Table 7.2: Maximum satisfiability for different SAT problems which self-replicating loops can effectively solve.

all carry a fully explored bit sequence since there is nothing there to prevent them from continuous replication, unless they have fully explored their SAT bits. If these peripheral loops survive the attack of monitors, their bit sequences must be satisfying assignments. Therefore, if we cannot find all satisfying assignments of a given predicate by this method, that alone has proved that the predicate is satisfiable since loops in the periphery are always fully explored and satisfying.

We know that generally for a SAT problem to be hard it must have a very small satisfying assignment ratio, otherwise other methods like random testing for arbitrary values should have solved it already, and it will not be hard. Therefore, these satisfying ratio bounds are believed to be reasonable. Actually, we can see that the lower the satisfying ratio, the harder the SAT problem will be if solved by traditional methods, and the better the self-replicating loop method can perform to find all satisfying boolean assignments.

All selection schemes in this section are designed and controlled in a particular manner in order to see how they affect various parameters. In real life we do not know in advance what the selection scheme will be for a particular SAT problem. All we have will be just a predicate for satisfiability testing. Even so, the efficiency of finding all satisfying assignments may still be improved by doing one or more of the following preparations before we start to breed our self-replicating loops:

- First, do early random tests to determine the satisfying assignment ratio of the predicate in question. If a high satisfying ratio is found with random testing, the predicate is easily satisfiable, and the self-replicating loop method should not be used.

- Rearrange clauses and bits in a way such that the most referenced bits will be the earliest to be explored.
- Change some of the unexplored binary code A's to B's. This will change the exploration sequences among the parent and child loops, which may help a bit in spreading out loops.
- For a big SAT problem more than one seed loops can be started at widespread locations of the cellular automata space. Each of those loops carries a partially explored bit sequence where explored bits can be in random positions of the sequence. This also helps to spread out loops. By adding more initial loops the searching process can also be speeded up, too.

7.5 Discussion

In this chapter it is shown that a controlled evolution in cellular automata space is possible. Here the self-replicating loop is used to carry SAT codes, the potential assignments to the variables of a SAT predicate. The evolution of the loops is controlled in such a manner such that the replication process of the loops will carry out the enumeration of all possible SAT assignments.

A selection process like competition among loops in the cellular automata space is then implemented by rules. Those loops which do not survive the environment pressure (imposed by the monitors) will die, leaving only those loops which carry satisfying SAT codes to the SAT predicate. This is the first demonstration that self-replicating structures can be used to solve problems as well as replicate.

With this new methodology, a new series of cellular automata models can be built, where self-replicating structures can be used to solve computationally expensive practical problems while they are going through the artificial evolution and selection. The codes that self-replicating loops can carry are not limited to only binary bits as in the examples of this chapter; it can be any arbitrary code.

The eventual goal will be to get autonomous and evolving self-replicating agents in the cellular automata space. Those self-replicating structures can make meaningful adjustments, given the hardships and changes of their environment, and produce intelligent solutions. Those environmental hardships and the solutions those self-replicating agents come up with, if mapped properly to a problem domain, can potentially solve many of our real world questions.

