The Trend cellular automata programming environment for artificial life, parallel computing, and simulation research

Hui-Hsien Chou[†1], Wei Huang[†], and James A. Reggia[‡]

[†]*Department of of Computer Science, Iowa State University, Ames, Iowa 50011, USA.* [‡]*Department of Computer Science, University of Maryland at College Park, College Park, Maryland 20742, USA.*

July 9, 2001

Abstract

Trend is a two dimensional cellular automata programming environment with an integrated simulator and a compiler. Trend has many unique features such as simulation backtracking, conflict catching, flexible template design, and no preset space size. The Trend language allows user-defined semi-reserve words, symmetrically rotatable statements, and other constructs specific to cellular automata programming. Trend is available freely both for Unix systems and the Java platform, and it can be used to study artificial life, massively parallel computing, or models of natural phenomena. Designed to be a general purpose programming environment, Trend can be applied to anything a user finds interesting.

Keywords: cellular automata; artificial life; self-replication.

1 Introduction to cellular automata programming

Cellular automata are massively parallel systems consisting of vast number of cells interlinked together *locally*. John von Neumann introduced cellular automata for his self-reproduction study, replacing a kinetic model he used previously [24]. His and subsequent related work has been trying to obtain a deeper understanding of the fundamental information processing principle and algorithm involved in self-reproduction, independent of how that might be physically realized. Cellular automata provide an ideal abstraction for that purpose, and have become a major vehicle for the *artificial life* research [16, 17]. Cellular automata have also been used in massively parallel computing [18], modeling [13] and simulation [23, 25]. Interested readers can find an overview of cellular automata and self-replicating systems in the review article [21]. Also, in the Parallel Computing Journal there has been a special issue dedicated to cellular automata and their applications recently [1].

The main focus of this article is to introduce a new cellular automata programming environment with an associated new programming language, *Trend*. Originally, Trend was developed to assist us in the study of complex cellular automata models of self-replication, including systems where self-replicating structures can emerge from a randomly initialized cellular automata space [6], and self-replicating structures capable of solving the computer-theoretical problem of satisfiability (SAT) [7]. Since Trend was designed to be a general purpose two dimensional cellular automata studying

[1]To whom correspondence should be sent. Current address: 339 Science II, Iowa State University, Ames, IA 50011, USA. E-mail: hhchou@iastate.edu. Homepage: http://www.complex.iastate.edu/hhchou.

tool, its potential application is far beyond what we have used it for thus far. Trend can be used for most research involving two dimensional cellular automata.

This article is organized as follows. We begin by briefly introducing the definition of cellular automata and some of their interesting properties in this section. Next, in Section 2, we introduce the Trend simulation environment and some of its unique features. The Trend programming language is discussed next in Section 3. Because Trend is a new cellular automata programming language that may be unfamiliar to many readers, we provide some Trend programming examples in Section 4 to help the readers understand it better. Readers who like to learn more about Trend can find more examples of Trend programming in its software releases. We also have an online resource page for Trend that will be constantly updated and expanded [22]. Finally, in the last section, we summarize the unique features of Trend, its potential applications, and our future plan to enhance it.

1.1 Cellular automata and their properties

Cellular automata are arrays 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) of a model. Cellular automata can be one, two or multiple dimensional arrays, but one and two dimensional cellular automata are more common due to the difficulty in visualizing higher dimensional models. One dimensional cellular automata have been extensively studied in the past and generated lots of interesting results [26, 27]. Two dimensional cellular automata have also been studied a lot, and are the primary research vehicle for self-replication studies in artificial life [3, 15, 5, 19]. Trend was designed to support two dimensional cellular automata programming, thus we focus mainly on 2-D cellular automata models in this article, although 1-D cellular automata can also be simulated with Trend.

Each cell in a cellular automata space can be in one of n possible *states* specified by a model. These internal states are usually represented by integers starting from 0 to $n - 1$, where 0 is usually the designated *quiescent* or "inactive" state. Non-zero states are "active" states. During each instance of simulated time steps, called an *epoch*, each cell uses a set of rules to determine its next state as a function of its current state **and** the state of its immediate neighbors. Which cells are considered to be the immediate neighbors of a cell vary from model to model. The two most popular choices are the *von Neumann neighborhood* and the *Moore neighborhood*, both are named after their creators. With the von Neumann neighborhood, a cell has four neighbors in its north, east, south and west sides. The Moore neighborhood includes four additional neighbors in its northeast, southeast, southwest and northwest corners. Generally, a cell is always part of its own neighborhood, therefore there are five neighbors in the von Neumann neighborhood and nine in the Moore neighborhood. The von Neumann and Moore neighborhoods are depicted in Figure 1.

In cellular automata cells are commonly governed by the same set of rules collectively called the *transition function*. These models are *homogeneous* cellular automata[2]. For example, one of the famous Game of Life rules [12] states "If a cell is in state 0, and exactly three of its eight neighbor cells in the Moore neighborhood are in state 1, then that cell should change to state 1 at the next instance of time. Otherwise, it remains in state 0." As we can see, each cellular automata rule is based solely on locally-available information to the *current cell* (a.k.a. the *center cell*) that is doing the transition. However,

[2]Cellular automata can also be non-homogeneous where cells follow different rules depending on their positions or the time step. However, Trend was not designed to support non-homogeneous models.

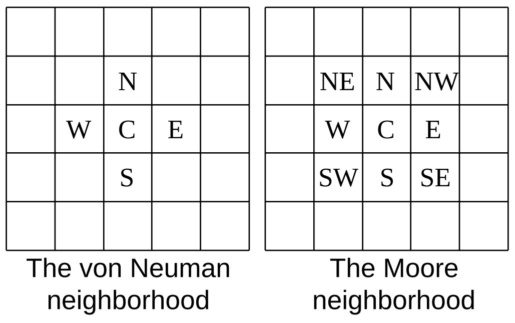


Figure 1: The von Neumann and Moore neighborhoods.

research has shown that cellular automata, through their application of the same rules by all cells simultaneously and repetitively over time, can produce very rich and at times striking behaviors. For this reason, cellular automata are being increasingly used as models in physics, chemistry, biology, and other scientific fields [13, 18, 23, 25].

To summarize, any well-defined cellular automata model requires the following four basic ingredients:

1. a neighborhood template for local references,
2. a set of states with a designated quiescent state,
3. a set of state change rules, i.e. the transition function, and
4. an initial space configuration that starts a simulation.

The quiescent state (0 in most cases) is unique in that, by cellular automata definition, if all cells in a neighborhood is in the quiescent state, then the transition function must produce the quiescent state as the next state of the center cell. In another word, there must be a designated state in a cellular automata model which means “nothing is here”, and since nothing is here, “nothing can happen”. As we will see later, this required cellular automata property has been used in Trend to speed up simulations. A “snapshot” of a cellular automata space at any time step is often called its *configuration*. Naturally, we need a starting configuration to start the simulation.

Some other properties of cellular automata are related to their rotational symmetry. Rotations of two dimensional cellular automata are carried out in 90° increments in our discussion. Therefore, rotating a space four times returns it to its original orientation. Cellular automata neighborhoods can be rotationally *symmetric* (as in the case of von Neumann and Moore neighborhoods above), or they can be *asymmetric*. An asymmetric neighborhood cannot be rotated onto itself, thus references of neighbor positions cannot be symmetry, resulting in *non-isotropic* cellular automata models. In *isotropic* cellular automata models, all neighbors have three other rotational *counterparts* in the neighborhood, and their transition functions can be programmed to treat them indistinguishable from one another. In essence, this makes cells in isotropic cellular automata models unable to tell the absolute north of its containing cellular automata space from east, south or west because the same result of transition mapping will be produced for inputs that are rotationally the same to each other. Another way to look at this is that isotropic cellular automata models can take any rotated initial configuration as input, and always produce the same output with the same rotation. Non-isotropic models may not produce the same rotated result if a rotated initial configuration is given, thus by comparing the difference a cell can determine what is the absolute north of its cellular automata space.

Note that a symmetric neighborhood is a necessary but not a sufficient condition to produce an isotropic cellular automata model. The transition function must also be programmed to be rotationally indifference. In Trend, both symmetric and asymmetric models are supported, although many Trend language constructs are specifically designed to exploit the rotational symmetry of a cellular automata model, and may not be very useful if a model is asymmetry.

1.2 The cellular automata template

In addition to the four basic ingredients above, more information can be specified about cell states to precisely define a complex model. Cell states can be defined to be *strong* or *weak* rotational symmetry. Strong rotational symmetry states are unoriented or rotationally indifference. Weak rotational symmetry states has a sense of their *direction*, and are always in groups of four. Weak states permute among one another in their group under successive rotations of an imaginary *symbol*. For example, the symbol designated $\uparrow$ in von Neumann's work [24] is directionally oriented and permutes among four different cell states $\uparrow$, $\rightarrow$, $\downarrow$, and $\leftarrow$ under rotations. They represent one *component* that can exist in four different orientations. The rationale behind weak rotational symmetry states is that a cell may not be able to tell the absolute north of its universe, it should still be able to tell its left from right, as many asymmetrical biomolecules can do. Rotational symmetry states only make sense in symmetric cellular automata models, but strong and weak states can coexist in the same model.

Modern cellular automata modeling involves the idea of *field division*. Data bits for storing the states of a cell can be subdivided into individual *fields* which can be referenced and computed independently by the transition function. Fields are functional divisions of the state variable of a cell into different bit groups, each encoding a different property of the model being studied. We can think of each field as a slice of a cell variable with its own name and value. Alternatively, we can treat each field as a different variable, as if each cell can now hold many variables. This cellular automata programming technique was originated from the CAM-6 machine [23] and has become an indispensable programming feature of Trend. The use of fields allows *field-oriented* cellular automata programming that can greatly simplifies the effort of a Trend programmer. Nevertheless, fields do not change the semantic of the original cellular automata definition in any way. We will see a few examples in Section 4 that illustrate the power of using fields.

In Trend, all information of a cellular automata model are collectively stored in a *template* file. In addition to the model definition, the following user-defined information are also stored in a template file: the names of neighbors and fields, the symbols to represent cell states, and the colors to represent fields on screen. These programming or visual informations are not part of the cellular automata model definition, but they are useful for the pragmatic purposes of Trend.

1.3 Previously available cellular automata simulators

Before powerful computers became available, studies of cellular automata had to be conducted analytically, as John von Neumann did with his 29-state self-reproducing universal cellular automata machine [24]. Later, when E.F. Codd constructed his 8-state self-reproducing universal machine, he had started using computers to work out some of his rules [8, 3]. It had been necessary in the past for someone interested in studying cellular automata to develop his own program for simulation. That was usually a time-consuming process even for simple models. The transition function was usually represented in a tabular format, as in von Neumann's and Codd's models. A tabular rule set is hard to understand by the others. Although there were several public domain cellular automata simulators when the work described in this article began in 1993, most of them were designed for a specific model, e.g. the Game of Life. What was needed is a general purpose cellular automata simulation environment that removes the burden of software development from cellular automata practitioners, allowing them to concentrate on their modeling effort, not the detail of their computing hardware and software for generating graphics.

Before Trend, the first general purpose cellular automata programming tool was the CAM-6 machine (abbrev. CAM below) [23]. It has both hardware and software components, consisting of a module that plugs into a single slot of a PC, and driving software that operates under PC-DOS. The control software for CAM is written in Forth, which is a semi-high level postfix language where operators are always after their operands in an expression. The Forth language adopted by CAM has been extended to contain a variety of keywords and constructs useful for defining cellular automata rules and for designing, documenting and running simulations. A source program for CAM is converted by the host computer to an internal table stored in the CAM hardware before a simulation starts. Simulations can be visualized on a color monitor with each cell represented by a colored dot.

In CAM, up to four bits are available for encoding the state of a cell, therefore a cell can have up to 16 states. There are some restrictions on the collective use of the four bits, e.g. the center cell cannot access all four bits of its neighbors at once. CAM has a limited set of predefined neighborhoods, and there is no general mechanism to allow the definition of arbitrary new neighborhoods. See [23] for more information about the CAM-6 machine[3]. The CAM machine is historically important because it is the first general purpose cellular automata simulation system that is widely available. It is also the first simulator to use hardware acceleration, to introduce the concept of data fields, and to provide a high level language to describe rules beyond tables. Although the capability of the CAM machine may seem less powerful today, it has inspired several new cellular automata simulators, including the one being presented here.

Another general purpose cellular automata simulator is *Cellular*, which is designed primarily to model physical systems [9, 10]. It consists of a compiler for its high level programming language *Cellang*, a simulator, and a viewer for visualizing the output. The Cellang language has components to describe both the cellular automata models and rules. Its model description determines how many dimensions there are in a cellular automata model, what fields each cell contains, and the bit depth of each field. There are three kinds of statements for rule description in Cellang: **if**, **forall** and **assignment**. A predefined variable `cell` refers to the current cell under consideration, and an assignment to it sets the next state of the current cell. Neighbors can be arbitrarily referenced using an indexing format such as `[0,1]`, which denotes the north neighbor. However, there are no named neighbor positions in Cellang. The Cellular system can be obtained online at [11].

In addition, there were several other cellular automata simulation systems, however, we did not find any of them to meet our need for developing complex cellular automata rules when we started our artificial life research in 1992. As noted by Arthur Burks [3], a pioneer in cellular automata programming, the procedure for cellular automata research is often repetitive:

> *The general principle is this. The investigator partly defines a transition function for his cellular space. He then specifies an initial automaton state and has the computer generate a finite fragment of the resultant cellular automaton in an attempt to produce one of the desired phenomena. He*

[3]There had been efforts to create a more powerful CAM-8 machine in the early 90's, but we do not know if such a machine eventually became commercially available.

> *repeats this step until it succeeds or until he decides it is not promising. In the latter case he tries an alternate partial definition. If the step succeeds, he augments the partial definition further in an attempt to produce other desired phenomena, and repeats the step.*

The ability to quickly program, test, rollback, modify and restart a cellular automata simulation is crucial to a successful cellular automata research. It is especially critical if one wishes to study complex cellular automata models often involved in artificial life research. Additionally, we need a simulator that provides a high level programming language, a large number of cell states, the support for data fields, and an integrated, easy to use user interface. To obtain a simulation tool that contains everything above, we had no choice but to design ours from scratch. The resulted new system, Trend, will be introduced next.

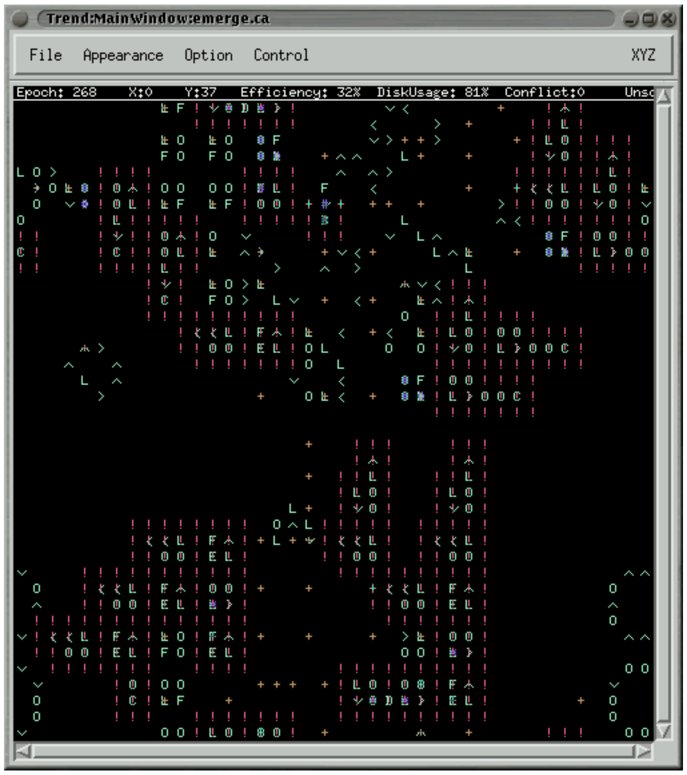
2 The Trend simulation environment

Trend provides a high level language for rule programming, a built-in text editor for entering and correcting programs on-the-fly, a large number of allowable cell states (2^{64}) that can be divided up to 31 data fields, a flexible template design window, a sophisticated mechanism for backtracking of cellular automata simulation steps, a colorful on-screen cellular automata space display, and an interactive graphical user interface. Trend is freely available for both Unix-like operating systems and Sun's Java Virtual Machine, which means it runs on all major computing platforms.

Trend was developed completely from scratch. Originally, a basic framework of the simulation engine was developed and that allowed some simple cellular automata models to be simulated. Then, great efforts were spent on the development of the high level cellular automata programming language and its compiler. Parallel to the Trend compiler development was the development of an evaluation module for executing the virtual machine code generated by the Trend compiler. Virtual machine code was used because it allowed portability between different Unix computing platforms. The Trend language was gradually enhanced to contain a complete set of familiar high level language constructs, and constructs specifically designed for cellular automata programming. Recently, we ported the whole Trend system to the Java Virtual Machine (JVM) with a new compiler which shares the same front-end language parser and semantic checker with the Unix version, but generates native JVM bytecode. This new program, jTrend, further expands the availability of Trend to desktop computers.

2.1 Using Trend

Users provide the cellular automata template information of their models either by loading a previously designed template file (with extension `.tmpl`), or by designing a new one using the Template Design Dialog after Trend has been started. As said above, the template information includes neighbor positions (the neighborhood), data fields, symbols to denote states on screen, etc., and is required for any cellular automata model. After Trend starts, it presents users with two major windows similar to the ones shown in Figure 2 and Figure 3. The Main Window displays the current cellular automata space, and the Text Window displays the current Trend program being edited. A cellular automata configuration file (with extension `.ca`) and a rule set file (with extension `.rule`) can be loaded automatically into the two windows when a template file with the same root name is being loaded if the



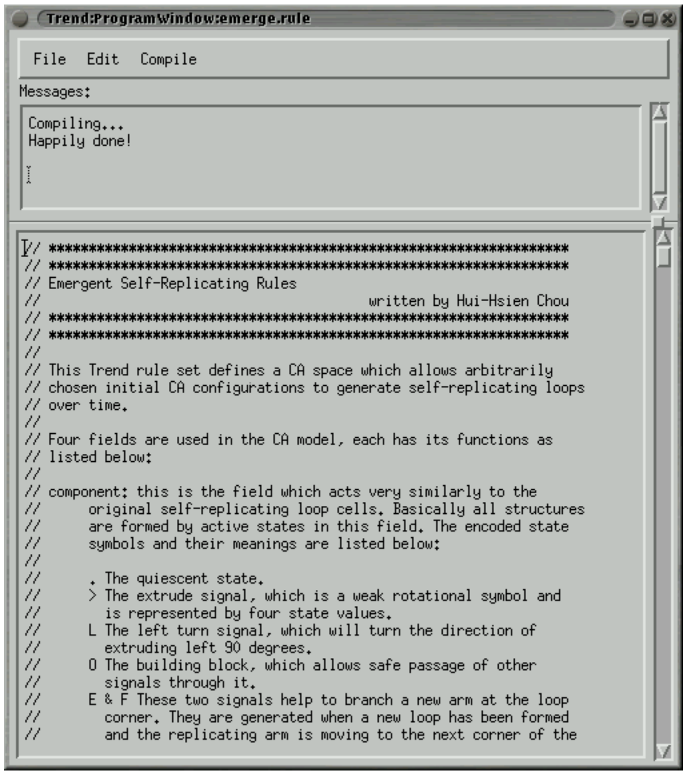
```
Epoch: 268    X:0    Y:37    Efficiency: 32%  DiskUsage: 81%  Conflict:0    Unsc
```

Figure 2: The Main Window of the Trend environment.

two files are located in the same directory with the template file. Users can edit the cellular automata space and the rule program directly within these two major windows.

Once an initial cellular automata configuration and a rule program have been prepared, a simulation can be started using either the keyboard arrow keys or the Control menu in the Main Window. Trend computes a new cellular automata space based on the current one and the rules specified in the Text Window. If there is any error during the simulation, Trend will stop and report the problem. Possible types of runtime errors will be discussed in Section 3 after we have introduced the Trend language. Trend's feature to catch certain runtime errors can be turned off to enhance its simulation speed. Its tracing mechanism is automatically turned on by default, and allows a simulation to go *backward* in time steps for users to discover rule design problems and correct them. Being able to go back and forth in a simulation is probably the most useful feature Trend provides to rule set developers. To enhance the speed of the simulation, Trend also provides an internal caching mechanism to store recent rule set evaluation results. Tracing, caching and catching of errors can all be controlled using the Option menu in the Main Window.

During a simulation users can control lots of things. They can stop the simulation at any time to examine the cellular automata space. They can choose different screen update frequencies, different font sizes and different cellular automata space sizes using the Appearance menu. Trend imposes no preset limit on the size of the cellular automata space that can be simulated. It is determined by individual computer memory size and CPU power, i.e., when users see sluggish performance in Trend, they know they probably have reached the limit on their computers. The cellular automata viewing area in the Main Window can be scrolled around if the actual space size is larger than the



```
Compiling...
Happily done!
```

```
// ******************************************************************
// ******************************************************************
// 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.
//
// Four fields are used in the CA model, each has its functions as
// listed below:
//
// component: this is the field which acts very similarly to the
//      original self-replicating loop cells. Basically all structures
//      are formed by active states in this field. The encoded state
//      symbols and their meanings are listed below:
//
//      . The quiescent state.
//      > The extrude signal, which is a weak rotational symbol and
//        is represented by four state values.
//      L The left turn signal, which will turn the direction of
//        extruding left 90 degrees.
//      O The building block, which allows safe passage of other
//        signals through it.
//      E & F These two signals help to branch a new arm at the loop
//        corner. They are generated when a new loop has been formed
//        and the replicating arm is moving to the next corner of the
```

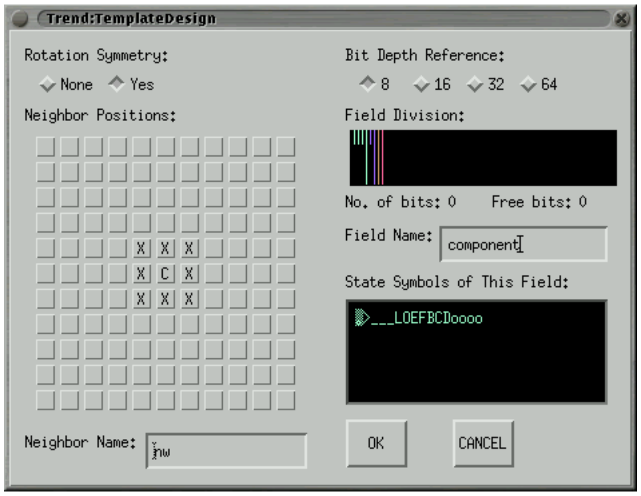
Figure 3: The Text Window of the Trend environment.

Main Window can display at once. Users can run a simulation for as long as they see fit. At the end of a simulation, the cellular automata space can be saved to a `.ca` file for further analysis, or it can be exported to an Encapsulated Postscript (EPS) file for inclusion in a publication.

The Trend simulator is designed to facilitate development of complex cellular automata models. Typically, users start with only a limited knowledge of how their rule sets will work with their models. Gradually, they walk through incremental steps of augmentation to their rule sets while the simulation is going on, i.e. when new rules are needed to lead the present cellular automata space to the next configuration desired by the users, they are simply added to the current rule set. The backtracking mechanism of Trend guarantees multiple levels of undoing and redoing, giving users 100% control to resolve rule conflicts, try out new rules, or modify their cellular automata models altogether. The built-in one-pass compiler in Trend runs almost instantly to compile rule sets of various sizes. There is no preset limit on the size of Trend programs. The execution of compiler generated code is also very efficient, and may become unnecessary once previous evaluation results have been stored in the cache. Because of the way cellular automata rules are usually constructed, the evaluation speed of the compiled code does not necessarily relate directly to the Trend source program size.

2.2 New template design

Users can use the Template Design Dialog shown in Figure 4 to specify new models. This dialog window consists of two main parts, dealing with neighbor and field definitions separately. On the left-hand side is a set of choices specifying the neighborhood structure of the cellular automata model


Trend:TemplateDesign

Rotation Symmetry: None [✓ Yes]

Bit Depth Reference: [✓ 8] 16 32 64

Neighbor Positions:

Field Division:

No. of bits: 0 Free bits: 0

Field Name: component

State Symbols of This Field:

___LOEFBCDoooo

Neighbor Name: nw

Figure 4: The Template Design Dialog. The left portion of the dialog is used for neighborhood definition. A Moore neighborhood is illustrated here, which is symmetric. The right portion is for field definition. Fields are named bit groups that can be used to store data of varying sizes in each cell.

being created. Users have to determine first if their neighborhood should be rotationally symmetric or not. This selection determines if the rotated *if* statement can be used inside their Trend programs developed for the model. For example, if their neighborhood is rotationally symmetric, the following rotated *if* statement can turn a cell to 1 if *any* of its north, east, south or west neighbors is 1:

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

This statement tests the north neighbor of a cell to determine if the assignment should be made when used without the *rot* prefix. If the neighborhood is rotationally symmetric and the *rot* prefix is given, this statement is automatically rotated by Trend to test against the east, south and west neighbors of a cell. More about the rotated *if* statement will be explained in Section 3.

If a neighborhood is not symmetric the *rot* rotation will not be possible because some neighbors may not find their counterparts in the other three orientations of a statement. If a neighborhood is designated to be rotational symmetry, neighbor positions in that neighborhood must be made symmetric, otherwise Trend will complain. In the Moore neighborhod shown in Figure 4, all neighbors can find their three rotational counterparts, so the neighborhood is symmetric. If the neighborhood is not symmetric the rotated *if* statement cannot be used inside Trend programs for this model.