7.6 The cellular automata rule listing

```
// ****************************************************************
// ****************************************************************
// SAT Problem Solving Self-Replicating Rules
//                                   written by Hui-Hsien Chou
// ****************************************************************
// ****************************************************************
//

// ****************************************************************
```

```
// ***********  The SAT Predicate Encoding Section  ***************
// *************************************************************
// A SAT predicate is encoded using the following four arrays, pos1[]
// pos2[], code1[], and code2[]. The position arrays listed the
// index of the variables in a SAT predicate, and the code arrays
// listed the expected boolean values of the variable. For example,
// as set in the following arrays, the predicate
//
//      (x_1 and not x_3) or (not x_1 and x_2) or (not x_2 and x_3)
//
// are encoded.

// This variable denotes the number of SAT clauses.
int noclauses = 3;

// First condition position for each clause. The number is the index
// of the binary bit within each SAT sequence.
int pos1[] = { 0, 1, 1, 2};

// First condition listing for each clause.
int code1[] = { 0, '1:code', '0:code', '0:code'};

// Second condition position for each clause. The number is the index
// of the binary bit within each SAT sequence.
int pos2[] = { 0, 3, 2, 3};

// Second condition listing for each clause.
int code2[] = { 0, '0:code', '1:code', '1:code'};

// *************************************************************
// *****************     Default  Rules        ******************
// *************************************************************
// The default action is to maintain no change if none of the rule
// changes the next state value for each field. Therefore, the current
// value is copied over to the next state for each field.

default code=code;
default pos=pos;
default direc=direc;
default clause=clause;
default special=special;
default color=color;

// *************************************************************
// ********* Direction to Neighbor Position Conversion *************
// *************************************************************
// This function maps a directional pointer in the 'direc' field to
// a neighbor position.

nbr PointTo(int x)
  rot if (x=='<:direc') return ea:;
```

```
    else return ce:;

// ****************************************************************
// ***************** Virus Broadcasting Rules *********************
// ****************************************************************
// If there is no virus in a cell, thus clause==0, then copy the virus
// value from either the north or west neighbor, if any of them exists,
// then modify the value by one modulo the total number of the clauses.
// This modified virus value is then stored in the cell.

if (clause==0)
    if (no:clause)
        clause=no:clause%noclauses+1;
    else if (we:clause)
        clause=we:clause%noclauses+1;

// ****************************************************************
// ************ The SAT Rules for Setting Flags ********************
// ****************************************************************

// If special is set at any of the destruction flags, reset it.
if (special=='#' || special=='!')
    special='.';

// If current cell is bound (thus direc!=0) and there is a destruction
// flag nearby, set the destruction flag in the current cell.
else rot if (direc && no:special=='#')
    special='#';

// This rule retracts the replicating arm once a collision is found
// (thus the '!' flag is set). It will copy the retraction flag
// until the end of the corner (judged by we:direc=='<') is reached,
// where the retraction flag is converted to the arm extrusion
// flag '*'.
else rot if (no:direc=='<,1' && no:special=='!')
    if (we:direc=='<')
        special='*';
    else
        special='!';

// This rule determines if
else rot if (code && direc=='<,2' && (special=='.' || special=='?') &&
        (ea:code=='D' || we:code=='D'))
    special='+';

// The arm extrusion failure checking. A failed attempt at new arm
// extrusion will result in flag '*' being set in the corner, which
// allows further attemps at the other direction later.
else rot if (code=='F' && direc=='<,1' && no:direc==0 && we:code=='o')
    special='*';
```

```
// This rule resets flag '+' to '-' after seeing L.
else if (special=='+' && code=='L')
    special='-';

// Code E will always clear the special field.
else if (code=='E')
    special='.';

// This rule resets the special flags '+' or '-' to either quiescent
// state or flag '*', depending the condition
else if (special=='+' || special=='-') {
    if (PointTo(direc):code=='o')
        special='.';
    else if (code=='A' || code=='B')
        special='*';

// The virus checking codes. If a SAT bit is on the current cell (thus
// pos!=0)...
} else if (pos) {

    // see if it violates the first variable expectation of the virus
    if (special=='.' && pos==pos1[clause] && code==code1[clause])
        special='?'; // yes, set the alarm flag ?

    // if alarm flag has been set, see if it violates the second
    // variable expectation. If both are violated, set the destruction
    // flag '#' to infect the loop. Otherwise, reset to normal, the
    // loop is not infected.
    else if (special=='?' && pos==pos2[clause])
        if (code==code2[clause])
            special='#';
        else
            special='.';
}

// ***************************************************************
// ***************     Rules for Bound Cells     *****************
// ***************************************************************

if (direc) {
    // if any of the destruction flag is set, reset everything to 0
    if (special=='#' || special=='!') {
        direc=0; code=0; pos=0; color=0;

    // do checking for real codes only
    } else if (code)

        // if D sees a '+' flag nearby, it disappears
        if (code=='D') {
            rot if (ea:special=='+') {
                code='.';
```

```
        direc='.';
        color=0;
    }

// if the closing of loop is detected, set Code D
} else rot if (direc=='<,2' && nw:direc=='<,1' && nw:code &&
                ne:direc=='<,3' && ne:code) {
        code='D';
        pos=0;

// the rule to close the loop in the child loop
} else if (PointTo(direc):code=='D') {
    rot if (direc=='<,2') {
        direc='<,3';
        code=no:code;
    }

// the rule to generate the EF sequence seeing flag '*'
} else if (code!='o' && PointTo(direc):code=='o' &&
                code!='E' && special=='*') {
    code='E';
    pos='0';
} else if (code=='E')
    code='F';

// the rule to prevent signal E getting copied beyound corner
else rot if (direc=='<' && ea:code=='E' && ea:direc=='<,1'
                && se:code=='F') {
    code='o';
    pos='0';

// same rule to prevent signal F getting copied beyound corner
} else if (code=='o' && PointTo(direc):code=='F')
    code='o';

// rules to explore binary bit A or B to 0 or 1 when seeing '+'
else if (PointTo(direc):special=='+') {
    if (PointTo(direc):code=='A')
        code='0';
    else if (PointTo(direc):code=='B')
        code='1';
    else
        code=PointTo(direc):code;
    pos=PointTo(direc):pos;

// rules to explore binary bit A or B to 1 or 0 when seeing '+'
} else if (PointTo(direc):special=='-') {
    if (PointTo(direc):code=='A')
        code='1';
    else if (PointTo(direc):code=='B')
        code='0';
```

```
            else
                code=PointTo(direc):code;
            pos=PointTo(direc):pos;

        // normal copying rules for signal flow in the loop
        } else {
            code=PointTo(direc):code;
            pos=PointTo(direc):pos;
        }

    // quiescent state changes to 'o' when seeing signal G
    else if (PointTo(direc):code=='G')
        code='o';

// ***************************************************************
// ***************    Rules for Unbound Cells     *****************
// ***************************************************************

} else {

    // quiescent state changes to 'o' when seeing signal G
    rot if (no:code=='G' && (no:special==0 || no:special=='?')
                    && no:direc=='<,3' && ne:direc!='<,2') {
        direc='<,3';

        // check to see if collision occurs
        if (so:direc==0 || so:color==no:color) {// no collision
            code='o';
            color=no:color;
        } else                          // yes, collision, set
            special='!';                 // flag '!' to retract arm

    // EF sequence extruding a new branch
    } else rot if (so:code=='E' && so:direc=='<,1' &&
                    (so:special==0 || so:special=='?') && no:direc==0) {
        direc='<,1';
        code='G';
        color='^';

    // The rule to set turn the signal flowing direction
    // when seeing signal L
    } else rot if (no:code=='L' && (no:special==0 || no:special=='?')
                    && no:direc=='<' && nw:direc==0) {
        direc='<,3';
        if (so:direc==0 || so:color==no:color)
            color=no:color;
        else
            special='!';
    }
}
```