Below the symmetry choice is a matrix of cells for defining neighbor positions. The “center cell”, marked by the letter ‘C’, is required to be included in any cellular automata neighborhood. It is sometimes also referred to as the “current cell” that is being evaluated. Additional neighbors are defined by clicking the other cells surrounding the center cell. If a neighbor is defined it will have an ‘X’ mark and its name will be shown in the Neighbor Name area when it is clicked. If the name of a neighbor is deleted from the Neighbor Name area, that neighbor will be removed from the neighborhood. Trend allows arbitrary neighbors to be defined within an 11×11 region around the center cell. There are no specific reason why the region has to be 11^2. We simply believe this neighborhood size is big enough for most cellular automata models. All neighbor names defined in a neighborhood automatically become the *semi-reserve words* recognized by the Trend compiler.

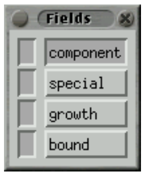


Figure 5: The Fields control palette. Field names are listed in the right column for users to set the focus field. On the left column is a set of check boxes users can use to control field displaying.

On the right-hand side of the Template Design Dialog is a set of choices for field definition. The Bit Depth Reference provides visual advice on how many bits have been allocated to fields and how many are still available. Users can use it to budget bit allocations. However, the Bit Depth Reference has no effect on the actual number of bits allocated for a cell. Trend uses byte as the unit for cell storage and allocates only enough bytes to represent all data fields in a cell. For example, if four fields are defined, each using five bits, then the actual allocated bytes for a cell will be three. Note that the smaller the bit depth, the less storage is required, and the faster Trend simulation will run. The Field Division area defines new data fields. Up to 31 fields can be defined here. If a field is defined, its name will be displayed in the Field Name area when it is clicked. Deleting a field name in this area also removes that field. Once a data field has been defined, users can pick a color for displaying it on screen. If its bit depth is less than eight, users can also define individual state symbols for that field. These are listed in the State Symbols of This Field area. Weak rotational symbols are defined by typing the special underscore '_' character after them. If a state is weak rotational symmetry, the next three states following it are reserved as its rotational counterparts. These four states are displayed on screen by Trend using the same chosen symbol in four different orientations.

Do not confuse the rotational symmetry of the weak states of a field and the rotation symmetry of the neighborhood. The weak states can rotate into one another in a group of four **only if** the neighborhood is symmetric and the rotated *if* statement are used in a Trend program. However, users can always choose a particular symbol to represent a group of four states on screen, even when the neighborhood is not symmetric. In the latter case, the chosen symbol is simply used to display the four states on screen, without having any rotational relationship among the states.

2.3 Displaying, editing and simulation control

In the display area of the Main Window, field states are represented by predefined symbols in the template. Different data fields can be displayed or hidden from view by setting the display checkboxes in the left column of the Fields control palette (Figure 5). The more data fields displayed, the harder it is to recognize each individual symbols. It is up to the users to determine the fields to be displayed at any given moment. Quiescent states (zeros) in each field are not displayed unless users select Show Quiescent Fields under the Appearance menu. Editing in the display area is simple, conducted by right-mouse clicking on a cell to be modified and by selecting a state from the resulted popup menu. The *focus field* that is being edited is controlled by the right column of the Fields control palette. Users can randomize the focus field by using the Randomize dialog available under the File menu. In that dialog, users specify the upper and lower bounds of state values, and a percentage to fill the cellular automata space with these values.

There is a Control menu in the Main Window to control the direction of a simulation. Users can also use the arrows keys on their keyboard to control. The "–>>" menu item (up-arrow key) moves the simulation forward continuously. The "–>" menu item (right-arrow key) moves the simulation forward one step at a time. The "stop" menu item (spacebar key) stops the current continuous forward or backward movement, if any. The "<–" menu item (left-arrow key) backtracks the simulation, and the "<<–" menu item (down-arrow key) starts backtracking continuously until the earliest saved simulation step has been reached, which is usually the first epoch. Backtracking is available only when the tracing mechanism is turned on under the Option menu. The tracing mechanism is very handy because it allows users to pinpoint the exact moment something goes wrong. Although the tracing mechanism saves only the *differences* between iteration steps to a temporary file, it can still eats up disk space for large and long simulations. Trend displays the file system usage status that allows users to stop the tracing mechanism when the file system is full. Additional status information displayed include the current epoch number, the coordinate of the mouse, the efficiency of code evaluations, and if there are any conflict or unsolved errors in the cellular automata space. We will talk more about these two errors in Section 3.

2.4 Program editing, compiling and debugging

Users edit their Trend programs inside the Text Window (Figure 3) and save them to `.rule` files. Trend programs are compiled into virtual machine code through the Compile menu. All compiling errors are shown in the Message area. Since Trend is designed to be interactive, only the first syntax or semantic error is reported. Users can easily modify their program and recompile again within the Text Window.

During a simulation, if runtime errors are discovered, they are reported by Trend in the Message area as well. When any error occurs, users have to correct the error before the simulation can continue. The Jump and Goto commands in the Edit menu are designed to help users move the cursor quickly to a specific character position in the Trend source program for debugging purposes. Commonly, When runtime errors are found, Trend reports the character positions of the errors. Users can select (highlight) a position value reported in the Message area, and choose the Jump command to move the editor cursor to that error spot instantly. The Goto command behaves similarly, but asks the user to provide the character position rather than getting it directly from the Message area.

3 The Trend programming language

Traditionally, cellular automata transition functions are defined in a tabular format, listing all mappings from neighborhood configurations to next state values. This kind of representation has at least three inconveniences:

- When the the number of states or the number of neighbors in a model gets bigger, the table grows exponentially in size, making it hard, if not impossible, to represent all mappings explicitly in the limited computer memory.
- A tabular representation of the transition function is not easy to understand because it does not explicitly convey the ideas of a model to readers. It is hard for readers to understand a table full of plain numbers, just like it is hard for readers to understand low level machine code.
- In addition, it is inconvenient for a cellular automata programmer to manually convert his ideas into the tabular format, but he must do so before he can start a simulation.

Because of these limitations, a high level structured programming language has been developed for cellular automata programming. With this new Trend language, 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 computed based on the cellular automata neighborhood input. Because the Trend language provides most modern programming language constructs, it allows transition functions expressed in it to be very complex, but still quite readable. This greatly extends the power of cellular automata programming when compared to tabular rule sets.

Trend is modeled after the popular programming language C, with cellular automata specific language constructs added to it. These constructs include statements to scan all neighbors in a neighborhood, data types to access data fields and neighbors, special *rotatable literals* representing weak rotational symmetry states, and a rotated *if* statement to exploit the rotational symmetry properties of many cellular automata models. Previous knowledge of C is not required to learn the Trend language, but it may be helpful.

Conceptually, a high level language for cellular automata programming should provide abstract, named data items to denote the data fields within each neighbor in a neighborhood. Based on these data items, new values are computed and assigned to names representing data fields in the center cell. These values are taken as the next state values of that cell. The users' job of filling out the transition table of a cellular automata model in the past now becomes the job of prescribing how the next state values can be computed from the neighbor values. This is a familiar approach to anyone with some high-level language programming experience. Therefore, adapting to cellular automata programming requires just learning a new programming language. The tedious job of the past, sometimes involves creating a complex rule table and a program just to start a basic cellular automata simulation, is no longer needed with Trend.

The Trend compiler is a full-fledged compiler bundled with the simulator program. Although the language is modeled after C, its compiler is not an extended C compiler. Instead, it is a new compiler implemented from scratch because cellular automata rules work in a parallel fashion among cells, which are intrinsically different to a sequential C program.

3.1 A preliminary example

It will help to look at a sample Trend program before we consider the details of the Trend language. The following is the Game of Life rule set written in the Trend language. It states that an active cell will be *born* if it has exactly three active neighbors, and an active cell will *die* if it has less than two or more than three active neighbors. Here life is a field (and the only field) of the Game of Life model.

```
default life=life;       // default is no change to cell values
int count;               // declare an integer variable 'count'
nbr y;                   // declare a neighbor variable 'y'
count=0;                 // initialize counter to zero
over each other y:       // count 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 a Trend program are executed in sequential order from the viewpoint of a cell. The fact that different cells can simultaneously follow different decision routes of the same Trend program makes up the parallelism of cellular automata programming. At the beginning of the example, *default* denotes that the statement after it will be the *default* action taken if no other statements generate a new value for the current cell. Here, it simply states that every cell should stay unchanged by copying over the life value. Customarily, default statements are used to catch all "leftover" conditions not explicitly defined in a Trend program.

After the default statement, *int* and *nbr* are used to declare two variables, `count` and `y`. One is used to store integer values and the other is used to store a neighbor position index. The program starts with an initialization statement to set the counter to zero, then counts the number of active neighbors around a cell. Two rules are used next to determine the situations for birth and death. If none of the two statements is applicable (e.g. if `count` equals to 2), the default statement will be used.

Comments of a Trend program can be marked by either the delimiter `//` or the `/*` and `*/` comment pair. Everything after `//` until the end of a line are ignored by the compiler, and everything enclosed between `/*` and `*/` are ignored by the compiler as well. Nested comment pairs are allowed in Trend.

Trend compiler does one-pass scanning of a source code to speed up the compilation process. Because of this, all variables and functions must be declared before their use. Normally, a Trend program starts with variable and function declarations, followed by the main rules. Default rules can be put anywhere within a program, although it is common to group them together at the beginning or end of the code.

3.2 Reserve words, names, and variables

Like ordinary programming languages, Trend has its own set of reserve words as shown below. These reserve words cannot be used for any other purposes in a program. Reserve words are shown using a special *slanted font face* in this article to distinguish them from the other language elements.

if int nbr fld rot default over void each else while for other break return length

Trend also has a special set of semi-reserve words called *names* that are defined not in the compiler, but in the template information loaded by the simulation program. These semi-reserve words are user defined field and neighbor names for a cellular automata model. They are shown in this article using a sanserif font face. Some possible names are given below:

North East South West ne se sw nw fieldA component life state

Finally, users can define temporary storage space in Trend as variables or arrays, just like other programming languages. Variables or arrays can store temporary computational results, and neighbor or field indices. However, their values are lost once an evaluation is done for a cell. They are shown in this article using a `typewriter font face`. Some examples of variable or array names are the following:

```
x y z count a b c from to pos layer domain next
```

3.3 Data types

There are three data types in Trend. The *integer* type is common to the other programming languages, but the *field* and *neighbor* types are unique in Trend. Types are not interchangeable, but users can write explicit functions to map one type to the others. See Section 4 for examples.

- *int* is an unsigned integer type whose value can be stored in the fields of each cell. In fact, fields accept only data of this type. During each epoch, each field of a cell is expected to get a new value, otherwise a runtime error will be reported. Symbols, such as 'O', 'L', '>', etc., can be defined in a template, then used in a Trend program to represent different field values. These symbolic literal values are converted to integer values by the compiler. The following subsection about data objects provides more information about literals.
- *nbr* is a special type to denote neighbor positions like north, south, east, or west. When combined with the *fld* data type, they can uniquely identify a particular field within a particular neighbor whose value is being referenced in a statement.
- *fld* is a special type to denote fields within a cell. Basically, the total bit depth of a cell (e.g., 8 bits) can be functionally divided into different fields (e.g., 2, 2 and 4 bits each) such that each field encodes a different meaning or function of a cellular automata model.

Data of type *int* can be manipulated and compared just like regular 32-bit unsigned integers in C. Data of types *nbr* and *fld* are more restricted; they cannot be used with mathematical operators since, for example, it is meaningless to add two neighbor positions together. Neighbor and field indices can only be compared by equal '==' and unequal '! =' operators. When an integer data is assigned to a field, only its lower bits that can fit into the size of the field will be preserved.

3.4 Object types

Like most programming languages, Trend provides different object types for programming convenience. Data objects are the basic constituents of expressions. They can be literals, variables, neighbor and field references, array elements, or function calls.

Literals

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

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

are all literals, assuming that the neighbor "North", field "fieldA", the symbol "O" and the weak rotational symbol ">" are all defined in the current template. `'O'` represents the *int* value of the symbol "O" defined for the *current field*, which is determined by the program context. It can take the numeric value 5 if "O" is the sixth symbol defined for the current field (state values start from zero). The literal '>,2' denotes the second rotation value of the weak rotational symbol ">". If the value of ">"

is 6, then '>,1', '>,2' and '>,3' will represent the values 7, 8 and 9 (rotation is clockwise). '>,1:fieldA' denotes the first rotation value of the symbol '>' for the explicitly designated field fieldA. Note that symbolic formats and numeric formats are interchangeable to some extent. We may use 5 and 8 in places of 'O' and '>,2', and vice versa. However, the symbolic format provides more descriptive information to the reader of a program. In addition, only the symbolic format will be automatically rotated in a rotated *if* statement (see below). Therefore, the symbolic format should be used whenever possible. Of course, symbols must all be defined in the cellular automata model template before they can be used to represent state values symbolically.

North: and :fieldA are literal values of *nbr* and *fld* date types, again, provided that they are both defined in the template. Note that there is no numeric format for *nbr* and *fld* literals in Trend, i.e. their exact numerical indices are unavailable. The mandatory symbol ":" **after** a *nbr* object and **before** a *fld* object is required to distinguish them from the integer data objects in Trend. For example, fieldA denotes the value of the field named "fieldA", so it is not a literal and has the *int* data type. However, :fieldA denotes the field index of the field named "fieldA", which is a literal and has the *fld* data type. The value of :fieldA can be assigned only to a *fld* variable. For example, after the following assignment statement,

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

the value of fieldA can be accessed through the *fld* variable `slice_ptr` as if it is a pointer to fieldA. On the contrary, if the assignment statement is written in the following manner, the current value of "fieldA" is copied into the *int* variable `slice_value`. This new copy is therefore independent of "fieldA" after the copy is made.

```
slice_value = fieldA;
```

To summarize, the syntax of the four different formats of literals are the following:

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

where `s` stands for a symbol defined in the model template for the current field and `n` stands for a number. Note that '`[`' and '`]`' denote optional parts of the symbolic literal and are not part of it. Both the rotation count `[,n]` and the field designation `[:field_name]` can be omitted in a symbolic literal. The rotation count, if presents, must be either 1, 2, or 3, representing one, two or three times clockwise 90° rotations of the base symbol.

The field designation (e.g. "`:fieldA`" in `'>,1:fieldA'`) of a symbolic literal is needed only when an effective *current field* cannot be resolved by the compiler. If a literal is used with a field name in an assignment or boolean expression, Trend can extract the current field information from the program context, so an explicit field designation is not needed. For example, in the following, an explicit field designation is not needed for any literal because that can be inferred from the program context:

```
fieldA='O';      // assign the symbolic literal 'O' to fieldA
```

```
    // 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’ as well.
    if (sw:component==’>,1’) component=’>,1’;
```

A statement like “fieldA=’O:fieldA’” is acceptable although it is redundant to specify the field name twice, once in the assignment target and again within the literal.

If a literal is used with a variable or array element whose intended field is not obvious from the program context, Trend compiler cannot figure out the current field for a literal. Therefore, field designations become necessary, as in the following examples (assuming `x` and `y` are *int* variables):

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

Trend reports errors when a current field cannot be inferred from the program context and an explicit field designation is not given in a literal.

Neighbor and field references

Neighbor and field references are unique in Trend and are given in this format:

nbr:*fld*

where *nbr* can be any valid *nbr* data objects: literals, variables, array elements or function calls. Similarly, *fld* can be any valid objects of *fld* type. When combined together this way, they denote the value of a particular field in a particular neighbor, which has the *int* data type. The “*nbr*:” part can be omitted. In that case, the the center cell is taken as the default neighbor.

Variables

Variables in Trend are named storage places to hold temporary data. Their values can be initialized by an assignment statement or a declaration with initialization. If their values are initialized in a declaration, their values cannot be altered later in a program. This semantic is established to ensure that every time a Trend program is executed, it will always have the same starting conditions. Presumably, preinitialized variables serve as tables or constants for a model, and should not be changed. Values of variables are undefined if they are used before being properly initialized.

Arrays

A chunk of temporary storage space can be allocated at once and referenced with the array indexing syntax “`array[index]`”. Only one dimensional arrays are supported in Trend currently. Array index must be of type *int*, but arrays can be of the *int*, *nbr* and *fld* types. For an N element array its index runs from 0 to $N-1$. Trend offers runtime checking of array indexing, and reports errors when an out-of-bound reference is made in a Trend program. The length of an array can be obtained by the special *length* keyword like "`array.length`". Similar to variables, a preinitialized array cannot be altered in a Trend program, and an uninitialized array has undefined values.

Function calls

A function is invoked by giving its name followed by a comma-separated list of arguments enclosed in parentheses. Arguments can be expressions of data objects, including other function calls. A function returns a value of its declared data type. Functions can only be used within expressions; they cannot form stand-alone statements like procedures. Procedures are similar to functions but are declared with the special *void* data type, which means that no value is returned. Therefore, procedures cannot be used in expressions since they do not return values.

Functions return values by the *return* statement. The *return* statement immediately terminates the current function and returns the value of its argument to the caller routine to be used within an expression.

3.5 Declarations

Variable and array declarations consist of a type name, followed by a list of variable or array names separated by commas, and ended by a semicolon. The difference between a variable and an array is that an array has the size part marked by the "[" and "]" symbols. An optional initialization part can follow each variable or array name. An array with initialization will have its size determined automatically by the initializer, therefore its size should not be given. 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, all with
// initialization.
int buf[]={ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
    x[]={ 'O:fieldA', '>,2:fieldB'}, yy=99;

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

// declare fld variables tag and trace, and initialize them
fld tag = :fieldA, trace = :fieldB;
```

Note that a field designation is usually needed when initializing an *int* variable with a symbolic literal like 'O:fieldA' because the compiler cannot determine the designated field of the literal. There can be many “O” symbols in different fields, each having a different state value.

Within a field, symbols for used states must be unique, but unused states can be represented by the same symbol, such as “?”, to denote the impossible occurrence of such states in a cellular automata space. Trend does not prevent the same symbol to be used multiple times within a field to represent different states, because this would prevent the default symbol usage just mentioned. However, if the same symbol is defined for multiple state values in a field, it will always represent the lowest of these state values when used as a symbolic literal in a program. This is sometimes confusing and should be avoided.

A function declaration consist of a type name, a function name, a left parenthesis, formal arguments (if any) separated by commas, a right parenthesis, and the function body. The function body must be a block statement, i.e., it must be enclosed by “{“ and “}”. The following is an example of function declarations. This particular function maps a neighbor position to an integer value which is treated like a pointer in the `dd` field. The symbol `'V'` is used to denote pointers to the center cell from neighbors, e.g., the north neighbor points south with `'V'`, the east neighbor point west with `'V,1'`, and so on. The symbol `'Q'` is used for similar purposes for diagonal neighbors. The extra “leg” of the letter Q is used as the arrow, e.g., the northwest neighbor points southeast with `'Q'`, and the southeast neighbor points northwest with `'Q,2'`.

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

3.6 Mathematical expressions and assignment statements

Mathematical expressions are formed by grouping together *int* data objects using mathematical operators. When evaluated, they produce *int* values that can be assigned to fields as next state values or stored in variables as temporary results. Mathematical expressions can also be used in comparisons to generate boolean values for use in control-flow statements, or they can be arguments to a function or procedure. A mathematical expression will never form a statement by itself. It must be part of a complete statement. Note that only *int* data objects can be operated mathematically. *nbr* and *fld* data objects cannot be combined with mathematical operators, only comparison operators.

Trend’s mathematical operators are a subset of those in C. Listed below are Trend operators in the order of ascending precedence. Operators with the same precedence level are evaluated from left to right:

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

```
( )
```

Parentheses have the highest precedence because they can be used to change the computational order. Multiplication ($*$), division (/), modulus (%), bitwise conjunction (&), bitwise disjunction (|) and bitwise exclusion (ˆ) operators are at the second precedence level. The addition and subtraction operators (+, $-$) have the lowest precedence among mathematical operators. Mathematical operators always take precedence over comparison operators (e.g. >, <, ==, etc.), which in turn take precedence over boolean operators (&&) and (||).

An assignment statement assigns a value obtained from a mathematical expression (for *int* type only) or a data object (for *int*, *nbr* or *fld* types) to a variable, array element, or a field (with the *int* data type only). If a new value is assigned to a field, that value is taken as the next state value of the field. The most common assignment statement contains a left-hand-side, an equal sign (=), a right-hand-side, and a semicolon as the terminator. The left-hand-side can be an object of any of the three data types *int*, *nbr* and *fld*, but in the latter two cases the right-hand-side can only be simple data objects of the same type because *fld* and *nbr* data objects cannot form expressions. The left-hand-side can be a field, variable or array element but it cannot be a literal or function.

The special increment and decrement assignments are formed by a left-hand-side, followed by either the ++ or the $--$ operator, 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 types *nbr* or *fld* since increasing or decreasing their values is meaningless. Again, the left-hand-side can be a field, variable or array element but not a literal or function. Note that Trend treats all assignments as statements, not expressions, so they do not produce values that can be embedded in other expressions. This is different to C.

3.7 Control-flow statements

Several common C control-flow statements are provided in Trend. These statements are used to change the order of execution under various conditions. Control-flow statements rely on boolean expressions to determine the flow pattern. Boolean expressions are formed by combining together comparisons using the logical operators "&&" and "||". "&&" has higher precedence over "||" but parentheses can be used to change the order of evaluation. Boolean evaluation is from left to right within the same precedence level. Just like the C programming language, Trend does *short circuit* boolean evaluation. Whenever a definite result can be determined during a partial evaluation of a boolean expression, parts of that boolean expression may be ignored if they cannot change the outcome of the evaluation. For example, in the following boolean expression,

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

the second comparison never gets evaluated because the first one is always true and makes the whole boolean expression true. It is not advisable to put a function call that may have global side effects in a boolean expression since the function call may not be executed at all!

Comparisons are usually two mathematical expressions or data objects connected by a comparison operator. A single mathematical expression in a place where a boolean result is expected will be taken as testing if that expression is not equal to zero, just like in C. For example, "`a+b`" is the same as a more elaborated "`a+b!=0`" if it is placed in a boolean context. The comparison operators offered in Trend are <, <=, ==, >=, >, and !=. Note that *fld* and *nbr* data objects can be compared only with == and != operators since it is meaningless to compare the value of these types other than equality.

All statements can be grouped together using "{" and "}" to form a block statement. A block statement can appear wherever a single statement can appear. Therefore, it permits multiple statements to be grouped into one single control flow construct. In the following discussion, a `statement` can be either a single or a block statement.

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 it is skipped. In the second case, `statement1` will be executed if the `boolean` is true, otherwise `statement2` will be executed. Nested *if* statements are allowed and the dangling *else* is resolved the traditional way, i.e., to associate it with the nearest *if*.

The *over* statement is unique in Trend. It is used for scanning all neighbors of a cell. It takes the following form:

```
over each [other] nbr_variable statement
```

In essence, it loops through all neighbors of a cell, and assigns each neighbor position index into the `nbr_variable`. The `nbr_variable` can then be referenced in the `statement` that follows. The optional tag "*other*" determines whether the center cell itself is included in the scanning process. For example, the following Trend code determines how many neighbors of a cell are in non-quiescent states (nonzero):

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

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

```
while ( boolean ) statement
```

When the `boolean` expression is true the `statement` will be executed repeatedly until the `boolean` becomes false. If the `boolean` is false on entry, the whole *while* statement will be skipped.

The *for* statement in Trend takes this form:

```
for ( initial_list ; boolean ; modify_list ) statement
```

In Trend, the `initial` and `modify` lists of a *for* statement can only be assignment statements separated by commas. Runtime protection for infinite loops is provided by Trend, but where a dead loop occurs cannot be precisely reported by Trend. Programmers have to make sure that their looping statements can always terminate. A *break* statement can be used inside the `statement` part of an *over*, *while* or *for* construct to forcefully terminate the current iteration and jump directly to the next statement following the loop.

As said earlier, procedure calls are very similar to function calls except that procedures are declared with type *void* and do not return values. Therefore, procedure calls form statements all by themselves. Procedures allow multiple code segments to be repeated with different given arguments.

3.8 The rotated *if* statement

Trend offers strong support for writing cellular automata rules by providing the *rotated if* statement. Rotated *if* statements exploit the symmetry characteristics of most cellular automata models and often reduce the size of Trend programs to 1/4 the size of them without using the rotated *if* statements. In addition, a single rotated *if* statement is easier to understand than quadruply replicated regular *if* statements for dealing with symmetrical conditions. We will see an example shortly.