Chapter 8

Conclusion

Cellular automata are massively parallel systems where only strictly local interactions are allowed [von Neumann, 1966; Codd, 1968]. Such a model of parallel computation has the beauty of regularity since all of the "cells" are running the same rule set. It also has the beauty of being scalable since we can build bigger and bigger cellular automata spaces just by adding more processing cells to the periphery of the current space. Assuming that cellular automata simulation chips can be built in quantity, we can easily make up our cellular automata space by connecting those chips in a grid-like manner, with local connections only. There are never long wiring, fan out or special topology problems. There is no bus congestion or scheduling problem either. Cellular automata are immune to these problems which usually occur in other parallel computer systems. All cellular automata processing cells are running simultaneously and never need to wait for one another.

Although simple in its architecture, cellular automata have been found to generate complex behaviors even from a simple rule set. One classic example is the *Game of Life* rule set. It is a very simple cellular automata rule set, yet it dictates a complex, feature-rich world of cellular automata structures such as the block, breeder, glider, etc., each has a special characteristic and many of which are moving and changing in the cellular automata space.

The collective behavior of cellular automata cells can be very interesting too. Actually, most modern cellular automata research is focused on the collective, global behavior of the models. This global behavior can involve self-organization, self-replication, competition or evolution. In a self-replicating structure, none of its components controls or even knows about the whole process of self-replication. It is the distributed yet collaborating behavior that leads people to refer to this kind of self-replicating structures as *artificial life*.

It has been shown in this work how self-replicating structures can be made to emerge in a cellular automata space and how they can be used to solve a classic computer-theoretical hard problem, the SAT problem. In the following sections, the findings and achievements of this work are summarized, and a list of some future research possibilities are discussed.

8.1 Improvements on cellular automata simulation environments

For the purpose of studying the self-replicating cellular automata structures several software tools were built during the course of the research. Some of them represent significant improvements over previously available cellular automata simulation tools. The most notable one is an integrated general purpose cellular automata simulation software based on a graphical user interface.

In the past, very few general purpose cellular automata simulation tools were available to cellular automata researchers. Either they were written with built-in restrictions suitable for specific tasks, limiting their usefulness for modelling other applications (e.g., Xlife for game of life simulations [Bennett, 1989]), or they require special hardware devices to operate, which also limits their availability to researchers [Toffoli & Margolus, 1987]. Although computer speed has increased several orders of magnitude in the past decade, to use a computer to study cellular automata phenomena, researchers generally still have to write their own simulation programs for a specific application and the particular computer platform they use. When I was starting my research several years ago, I could not find any applicable general purpose cellular automata simulation software which fit my needs.

This difficulty has been ameliorated by the creation of the integrated, general purpose cellular automata simulation environment described in Chapter 4. This simulation environment allows arbitrary neighborhood templates and data fields to be defined by the user via an easy-to-use graphical user interface. It takes care of simulation details and also provides for backtracking with the simulations. The backtracking functionality is the only one avaliable in any cellular automata simulation environment that the author knows about.

Another major improvement made in this research over previous cellular automata simulation tools is the creation of a high level cellular automata rule set programming language, Trend.

Cellular automata rule set definition in the past has always been laborious and nonintuitive. Researchers sometimes build the rule set transition functions into their own cellular automata simulation software. Usually, a table is used to represent the rule set transition functions, and the simulation program reads in the table to control the simulation. Either way, rule development is inflexible and inconvenient. A table-like rule set also does not convey to the reader the high level meanings and purposes of the cellular automata rules. In addition, the describable cellular automata rule set complexity is limited by the table size, which in turn is limited by the available computer memory. Altogether, using a table lookup method would have been impractical for the research presented in this work.

The high level cellular automata programming language developed in this work addresses all of the problems mentioned above. The Trend language is a high level language very similar to the popular C programming language, so researchers can easily translate their ideas into rules with this language without worrying about the formating details of the rule table or its size. Readers of a cellular automata rule set expressed in the Trend language can easily understand the semantics of the rules. The Trend compiler automatically converts the rule set into low level code for actual simulation evaluations, so performance is not compromised by the ease of use. Actually, the addition of invariant skipping and caching mechanisms on top of rule set evaluation makes the simulation even more efficient than traditional table lookup cellular automata simulators without these two additional mechanisms.

The most significant benefit of the new cellular automata simulation environment and the high level Trend programming language is that they now allow much more complex cellular automata models to be defined and simulated by computers. To gain an idea about how the envelope of cellular automata modelling complexity has been expanded by the new software tools, some previous software tools for cellular automata simulation and their capabilities are listed below for comparison. This list is not meant to be complete and thorough; it just serves the purpose of giving the reader an idea about how things have been dramatically improved by the new tools.