A rotated *if* statement is formed by adding the reserve word *rot* in front of a regular *if* statement, and by using *rotatable* literals in the boolean expression and the statement body of the *if* statement. All symbolic literals of *int*, *nbr* and *fld* types are rotatable in a rotated *if* statement, but numeric *int* literals are not. For example, the following function can replace the previous one on page 18. It also maps a neighbor position to one of the eight pointer symbols, but requires just two rotated *if* statements instead of the eight regular *if* statements used in the previouse function. In addition, the meanings of the rotated *if* statements are more obvious here with less cluttering of code. They just indicate two specific pointers that should be returned according to the position of the input `x`, and have Trend automatically replicate the other test conditions for returning the other pointers:

```
int ntod(nbr x) {
  rot if (x:==no:) return 'V:dd';       // north points south
  else rot if (x:==nw:) return 'Q:dd';  // 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 results, properly rotated, for different orientations of the same starting cellular automata configuration. Isotropic cellular automata rule sets are important because they make cellular automata simulations independent of the global orientations of the cellular automata space. For some cellular automata models, e.g. the self-replication structures [20, 6], it is especially important that the same outcome will appear in the cellular automata space no matter how the initial cellular automata configuration is oriented. The rationale is that we cannot assume the first self-replicating molecule on earth knows where the north pole of the earth is. Similarly, we cannot assume that a self-replicating structure on a cellular automata space knows where is the top of its embedding space.

The rotated *if* statement in essence is another kind of looping constructs that can be iterated up to four times, testing the four different orientations of its boolean condition. Whenever an orientation

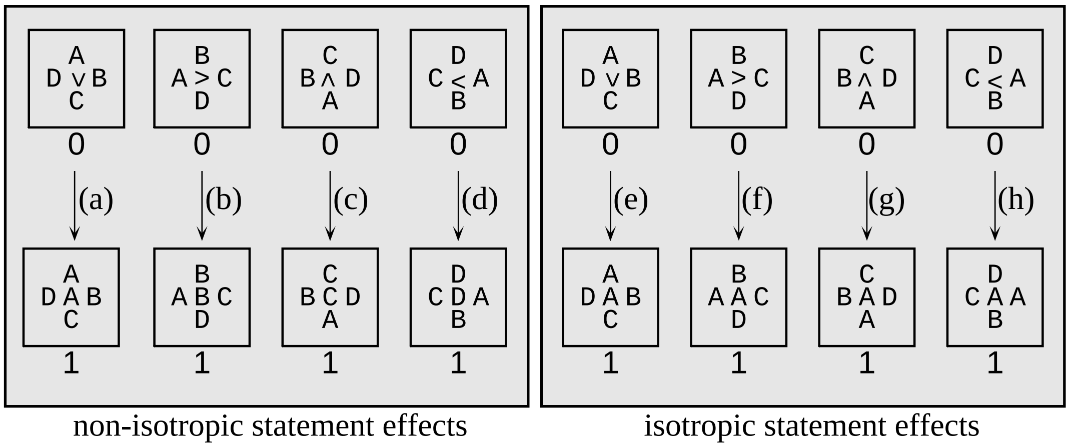


Figure 6: Comparing non-isotropic and isotropic rotated *if* statements. Parts (a), (b), (c) and (d) are the effects of a non-isotropic rotated *if* statement with an input cellular automata configuration in four different orientations. Parts (e), (f), (g) and (h) are the effects of a modified isotropic rotated *if* statement with the same input configurations. We can see that with the isotropic statement, the center cell always gets the same value behind the arrow >, no matter how the input configuration is oriented.

of the statement makes its boolean condition true, its statement body will be executed under the *same* orientation, and the looping will stop. Therefore, the outcome of a rotated *if* statement depends on the order of testing for the four different orientations. It can be non-isotropic. That is, if the same rotated *if* statement is run again with a cellular automata configuration rotated clockwise 90°, it may **not** produce the same result rotated 90° because the test order can be different. It is a programmer's responsibility to ensure an isotropic rule set when using the rotated *if* statement. However, this does not imply that not using the rotated *if* statement will guarantee an isotropic rule set. In fact, it is easier to produce a non-isotropic rule set when rotated *if* statements are not used.

For example, the following non-isotropic rotated *if* statement copies four different values to the center cell depending on the orientation of the cellular automata space, as seen in parts (a), (b), (c) and (d) of Figure 6. Since a rotated *if* statement is evaluated with the upright orientation first, the value in the north neighbor always gets copied, 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.

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

It is easy to guarantee an isotropic rotated *if* statement if one of the following two conditions is met by that statement for all possible input configurations:

- Either *none* of the four orientations of a configuration makes its boolean condition true; or
- Only *one* of the four orientations of a configuration makes its boolean condition true.

If more than one of the four orientations of a configuration make its boolean condition true, only one of them can be chosen to evaluate the statement part of a rotated *if* statement. Therefore, the outcome is non-isotropic because the choice can be arbitrary for different cellular automata configurations. When there is only one choice, then no matter how the input configuration has rotated, that choice will always be taken, so the rotated *if* statement is isotropic. If all rotated *if* statements in a Trend program are isotropic, and there are no other statements that cause non-isotropic evaluations, then the Trend program as a whole is isotropic.

To make a rotated *if* statement isotropic, users can always add (by && operator) a comparison to its boolean condition such that only one of the four orientations can make this comparison true, thereby ensuring that the whole rotated *if* statement is isotropic. This is usually a comparison of weak rotational states used as directional pointers. Since pointers can only point to *one* direction, the added comparison becomes an easy isotropy protection clause. For example, if the previous non-isotropic statement is modified as follows, it becomes isotropic. The modified statement copies the same value to the center cell no matter how the cellular automata space is oriented, as seen in parts (e), (f), (g) and (h) of Figure 6. This statement is evaluated according to where the pointer '>' is pointing, therefore the value of the neighbor behind the pointer always gets copied.

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

A rotated *if* statement takes the following form:

```
rot [(alignment_boolean)] if ( boolean ) statement
```

The optional "`(alignment_boolean)`" has two uses. Sometimes, it is desirable to make the isotropy protection comparison obvious. In that case the isotropy protection comparison can be moved into the alignment boolean. Another use is when multiple orientations make the boolean condition true, thus the statement is non-isotropic, but a specific orientation can be given a higher precedence using the alignment boolean. That is, rather than starting from the upright orientation to test a rotated *if* statement, users can *align* the starting orientation with some pointers in the alignment boolean. No matter what the purpose is, if an alignment boolean is given in a rotated *if* statement, the rotated *if* statement will be tested against the alignment boolean first, until a true value is obtained. Then, starting from the orientation that makes the alignment boolean true, up to *four* more clockwise rotations will be made to test the inner boolean condition until it either becomes true and the statement gets executed, or until the four tests are exhausted and the whole rotated *if* statement is skipped. Note that testing rotations are always clockwise which cannot be changed. Only the starting orientation can be selected by the alignment boolean. For example, the isotropic rotated *if* statement above can be rewritten in the following manner, which still is isotropic:

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

It may seem that if the alignment boolean is added, a rotated *if* statement will automatically become isotropic. This is **not** necessary true. For example, the following rotated *if* statement has the alignment boolean, and is very similar to the previous examples, but it is not isotropic. When applied to the same cellular automata configurations, its effects are given in Figure 7.

```
rot (value<='>,1') if (no:value)
  value=no:value; // copy the neighbor behind or left of the pointer
```

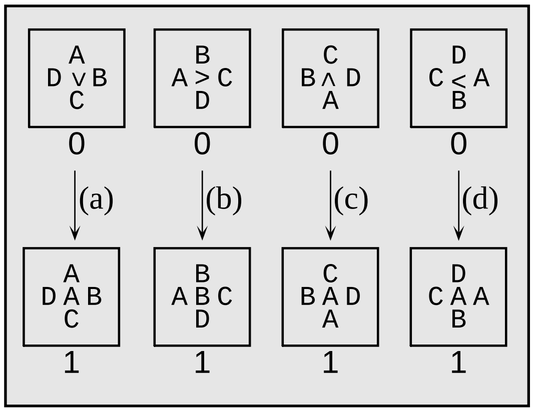


Figure 7: Effects of a nonisotropic rotated *if* statement with the optional alignment boolean.

An *else* part can follow any rotated *if* statement just like the regular *if* statement. Nested *if* statements are acceptable inside or outside a rotated *if* construct, but nested rotated *if* statements are not allowed because they are meaningless. Note that this restriction is applied contextually, not dynamically. Within a rotated *if* statement, calls can be made to functions or procedures that use their own rotated *if* statements. This is allowed because the rotated *if* statement inside a function or procedure is independent of the current rotated *if* statement where the function or procedure is called. In other words, the *rotation* of a rotated *if* statement does not go beyond function boundaries.

3.9 Default statements

Sometimes it is more convenience to designate a *default* statement for a field. That way, if the field has not been assigned a new state value after the execution of regular Trend statements, the default statement can be used to obtain a next state. Default statements are formed by adding the tag *default* in front of any regular Trend statements, except declarations. Often, a default statement just copies the current value of a field into itself, i.e., it simply states that "if none of the Trend rules changes the current value of a field, then that field should stay unchanged". Nevertheless, default statements can use all Trend statements if needed, not just assignment statements.

3.10 Compilation and runtime errors

All compiling errors are shown in the Message area of the Text Window (cf. Figure 3) when they are discovered. Since Trend is designed to be highly interactive, only the first syntax or semantic error is reported. Users can easily modify their code right in the Text Window and recompile it with a few keystrokes. Once a program has no compiling errors, it can be used to start a simulation. However, it may still have runtime errors. Basically, four types of runtime errors are detected by Trend, as follows:

- The *unsolved* errors. During every epoch, each field in each cell must get a new value from the execution of the Trend program. If this is not the case, Trend will report "unsolved" errors during runtime and highlight cells with unsolved fields with a white background. Users need to check the unsolved cells and determine why their values cannot be determined by the Trend program. Frequently, the unsolved errors are caused by failing to provide a default statement for a corresponding field or by forgetting to compile the program before running the simulation.

- The *conflict* errors. Contrary to unsolved errors, conflict errors occur when a Trend program assigns more than one value to a given field during its execution. They are usually caused by logical errors in a program, and are strong hints that some rules may not be correct. Runtime conflict checking can be disabled by users to speed up a simulation, but this is not recommended. Cells with conflict errors are marked with a big "X".
- The *infinite loop* errors. When a looping statement does not terminate during runtime, an infinite loop error occurs. Trend catches this error by automatic timeout after the program execution has been going on for a certain period of time, without returning. Users can abort a simulation when this error is detected, then try to determine where their program gets stuck.
- The *memory fault* errors. An assignment to an out-of-bound array element triggers this error. Trend catches this error and reports where in the source program this error occurs.

A condensed Trend grammar is given in the Appendix to summarize the Trend language in a formal way.

4 Some Trend programming examples

We present a few Trend programming examples in this section to illustrate the Trend language. As a general purpose programming language, Trend's potential applications are enormous and cannot all be presented here. Interested readers can find additional Trend examples in the Trend software distribution.

4.1 Backward compatibility

Sometimes, it may be desirable to run old cellular automata models using Trend. Before the advent of high level cellular automata programming languages, previous cellular automata models used a tabular format to encode the cellular automata transition function in (neighborhood, next state) pairs. Although it is no longer necessary to write cellular automata rules this way, Trend does have the capability to easily adopt such a table into its rules, if necessary. Therefore, old models can be run in Trend without much modification.

The following Trend program implements the well-known, Langton's self-replicating loop [15], using tables. The transition function domain values, each of them consists of the center, north, east, south and west neighbor values of the von Neumann neighborhood, are listed in the array `domain[]`. Their corresponding next state values are listed in the array `next[]`. This translated rule set is short and simple; it just loops through the table trying to find a matching neighborhood value for the current cell, then sets its next state accordingly. The looping statements are independent of the table content, so they can be used to implement other table-based cellular automata models. Note the use of the table *length* operator to determine the table size dynamically in Trend. Because this model is isotropic, the use of a rotated *if* statement naturally extents the neighborhood values for the other three orientations not explicitly listed in the table. Without using the rotated *if*, the tables would have been four times their current sizes.

At first glance, the loop might seem inefficient, since it always scans sequentially through the table to find a matching value. A binary search or hashing algorithm would do better than the sequential

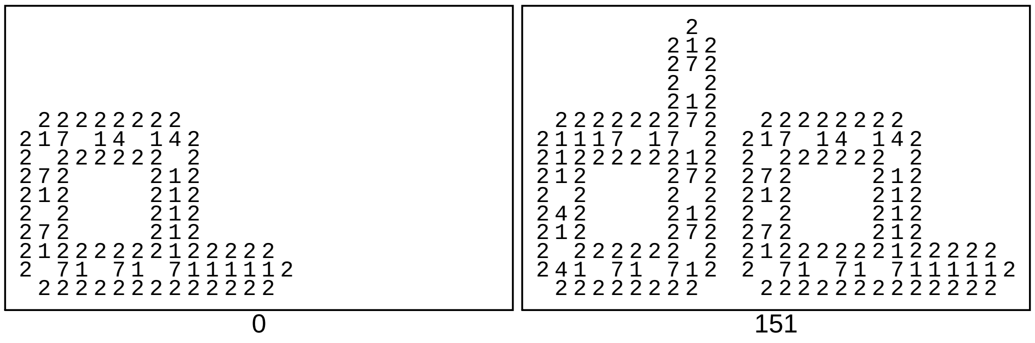


Figure 8: Langton's self-replicating loop. The loop at epoch 0 produces a replica of itself in epoch 151.

search. However, since Trend caches all evaluation results and has its own fast lookup mechanism, it is actually **not** necessary to do any clever table lookup in a Trend program. Within one full replication cycle (151 epochs for Langton's loop), Trend will have all table information in its own internal cache. After that, cell evaluations will no longer go through the compiled code. Instead, Trend will obtain next state values directly from its internal cache.

This program correctly implements Langton's self-replicating loop in Trend. The result is shown in Figure 8. Note that we choose to represent state values with their actual numerical values in order to facilitate comparison with the tables.

```
// This rule set demonstrates how the traditional table based rule
// sets can be easily mapped to Trend's modern programming language
// constructs. Note that there is more than one way of doing this.

/* 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,
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 for 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};

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

default state=state; /* default is no change to a cell */

/* loop through all table entries */
for (i=0; i<next.length; i++)
  rot if (domain[i]==state*10000+no:state*1000+
                     ea:state*100+so:state*10+we:state) {
    state=next[i];
    break;
  }
```

4.2 A simple reaction-diffusion system

Reaction-diffusion systems are 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 catalyst system, catalyst A can help producing catalyst B but inhibit catalyst C, catalyst B can help producing catalyst C but inhibit catalyst A, and catalyst C can help producing catalyst A but inhibit catalyst B. If a mixture of these three catalysts are put together, spiral waves consisting of the three catalysts catching each other's tail can usually occur through time. A well-known reaction-diffusion example is the Belousov-Zhabotinsky reaction [14].

The following Trend program is translated directly from a CAM Forth program in chapter 9 of the book [23]. It demonstrates that the CAM Forth language can be easily translated into Trend. This cellular automata model tries to catch the spirit of the reaction-diffusion system in a very simple manner. There is only one self-annealing reactant in this model. The existence of a reactant is represented by the one bit field value. During each iteration, a cell scans its neighbors to accumulate density information. The Moore neighborhood is used here. The alarm bit is set according to the conditions in the array `table[]`. A cell sets its alarm bit if it has over three active neighbors, or if it has exactly two active neighbors. The alarm bit then triggers the counter to countdown, that in turn disables the cell's value until the the counter becomes zero, when it will enable value again.

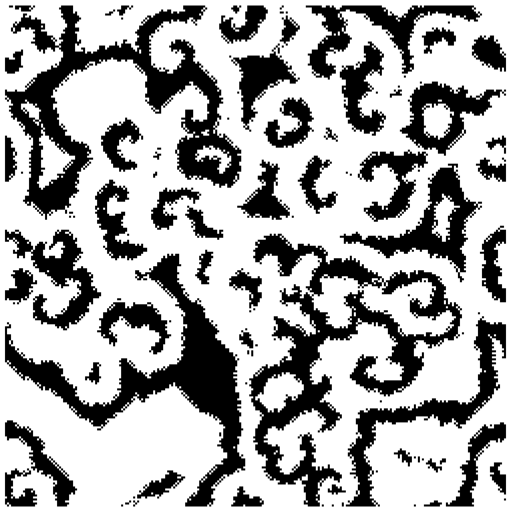


Figure 9: A simple reaction-diffusion simulation. The cellular automata space (250 by 250 cells) is setup by randomly initializing the three data fields value, alarm and counter. It puts individual cells at different phases of the transition. This picture is taken after about 800 iterations of the simulation.

Each cell in the cellular automata space is running independently. Starting from a randomly initialized cellular automata configuration, because of interactions among cells, we can see the formation of spirals after the simulation has been running for a while, as shown Figure 9. For additional simulations with more catalysts, the cellular automata rules given in [2] can be used to replace the very simple program 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 variable used in the scanning of neighbors */

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

/* scanning neighbors */
sum=0;
over each other y: /* loop through all neighbors */
  if (y:value) sum++;  /* if a neighbor is active, increase sum */

/* alarm bit is set according to the density collected */
alarm=table[sum];

/* value is cleared if counter is running, otherwise it is enabled */
if (counter==0)
  value=1;
else
  value=0;
```

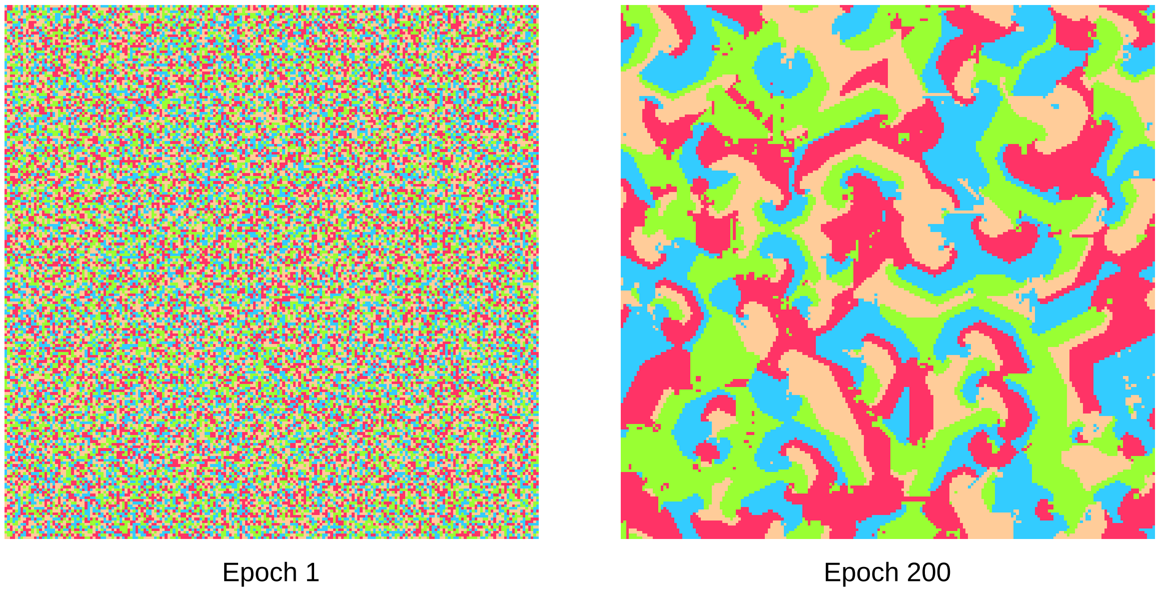


Figure 10: A simple genetic drift simulation. The cellular automata space (200 by 200 cells) is initialized randomly (epoch 1). After 200 iterations, the four different species consolidate their territories into different colored regions.

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

4.3 Scissor, rock, dynamite and paper

The following Trend program is translated directly from a *Cellular* implementation available on the Internet [11]. It demonstrates how the *Cellang* languages can be translated to Trend. This is a *genetic drift* model where four species scissor, rock, dynamite and paper are competing for cellular automata space. The existence of a species is represented by its corresponding nonzero bit field. Each cell can have only one surviving species at any time. During each iteration, a cell scans its neighbors to accumulate competition information, then, depending on its occupying species and the information gathered, a cell may be taken over by another species.

In Figure 10, we can see the result of a simulation after 200 steps from a randomly initialized space. In *Cellular*, different field values can have different colors, but all fields must share the same color scheme, thus multiple fields cannot be displayed at once. In Trend, different fields can be displayed simultaneously with different colors, but each field uses only one color. Trend does use different symbols to represent different states of a field. In order to create the coloring scheme similar to the original *Cellular* implementation, we created four fields in our model for each of the species. Only one of the four can be active in any cell at any time. Initialization is achived by using one additional field, init, which can be randomly initialized and guarantee only one species is occupying a cell at the beginning.

```
// field array for scanning fields to avoid using explicit names
fld choices[] = { :scissor, :rock, :dynamite, :paper};

// threshold value to determine species change
int threshold=3;

// temporary index variables
int i;
nbr y;

// counters for the four different species in a cell’s neighborhood
int sum[4];

// as usual, default is no change to any field
default init=init;
default scissor=scissor;
default rock=rock;
default dynamite=dynamite;
default paper=paper;

// the init field is used only between epoch 0 -> 1 to initialize
// the four different fields, making sure that only one field is
// set. After that, the init field is reset to zero. The initial
// value range for init is 1 through 4.
if (init) {
  choices[init-1]=1;
  init=0;

// the rule set that determines the drift behavior
} else {

  // first the counters are reset to zeros
  for (i=0; i<sum.length; i++)
    sum[i]=0;

  // scan all four fields in Moore neighbors
  over each other y:
    for (i=0; i<sum.length; i++)
      if (y:choices[i]) // an active neighbor, increase the counter
        sum[i]++;

  // determine if a species migration is called for
  for (i=0; i<sum.length; i++)
    if (choices[i] && sum[(i+1)%4]>=threshold) {
      choices[i]=0;
      choices[(i+1)%4]=1;
```

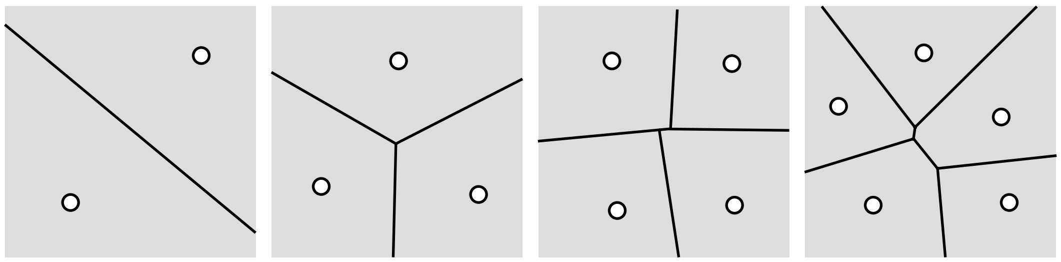


Figure 11: A Voronoi diagram divides the space into regions centered around individual anchor points. The diagram itself is represented by line segments in the space. The Voronoi diagrams for two, three, four and five anchor points are shown.

```
        break;
      }
  }
```

4.4 Massively parallel computing

A Voronoi diagram in a two dimensional space is the collection of points in the space which are in equal distance to their nearby *anchor points*. A Voronoi diagram divides the space into regions centered around these anchor points. For example, the Voronoi diagrams for two, three, four and five randomly chosen anchor points are shown in Figure 11.

A Voronoi diagram can be constructed in a cellular automata space using cellular automata rules running in parallel, as shown in Figure 12. At epoch 0, some random anchor points are set in a space of 250 by 250 cells. An expanding square wave then goes out from each anchor point as seen in epoch 5 and all subsequent epochs. When waves originated from two anchor points collide, they from a Voronoi segment at the crash site. Since waves are expanding at equal speed from anchor points, we know that the Voronoi segments established by them will be in equal distance to all nearby anchor points. The collection of all Voronoi segments forms the Voronoi diagram for the cellular automata space. Note that cellular automata Voronoi diagrams are different to geometric Voronoi diagrams because distances are measured by the L_∞ metric, i.e. $dist(p,q) = \max(|p_x - q_x|, |p_y - q_y|)$, in cellular automata. In a geometric space the Voronoi segments can be seen as formed by the collision of expanding cycles from anchor points.

One way to examine the correctness of a Voronoi diagram is to see if all of its segments are connected. If there is any unconnected segment, the Voronoi diagram is not fully constructed or may be incorrect. In Figure 12, we can see that each square wave gradually claims a portion of the cellular automata space for its anchor point. By epoch 40, a Voronoi diagram has been fully established with no disconnected segments. Nothing will change after that in the cellular automata space.

The Trend program which constructs the Voronoi diagram of a cellular automata space is shown below with a companion Figure 13 to demonstrate the various situations that must be considered during the construction process. This model uses the Moore neighborhood and has only one data field, "value". The symbol '>' is used to represent the four weak rotational symmetric states denoting an expanding wave toward quadrilateral directions. The symbol 'Q' is used to represent the other four weak rotational symmetric states denoting an expanding wave toward diagonal directions. Since there are no suitable symbols in the ASCII character set to represent diagonal movement, the extra