- XLife. Game of life simulation. Fixed Moore neighborhood (9 neighbors). One bit field depth (alive or dead). Cannot change the rule set or the field depth without reprogramming the code [Bennett, 1989].

- CAM-6[1]. General purpose cellular automata simulator with dedicated hardware accelerator. Several neighborhoods to choose from but all are limited within the 3 by 3 grid of the Moore neighborhood. Four bit planes provide a total of 16 states in each cell. Some limitations on how the bit planes can see each other. The middle level Forth programming language is used to describe cellular automata rules, which is compiled into tables stored in the CAM-6 for simulations [Toffoli & Margolus, 1987].

- xca. Self-replicating loops simulation. Two neighborhoods, Moore and von Neumann, to choose from. Ten states in each cell maximum. Rules are defined in tables [Reggia *et al.*, 1992].

And, by comparison:

- Trend. General purpose cellular automata simulator with no special hardware requirement. Portable to all major computer platforms. Neighborhood template limited only by a 11 by 11 grid but even this could be expanded by reprogramming. A total of 64 bit planes provides up to 2^{64} states in each cell. Bit planes division into different data fields provides great flexibility. High level programming language interface for rule set programming.

8.2 Discovery of a cellular automata rule set for emergent self-replicating structures

With the help of the improved cellular automata simulation tools, an emergent self-replication rule set, which allows a randomly initialized cellular automata space to generate self-replicating structures, was discovered. This is the first rule set reported for emergent self-replication in a cellular automata space. It is also an important demonstration of the self-organization potential of the cellular automata models.

Self-organization is one important aspect of what makes living things different from non-living machineries. Although we can build very complex machines today, none of them can be built by the self-organization of component parts into place, nor can any man-made machines self-replicate. On the contrary, living things can generally direct the replication and construction of themselves. The theoretical study of self-organization as evidenced in the self-replicating behavior sheds light on how we may build more autonomous, self-replicating and self-repairing machines or computer programs in the future.

The emergent self-replication cellular automata rule set employs a *general purpose self-replication rule set* which leads to progressively more diverse and more powerful self-replicating structures. In the past, each individual self-replicating structure developed needed its own supporting rule set to work. It was difficult for different self-replicating structures to co-exist in the same cellular automata space due to rule set differences, not to mention to have them cooperate or compete in the same space. Now, with this new emergent self-replication rule set, structures ranging from

[1] CAM-7 and CAM-8 are supposed to be better than CAM-6, but technical specifications are not yet available to the author.

only 2 by 2 cells to that of the size of the cellular automata space can be supported by the same, general purpose set of rules. This is a very significant step forward for self-replication studies in particular and self-organization studies in general. The general purpose self-replication rule set makes only limited assumptions about the shape of self-replicating structures and only prescribes how self-replicating steps can be performed in a shape and size independent manner. This kind of extraction of function, independent of cellular automata structure size and shape context, is a very important step toward building more complex and general cellular automata structures and rule sets.

In addition, the emergent self-replication rule set demonstrates how a bootstrapping process can be implemented among cellular automata structures so that smaller loops can gradually generate larger and more complex loops. This feature, combined with the characteristic code carrying mechanism described in the following section, makes a fully evolvable cellular automata universe a step closer to reality.

8.3 Evolution and selection with self-replicating loops

The other major research effort of this work was to make the self-replicating loop carrying code in addition to its own self-replicating code. This allows other functions to be built into the self-replication process of the loops. In the past, the sole function of self-replicating structures has been to replicate themselves. There has never been an attempt to also add a purpose to their self-replication. This work presents the first attempt to encode additional information into the self-replicating structures, specifically, information that represents boolean assignments to the variables of a SAT predicate.

With the loop now carrying this additional code, meaningful evolution and selection becomes possible in the cellular automata space. Self-replicating loops no longer die simply because of space limitations, but because of the unsatisfied predicate assignments they carry. In other words, the selection phenomenon is geared around finding satisfying boolean assignments to a SAT predicate, and the self-replicating loops evolve to generate different assignments to a SAT predicate. Only those which carry satisfying assignments will survive the selection process.