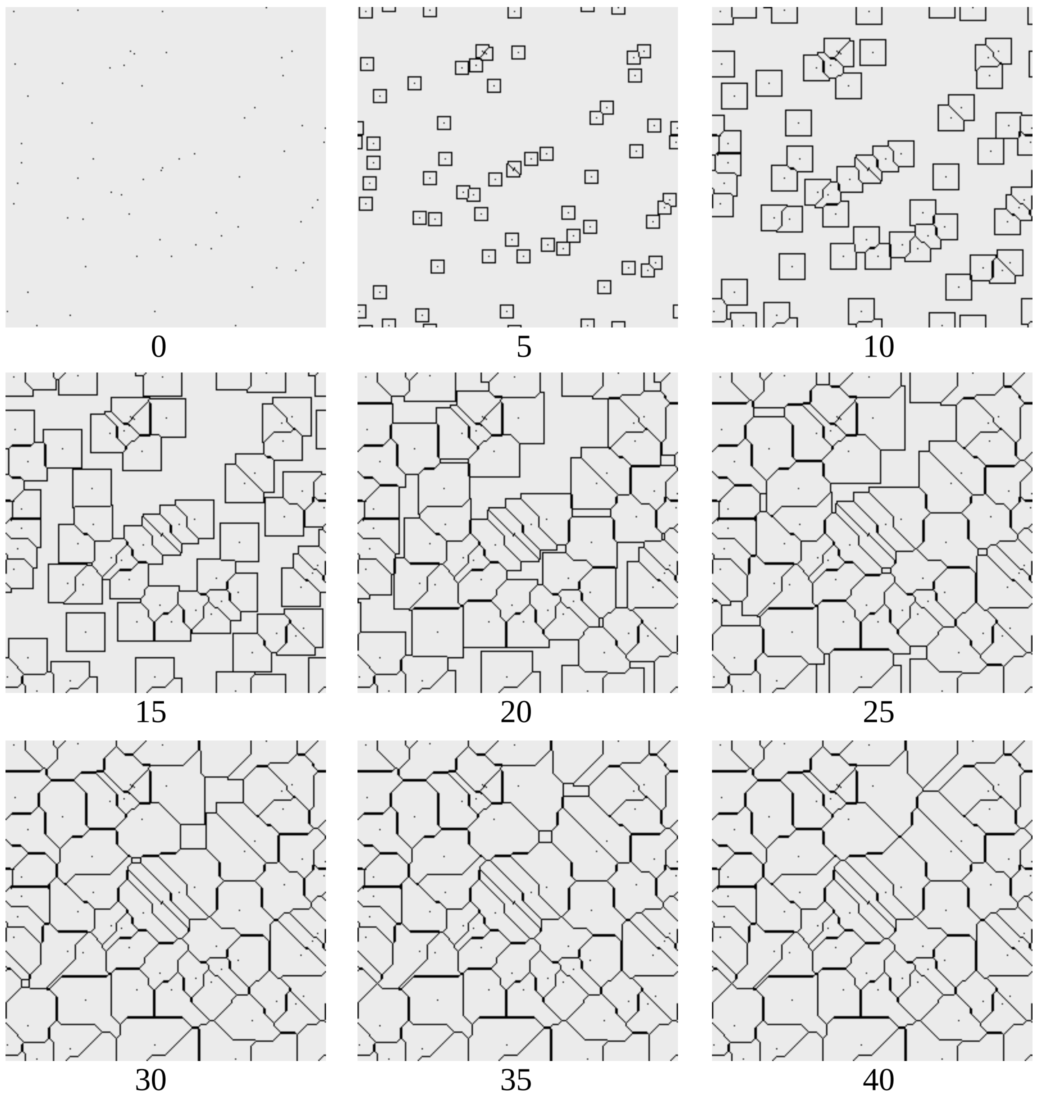


Figure 12: A Voronoi diagram is found by sending expanding square waves from each anchor point. The initial configuration has some random anchor points in a 250 by 250 space. To reduce the size of the figures, each cell is represented by only a single pixel, thus we can only tell the difference of active (black) and quiescent (grey) states. The cellular automata space is wrapped around the four sides in a torus-like shape, as in all Trend models.

“leg” of the letter ’Q’ is used for that purpose. The symbol ’*’ is used to denote anchor points, the symbol ’%’ is used to denote the Voronoi points found, and the symbol ’S’ is used to denote anchor points after epoch 0, so they will not send out additional waves. Only a total of 12 states are used in this rule set.

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

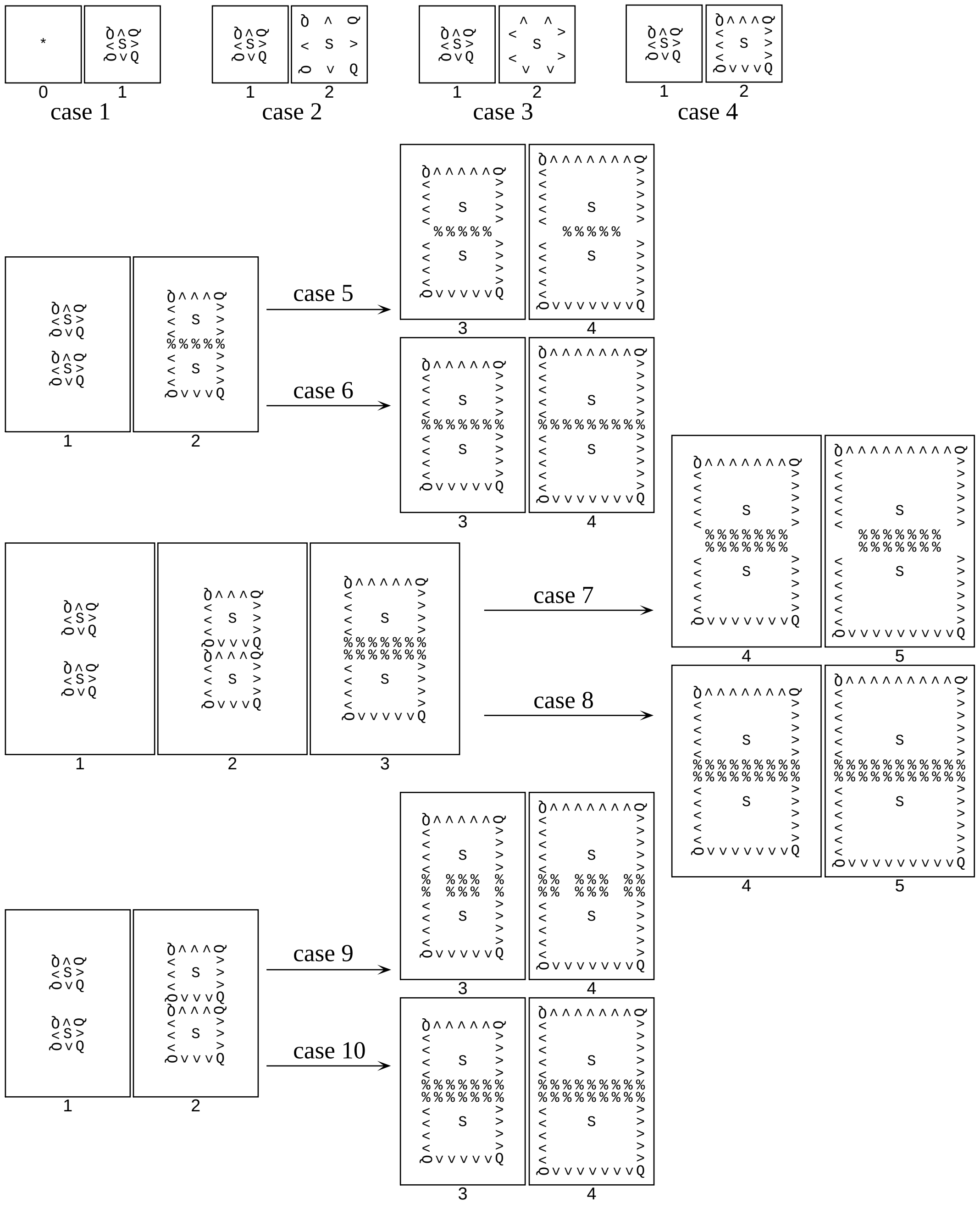


Figure 13: Voronoi rules in action. Ten cases are illustrated in this figure to explain the purpose of some rules in the Voronoi program. Please refer to the comments accompanying the rules for explanation.

```
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 been discussed before in the language
section about the rotated if statement. */

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 to determine if there is any wave moving toward it. The
  variable 'sum' records how many such waves are found. If it is more
  than 1, a collision of waves occurs, and the quiescent cell must be
  converted to 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 the
  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 moving 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 to 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 connect 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 OR’ed condition in
the following rule makes sure the Voronoi segment is still
expanding, as seen in case 6. The other two OR’ed conditions work
similarly, but are for double-line Voronoi segments which are formed
when two expanding waves reach each other at the same time. The
behavior of a double-line Voronoi segment with and without these two
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

    /* An anchor point always change to 'S' after epoch 0, or multiple
    waves will come out from the same point. */

    value='S';

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

    /* A pointer in the expanding wave will return to the quiescent
    state in the next epoch, unless there is a collision, as seen 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, pointers at the four corners of a square wave 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
    with the two extra OR'ed conditions is shown in case 10. */

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

The computational speed for finding cellular automata Voronoi diagram this way is $\mathcal{O}(d)$, where d is the longest dimension of the cellular automata space. In another word, this algorithm is running at the *speed of light* of the cellular automata space[4]. A complexity of $\mathcal{O}(n \log n)$ is usually needed for sequential algorithms to find geometrical Voronoi diagrams. The speed of these algorithms depends on the number of anchor points n, but the cellular automata method does not.

[4]The speed of light is referred to as the fastest speed a signal can travel in a particular cellular automata model. With the Moore neighborhood, it is one cell per epoch.

4.5 Artificial life

As said previously, von Neumann developed cellular automata in order to prove that it is possible to create machines that can compute and reproduce itself [24]. He eventually succeeded in creating a 29-state model that was capable of universal construction and computation. After von Neumann's seminary work, others have taken his approach and created more self-reproducing models. E.F. Codd created a universal cellular automata machine using 8-state cells [8]. Christopher Langton took one of Codd's information preserving "loop" and turned that into a self-replicating loop [15] (cf. Section 4.1.) Although Langton's loop is not capable of universal computation anymore, it can be readily simulated on a personal computer with its small size. John Byl created a smaller version of Langton's loop [5], and Reggia *et al.* created an even simpler self-replicating loop with only six cells [19].

Since von Neumann's original work, focus has been to create ever simpler self-replication structures to facilitate the implementation on a computer. The rationale is that the first self-replicating molecule on earth might not be capable of universal computation. It was when more of them were put together that they formed ever more complex organisms. Therefore, maybe we should start from the very basic self-replicating machinery, then build our way up in model complexity. That is a different approach than von Neumann's, and is the approach take by us in creating the emergent self-replication models where autonomous *monomer* cells can gather together to form the smallest self-replicating structures, and these structures can grow bigger and become more complex [6]. For more information on the subject of cellular automata self-replicating studies, interested readers can read the survey article in [21].

In this last example, we show that Trend allows programmers to develop complex cellular automata rules by dividing their programming effort among separate data fields, i.e., Trend allows a *field-oriented* programming approach. When programmers are working on some features of their models, they only need to concentrate on those fields relevant to these features. They do not need to worry about other fields and functions. This allows several programmers to independently create individual features of a complex rule set, then put them together at the end.

To illustrate, look at a previously published self-replication model that is capable of solving the satisfiability problem (SAT) [7]. In its original design (Figure 14a), self-replicating loops carrying the unknown bits 'A' will gradually turn them into either '0' or '1' depending on whether they stay with the parent or child loops during a replication cycle. This *differential replication* feature, when coupled with additional *monitor* mechanisms in the cellular automata space, removes loops carrying unfit answers from the space, leaving only satisfiable loops to continue their replication. Whatever are left in the cellular automata space after several generations of replication become the satisfying answers to a particular SAT problem encoded in the rules.

Functions of this model can be divided into the following: self-replication of a loop, data differentiation from 'A' to either '0' or '1', selection of unsatisfying bit sequences, and removal of loops carrying unsatisfying bit sequences. These functions more or less are independent of each other, so they can be altered separately. To confirm this, we asked an undergraduate student to change the self-replicating *loop* to a self-replicating *square* that can pack more data into the same size cellular automata space [4]. In the past, most self-replicating structures were designed to be loop-shaped to prevent unwanted interferences between nearby signals. However, when we started using a dedicated data field to control the direction of signal movement, as in the SAT model, the interference problem should have been automatically resolved. Our student successfully creates his self-replicating squares as expected, but it is surprising to see how easily we can incorporate his new rules into our original SAT rules, creating

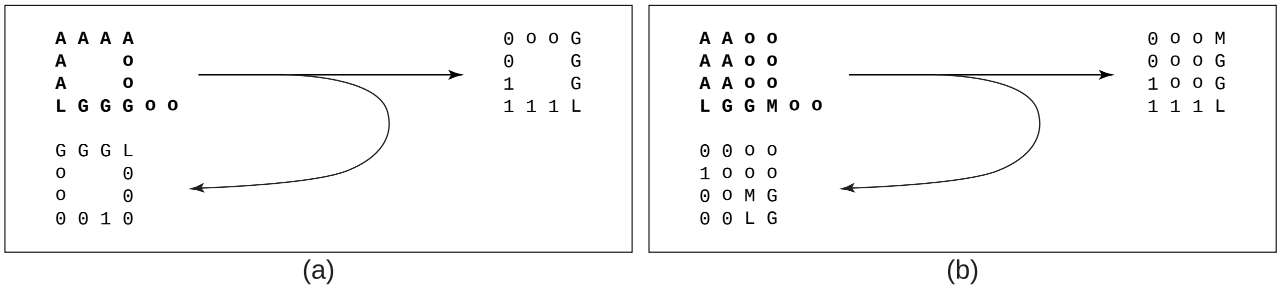


Figure 14: The SAT problem-solving self-replicating structures. (a) The original model. A loop-shaped initial structure shown in **bold** generates two survival offsprings in the cellular automata space after a couple of generations. The SAT bit sequences carried by the offsprings are read off clockwise after the L signal, i.e. 111100 and 000100. They represent the only satisfying answers to a SAT predicate embedded in the rules. (b) The new model created by our student. A square-shaped initial structure shown in **bold** also generates two survival offsprings after a couple of generations. The SAT bit sequences they carry are the same as the loops, but their shape and replication mothod are different.

a new model that can solve the SAT problem *squarely* (Figure 14b).

The undergraduate student does not know how the other parts of the SAT model work (bit evolution, monitor selection, etc.). He focuses only on creating his self-replicating square based on a specific data field. After he has created his rules, we can combine them with the other SAT rules operating on different data fields by changing just a few lines of the code that couple bit sequences with the self-replicating squares, so they can be carried with the squares. The Trend programs for the SAT model are too long to be reproduced here. They are included in the Trend software distribution for anyone who is interested.

5 Discussion

The CAM-6 machine and *Cellular* system described in the introduction are more or less designed for simulating physical systems, notably reaction-diffusion systems [14, 23]. Their design objective is very different than Trend. Trend is designed more for the development of complex cellular automata rule sets. While developing a cellular automata program which supports a desired behavior such as self-replication, we need to constantly observe the current simulation results, modify the program, go back a few time steps, and run the simulation again. This kind of highly interactive *trial-and-error* approach is not easily supported in the other systems.

In addition, the Trend programming language contains many cellular automata specific constructs that are not found in the other languages. The rotated *if* statement is a new invention in the Trend language. It can fully exploit the rotational symmetry of a cellular automata model, and potentially reduce the size of an isotropic cellular automata program to only 1/4 its size without using the rotated *if*. Trend has been carefully designed to support cellular automata rule sets which are both isotropic and conflict free. Preventive features, like conflict catching, dead loop detecting, and unsolved reporting, are also unique in Trend.

The Trend language presents a standard C like syntax which is familiar to most users. In Trend, all neighbor and field names automatically become semi-reserve words in the language. These reserve words are given special treatment by the Trend compiler and are quite handy in describing cellular automata rules at a higher conceptual level. Trend also allows the definition of symbolic literals. Such symbols are used both to display the cellular automata space on screen and to represent individual states in a Trend program. Therefore, the direct correspondence between what the user writes in his code and what he sees on the screen greatly simplifies efforts to recognize and correct programming errors. 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 the other cellular automata languages.

It is worth mentioning, however, that the *Cellang* language does have some interesting features which are not available in Trend. These include the special `time` and `random` system variables, support of multi-dimensional cellular automata, and the object-oriented *agents* programming mechanism. The system defined variables and the agents mechanism do not belong to the standard definition of homogeneous cellular automata. They are extensions to the standard model. When Trend was being developed, we had a strong desire to follow the standard of a homogeneous, self-contained, time and position indifferent cellular automata model. Therefore, those extra features of *Cellular* were not pursued in this work.

The one-pass compiler of Trend currently imposes a no-forward-reference restriction on Trend programs. Recursive calls are allowed, but indirect recursive calls with two or more functions involved are not possible at the moment. Although this does not consist a major limitation to the power of the Trend language, it may be inconvenient for some occasions. This restriction may be addressed in the next revision if we find enough actual applications that call for this feature.

Currently, there are two different Trend implementations, one for Unix-like systems and one for the Java Virtual Machine. These two implementations are fully compatible at the input/output level and share the same cellular automata, template and program files. However, internally, there are some difference between them. On the Java Virtual Machine, Trend generates the Java Virtual Machine code. On Unix systems, it generates our own set of virtual code. The benefit of using virtual machine code rather than machine code is certainly the portability. It allows the Trend systems to be ported easily to the other platforms. On the downside, running virtual machine code requires another layer of interpretation during runtime, which can slow a simulation down. Although this speed penalty is not a major issue for many cellular automata models where the caching mechanism of Trend can reduce or eliminate runtime evaluations dramatically, it may become an issue with certain models and when the simulated space size is increased. If the speed of Trend becomes a bottleneck, the code generation module of the Trend compiler can be modified to generate native machine code. It is worth noting that the Trend implementation on Java runs roughly at 1/4 the speed of the Unix version in the same machine. This slowdown is probably due to the overhead of implementing the more powerful and general features of the Java Virtual Machine. However, in exchange for the speed, the Java implementation of Trend (called jTrend) is highly portable. It can run on Unix systems, PC and Mac without any modification. We believe the Java Version of Trend will make it more widely available to new users.

The Trend system is a general purpose cellular automata simulation environment. For any general purpose system, its potential applications are often beyond the imagination of the system designer. Even seasoned users of Trend often discover new usages of the language and system. It is possible that some Trend programmers can discover new applications of it in the future. A library of useful Trend applications can be maintained to avoid re-engineering the same features and allow them to be shared by the other Trend users.

Acknowledgement

This work was supported in part by NASA grant NAGW-2805 to J. Reggia., by an Iowa State faculty startup support to H.-H. Chou, and by NSF SEGR grant IIS-0001542 to H.-H. Chou.

References

[1] S. Bandini, G. Mauri, and R. Serra. Cellular automata: from a theoretical parallel computational model to its application to complex systems. *Parallel Computing*, 27:539–553, 2001.

[2] M.C. Boerlijst and P. Hogeweg. Spiral wave structure in pre-biotic evolution. *Physica D*, 48:17–28, 1991.

[3] 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.

[4] Michael Burman and Hui-Hsien Chou. A new SAT problem-solving self-replicating model. Available with the Trend software distribution, 2001.

[5] John Byl. Self-reproduction in small cellular automata. *Physica D*, 34:295–299, 1989.

[6] Hui-Hsien Chou and James A. Reggia. Emergence of self-replicating structures in a cellular automata space. *Physica D*, 110:252–276, 1997.

[7] Hui-Hsien Chou and James A. Reggia. Problem solving during artificial selection of self-replicating loops. *Physica D*, 115:293–312, 1998.

[8] E.F. Codd. *Cellular Automata*. Academic Press, New York, 1968.

[9] J. Dana Eckart. A cellular automata simulation system: Version 2.0. *ACM SIGPLAN Notices*, 27(8), August 1992.

[10] J. Dana Eckart. A *cellular* automata simulation system, 1995. Published online and available as of 7/7/2001 at <http://www.cs.runet.edu/˜dana/ca/tutorial.ps>.

[11] J. Dana Eckart. The Cellular automata simulation system. Available on the Internet as of 6/26/2001, 2001. http://www.cs.runet.edu/˜dana/ca/cellular.html.

[12] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[13] M. Gerhardt, H. Schuster, and J. Tyson. A cellular automaton model of excitable media including curvature and dispersion. *Science*, 247:1563–1566, 1990.

[14] J. P. Keener and J. J. Tyson. Spiral waves in the Belousov-Zhabotinsky reaction. *Physica D*, 307, 1986.

[15] Christopher G. Langton. Self-reproduction in cellular automata. *Physica D*, 10:135–144, 1984.

[16] Christopher G. Langton. Artificial life. In Christopher G. Langton, editor, *Artificial Life*, pages 1–47. Addison-Wesley, Reading, Massachusetts, 1989.

[17] 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, Reading, Massachusetts, 1991. Addison-Wesley.

[18] K. Preston and M. Duff. *Modern Cellular Automata*. Plenum Press, New York, 1984.

[19] 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, 1993.

[20] 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.

[21] James A. Reggia, Hui-Hsien Chou, and Jason Lohn. Cellular automata models of self-replicating systems. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 47. Academic Press, San Diego, California, 1998.

[22] The ISU Complex Computation Lab. Trend download and information page. Available on the Internet, 2001. http://www.complex.iastate.edu.

[23] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines*. MIT Press, Cambridge, Massachusetts, 1987.

[24] John von Neumann. *The Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966.

[25] Stephen Wolfram. *Theory and Applications of Cellular Automata*. World Scientific, Singapore, 1986.

[26] Stephen Wolfram. *Cellular Automata and Complexity*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[27] Andrew Wuensche and Mike Lesser. *The Global Dynamics of Cellular Automata*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

Appendix: A concise Trend language grammar

The following Trend grammar is converted from the actual grammar we used to implement Trend using automatic parser generator tools. Some conversions are necessary in order to make the grammar more concise and readable. Note that a context-free grammar alone cannot capture all the semantic requirements of a language. The Trend language semantics have been discussed in Section 3.

This grammar uses extended BNF notations. Extended BNF notations are similar to regular expressions, e.g. parentheses '(' and ')' group subexpressions together, '|' connects alternative choices, '?' denotes zero or one occurrence, '*' denotes zero or more recurrences, and '+' denotes one or more recurrences. Each *production* of the grammar consists of a nonterminal name, a colon, a list of its possible derivations separated by '|', and a terminating ';'. Do not confuse the BNF grammar symbols above with language-specific symbols. Symbols or words inside single quotes (e.g. 'default', '+', or '&&') represent terminal tokens of the Trend language. These terminal tokens are reserve words or symbols of the language that can be used *as is* in a Trend program (without the quotes, of course). CAPITALIZED words in this grammar are also terminal tokens, but their textual representations are user-defined. The following is a brief outline of user-defined terminal tokens and their meanings:

- NEW_VARIABLE denotes a variable that must begin with a letter but can include letters, digits or underscores ('_') in its name.
- VALUE denotes an integer value either in numeric forms (e.g. 1, 2, or 33) or symbolic forms (e.g. '>', 'O', or '1').
- RVALUE denotes a rotatable literal representing a weak rotational state defined in a cellular automata template (e.g. '>,1' or 'Q,3'). RVALUEs are special in Trend because they are automatically rotated by the compiler into the other three counterpart states within a rotated *if* statement. Note that a symbol (e.g. '>') can denote either a VALUE or a RVALUE, depending on how that symbol is defined in a template.
- INTEGER denotes a variable or array for storing integer values.
- FIELDNAME denotes a field, a field variable or a field array.
- NEIGHBORNAME denotes a neighbor, a neighbor variable or a neighbor array. Similar to RVALUE, NEIGHBORNAME is automatically rotated inside a rotated *if* statement by the Trend compiler if it is a template-defined neighbor name.
- PROCEDURE denotes a procedure that does not return a value.
- FUNCINTEGER denotes a function that returns an integer value.
- FUNCNEIGHBOR denotes a function that returns a neighbor index.
- FUNCFIELD denotes a function that returns a field index.

Everything else in lower case letters are grammar production names that are defined within the grammar itself. The starting production for this grammar is `program`.

```
program: ( declaration | statement | 'default' statement )+ ;
declaration: variable_decl ';' | function_decl ;
variable_decl: ( 'int' | 'nbr' | 'fld' ) name_list ;
function_decl: ( 'int' | 'nbr' | 'fld' | 'void' ) NEW_VARIABLE
        '(' (argument_list)? ')' '{' function_stmts '}' ;
argument_list: variable_decl ( ';' variable_decl )* ;
name_list: name ( ',' name )* ;
name:  NEW_VARIABLE | NEW_VARIABLE '=' i_value
        | NEW_VARIABLE '[' VALUE ']'
        | NEW_VARIABLE '[' ']' '=' '{' i_list '}' ;
i_list: i_value ( ',' i_value )* ;
i_value: VALUE | RVALUE | ':' FIELDNAME | NEIGHBORNAME ':' ;
function_stmts: ( variable_decl ';' | statement )+ ;
statements: '{' (statement)+ '}' | statement ;
```

```
statement: field_assignment ';' | var_assignment ';' | condition
        | PROCEDURE '(' (par_list)? ')' ';' | function_return ';'
        | 'break' ';' | loop_head statements ;
field_assignment: field '=' expr | field '+' '+' | field '-' '-' ;
var_assignment: integer '=' expr | integer '+' '+' | integer '-' '-'
        | ':' field '=' ':' field | neighbor ':' = neighbor ':' ;
condition: if_head | if_head 'else' statements ;
if_head: normal_if | 'rot' normal_if | 'rot' '(' boolean ')' normal_if ;
normal_if: 'if' '(' boolean ')' statements ;
function_return: 'return' | 'return' expr | 'return' neighbor ':'
        | 'return' ':' field ;
loop_head: 'over' 'each' ('other')? NEIGHBORNAME ':'
        | 'while' '(' boolean ')'
        | 'for' '(' (assignment_list)? ';' boolean ';'
          (assignment_list)? ')' ;
assignment_list: var_assignment ( ',' var_assignment )* ;
boolean: conjunction ( '||' conjunction )* ;
conjunction: disjunction ( '&&' disjunction)* ;
disjunction: '(' boolean '||' conjunction ')' | comparison ;
comparison: expr | neighbor ':' ( '=' '=' | '!' '=') neighbor ':'
        | ':' field ( '=' '=' | '!' '=' ) ':' field ;
        | expr ( '>' | '>' '=' | '=' '=' | '<' '=' | '<' | '!' '=' ) expr
par_list:  parameter ( ',' parameter )* ;
parameter: expr | ':' field | neighbor ':' ;
integer: INTEGER | INTEGER '[' expr ']' ;
field: FUNCFIELD '(' (par_list)? ')' | FIELDNAME
        | FIELDNAME '[' expr ']' ;
neighbor: FUNCNEIGHBOR '(' (par_list)? ')' | NEIGHBORNAME
        | NEIGHBORNAME '[' expr ']' ;
expr: factor ( ( '+' | '-') factor )* ;
factor: literal ( ( '*' | '/' | '%' | '&' | '|' | '^' ) literal )* ;
literal: neighbor ':' field | field | integer | VALUE | RVALUE
        | INTEGER '.' 'length' | FIELDNAME '.' 'length'
        | NEIGHBORNAME '.' 'length'
        | FUNCINTEGER '(' (par_list)? ')' | '(' expr ')' ;
```