The ability to add problem-specific code to the self-replicating loop body and the selection mechanism built into the cellular automata space suddenly make the self-replication phenomena potentially **useful** to us. In addition to the theoretical interest of studying self-replication and self-organization modelled within cellular automata space, we can now also study solving some practical problems by the self-replicating loops.

Combining the application code carrying, artificial selection self-replication rule set together with the general purpose self-replicating and growing features of the emergent self-replication rule set, a new level of cellular automata implementation is near achievement. Now artificial self-replicating structures can not only breed and grow, but can also evolve and compete, and their effort in striving to survive in their universe may at the same time ultimately be helpful in solving real world problems.

8.4 Future prospects

The current primary limitation of the new general purpose cellular automata simulation environment is speed. The simulation environment currently runs on sequential computers. Even though the simulator is not slow compared to other, less flexible simulation software under similar resource limitations and when they are implementing similar cellular automata models, the new simulator can be slow when simulating the much more expanded and complex cellular automata models now made possible by its extended abilities. Therefore, when larger and even more complex cellular automata spaces and rule sets are developed, it will be necessary to port this simulation environment to a more powerful, perhaps massively parallel computer. Thanks to the hardware transparency the simulation environment provides to the user, all Trend language programs will be executable no matter whether the simulator is running on a sequential computer or a parallel supercomputer.

Similar applications for self-replicating loops can be studied further. Instead of carrying just binary codes 0 and 1 as when solving the SAT problem, we can let the loop carry more powerful and more complex codes. When loops are allowed to evolve and change based on the fitness of the complex code sequence they carry, much more complex cellular automata behaviors are expected. This is a new approach toward *artificial life* and one that will generate more interesting results in the future.

Another possible direction of research may be to build a new cellular automata rule set with many functioning units, such as the ability to sense, to search, to communicate and to make boolean decisions. These new units, when combined with the current functions of signal passing, turning, growing and replication in the self-replicating rule set, could potentially create a cellular automata universe so feature rich that these units may go through a *phase transition* process to form cooperating self-replicating structures. This has been suggested by some scientists in the complexity study fields [Kauffman, 1993].

The shape of a self-replicating structure can be changed to some other form to better utilize the available cellular automata space. Currently, the most popular self-replicating structure shape is the rectangle loop shape. This shape is not very economical since quiescent cells within the loop are not used for any purpose. We can try to change the loops to some other forms, probably a solid rectangle, to improve the cell utilization density of the self-replicating structures.

We can also try to allow self-replicating structures to merge, cooperate and communicate with each other. This may lead to the discovery of the first multi-cellular self-replicating structures in the future.

Bibliography

[1] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.

[2] Jon Bennett. Xlife — game of life simulator for x window system under unix, 1989. Available from <ftp://ftp.digital.com/pub/X11-contrib/xlife.tar.Z> as of 7/17/96.

[3] M.C. Boerlijst and P. Hogeweg. Spiral wave structure in pre-biotic evolution. *Physica D*, 48:17–28, 1991.

[4] Arthur W. Burks. Von Neumann's self-reproducing automata. In Arthur W. Burks, editor, *Essays on Cellular Automata*, pages 3–64. University of Illinois Press, Urbana, Illinois, 1970.

[5] John Byl. Self-reproduction in small cellular automata. *Physica D*, 34:295–299, 1989.

[6] Hui-Hsien Chou, James A. Reggia, Rafael Navarro-González, and Jayoung Wu. An extended cellular space method for simulating autocatalytic oligonucleotides. *Computers & Chemistry*, 18(1):33–43, 1994.

[7] E.F. Codd. *Cellular Automata.* Academic Press, New York, 1968.

[8] K. Eric Drexler. Biological and nanomechanical systems: Contrasts in evolutionary capacity. In Christopher G. Langton, editor, *Artificial Life*, pages 501–519. Addison-Wesley, Reading, Massachusetts, 1989.

[9] J. Dana Eckart. A *cellular* automata simulation system, 1995. Available from <http://www.cs.runet.edu/˜dana/ca/tutrial.ps> as of 7/17/96.

[10] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[11] M. Gerhardt, H. Schuster, and J. Tyson. A cellular automaton model of excitable media including curvature and dispersion. *Science*, 247:1563–1566, 1990.

[12] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[13] J.J. Hopfield, José Nelson Onuchic, and David N. Beratan. A molecular shift register based on electron transfer. *Science*, 241:817–820, August 1988.

[14] Stuart A. Kauffman. *The Origins of Order — Self-Organization and Selection in Evolution.* Oxford University Press, New York, 1993.

[15] J. P. Keener and J. J. Tyson. Spiral waves in the Belousov-Zhabotinsky reaction. *Physica D*, 307, 1986.

[16] Christopher G. Langton. Self-reproduction in cellular automata. *Physica D*, 10:135–144, 1984.

[17] Christopher G. Langton. Artificial life. In Christopher G. Langton, editor, *Artificial Life*, pages 1–47. Addison-Wesley, Reading, Massachusetts, 1989.

[18] Christopher G. Langton. Life at the edge of chaos. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 41–91. Addison-Wesley, Reading, Massachusetts, 1991.

[19] Richard J. Lipton. DNA solution of hard computational problems. *Science*, 268:542–545, April 28, 1995.

[20] Rafael Navarro-González, James A. Reggia, Jayoung Wu, and Hui-Hsien Chou. A novel computational method to simulate non-enzymatic self-replication. *Origins of Life*, 24:170–171, 1994.

[21] L. Orgel. Molecular replication. *Nature*, 358:203–209, 1992.

[22] J. Oró, S. Miller, and A. Lazcano. The origin and early evolution of life on earth. *Annual Review of Earth Planet Science*, 18:317–356, 1990.

[23] C. Ponnamperuma, Y. Honda, and R. Navarro-González. Chemical studies on the existence of extraterrestrial life. *Journal of British Interplanetary Society*, 45:241–249, 1992.

[24] K. Preston and M. Duff. *Modern Cellular Automata.* Plenum Press, New York, 1984.

[25] James A. Reggia, Hui-Hsien Chou, Steven L. Armentrout, and Yun Peng. Simple system exhibiting self-directed replication: Annex of transition functions and software documentation. Technical Report CS-TR-2965/UMIACS-TR-92-104, Computer Science Department, University of Maryland, College Park, MD 20742, 1992.

[26] James A. Reggia, Steven L. Armentrout, Hui-Hsien Chou, and Yun Peng. Simple systems that exhibit self-directed replication. *Science*, 259:1282–1288, February 26, 1993a.

[27] James A. Reggia, Hui-Hsien Chou, Steve L. Armentrout, and Yun Peng. Minimizing complexity in cellular automata models of self-replication. In L. Hunter, D. Searls, and J. Shavlik, editors, *ISMB-93: Proc. First Intl. Conf. Intelligent Systems for Molecular Biology, Washington D.C., July 1993.* AAAI Press, Menlo Park, California, 1993b.

[28] Conrad Schneiker. Nanotechnology with Feynman machines: Scanning tunneling, engineering and artificial life. In Christopher G. Langton, editor, *Artificial Life*, pages 443–500. Addison-Wesley, Reading, Massachusetts, 1989.

[29] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines.* MIT Press, Cambridge, Massachusetts, 1987.

[30] Günter von Kiedrowski. A self-replicating hexadeoxynucleotide. *Angewandte Chemie International Edition in English*, 25(10):932–935, 1986.

[31] John von Neumann. *The Theory of Self-Reproducing Automata.* University of Illinois Press, Urbana, Illinois, 1966.

[32] Stephen Wolfram. *Theory and Applications of Cellular Automata.* World Scientific, Singapore, 1986.

[33] Stephen Wolfram. *Cellular Automata and Complexity.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[34] Andrew Wuensche and Mike Lesser. *The Global Dynamics of Cellular Automata.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.