References

[1] Allinson, N M, and M J Sales, "CART — A cellular automata research tool", *Microprocessors and Microsystems*, **16**:8 (1992), 403–415.

[2] Bohgosian, Bruce M., "A Cellular Automata Simulation of Two-Phase Flow on the CM-2 Connection Machine Computer", *Proceedings of Supercomputing '88, Volume II: Science and Applications*, IEEE Computer Science Press, (1988), 34–44.

[3] Chen, Shiyi, Hudong Chen and Gary D. Doolen, "How the Lattice Gas Model for the Navier-Stokes Equation Improves When a New Speed is Added", *Complex Systems*, **3** (1989), 243–251.

[4] Dewdney, A. K., "The Cellular Automata Programs That Create Wireworld, Rugworld and Other Diversions", *Scientific American*, **262**:1 (January 1990), 146–149.

[5] Dewdney, A. K., "A Cellular Universe of Debris, Droplets, Defects and Demons", *Scientific American*, **261**:2 (August 1989), 102–105.

[6] Dewdney, A. K., "The Hodgepodge Machine Makes Waves", *Scientific American*, **259**:2 (August 1988), 104–107.

[7] Gardner, Martin, "The Fantastic Combinations of John Conway's New Solitaire Game of 'Life',", *Scientific American*, **223**:4 (April 1970), 120–123.

[8] Harbison, Samuel P. and Guy L. Steele Jr., *C: A Reference Manual*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[9] Langton, Chris and Dave Hiebeler, *Cellsim*, available by anonymous ftp from isy.liu.se, Version 2.5.

[10] Lim, Hwa A., "Lattice Gas Automata of Fluid Dynamics for Unsteady Flow", *Complex Systems*, **2** (1988), 45–58.

[11] von Neumann, John, *Theory of Self–Reproducing Automata*, edited and completed by A. Burks, University of Illinois Press, Champaign, IL (1966).

[12] Palmer, Ian J., *Scamper*, available by anonymous ftp from ftp.uu.net.

[13] Sieburg, Hans B. and Oliver K. Clay, "The Cellular Device Machine Development System for Modeling Biology on the Computer", *Complex Systems*, **5** (1991), 575–601.

[14] Stephenson, I., *Creature Processing: An Alternative Cellular Architecture*, Technical Report ASEG92.04, Department of Electronics, University of York.

[15] Teixeira, Christopher M., *Continuum Limit of Lattice Gas Fluid Dynamics*, PhD Thesis, M.I.T. Department of Nuclear Engineering, Sep 1992.

[16] Toffoli, Tommaso and Norman Margolus, *Cellular Automata Machines: A New Environment for Modeling*, The MIT Press, Cambridge, Massachusetts, 1987.

[17] Ulam, Stanislaw, "Random Processes and Transformations", *Proceedings of the International Congress on Mathematics*, 1950, Vol. 2 (1952), pp. 264–275.

```
0
[32, 20] = 1, 1
[32, 40] = 1, 2
[20, 32] = 1, 3
[40, 32] = 1, 4
[32, 32] =  , 2
[10, 10] =  , 1, 3
```

Figure 12: An example input/output for a *Cellang* program with agents

value of one of the agents. Assuming that there is only one agent associated with the cell, `ball` will be the value of that agent.

By default, agents cease to exist (and thus to be associated with a given cell) at the end of every time step. In order to have agents reappear during the next time step, they must be explicitly propagated by the use of an agent statement. The agent statement is conditional in the same manner as the assignment statement, and indicates which cell (appearing on the right of the `->`) the agent on the left should be associated with during the next time step. New agent values can be created (e.g. `agent(1) -> [0, 1]`) or an existing value can be reassociated (e.g. `ball -> [0, 1] when ball.direction = 1`). Thus to have two or more balls annihilate one another upon colliding, simply do *not* recreate/reassociate them.

The input and output of agent field values is an extension of that for cell field values. Figure 12 shows a sample input time block for use with the pinball program. Cell field values are always given first, with any fields beyond that specifying zero (0) or more agents. Thus the first four cell locations have bumpers with a single agent at each of those cells. Agents are specified in field groups, with all of the fields specified in the cell declaration being used before another agent is created and associated with the indicated cell. As with cell field values, agent field values can be skipped by simply giving the comma (,) with no value. The next to the last line of Figure 12 specifies no change in the `bumper` field, but does place a single east bound agent. Alternatively, the last line shows no change in the `bumper` field, but places two agents at the cell, one traveling north and one going south. Output of agents is done in a similar manner. On output, however, only the agent field values of existing agents will be output. Thus, when an agent is no longer associated with a cell *only* its absence from the output is an indicator of this change.

One final note concerning agents: programs should make no assumptions concerning the order in which agents are associated with a cell. Thus, multiple `forall` statements which iterate through the agents associated with a cell might present the agents in a different order for the same cell during the same time step. Likewise, the order in which agents and input and output does not necessarily reflecting the ordering that may be seen with a `forall` statement.

6 Current Status and Availability

The current implementation of the *Cellang* compiler, as a part of the *Cellular* system, generates code for both uni-processor and shared-memory multi-processor systems. The entire system is written in C and the compiler produces C as an intermediate code, making the system highly portable.

The *Cellular* system is designed to be compiled under both the Unix and Windows NT operating systems and currently supports the viewing of automata using X-Windows (Unix), IRIS Graphics Library (Unix), or OpenGL (Unix and Windows NT). Instructions on the where to get a copy of the system can be obtained by contacting the author.

```
2 dimensions of
        # A value of 1 indicates the presence of a bumper.
        bumper of 0..1
agent of
        # Indicates the direction of travel for the agent.
        # (1 = north, 2 = east, 3 = south, 4 = west)
        direction of 1..4
end

# Count the number of agents associated with this cell.
count := 0
forall a:agent
        count := count + 1
        ball := a
end

# Only single agents do anything.
if count = 1 then
        if cell.bumper then
                # The agent has collided with a bumper and divides.
                if ball.direction = 1 then
                     agent(2) -> [1, 0]
                     agent(4) -> [-1, 0]
                elsif ball.direction = 2 then
                     agent(1) -> [0, 1]
                     agent(3) -> [0, -1]
                elsif ball.direction = 3 then
                     agent(2) -> [1, 0]
                     agent(4) -> [-1, 0]
                else
                     agent(1) -> [0, 1]
                     agent(3) -> [0, -1]
                end
        else
                # The agent continues to travel.
                ball -> [0, 1]  when ball.direction = 1
                     -> [1, 0]  when ball.direction = 2
                     -> [0, -1] when ball.direction = 3
                     -> [-1, 0] when ball.direction = 4
        end
end
```

Figure 11: A "pinball" simulation using agents

they encounter a bumper. Each ball in the simulation has its own direction of travel, which it maintains until it encounters either a bumper or another ball. If two or more balls meet, they annihilate one another.

Several of the language features already discussed support the use of agents. The cell declaration can consist of two parts. The first specifies `bumper` as a classic cellular automata field, while the second indicates that the field `direction` is a part of every agent. A cell can have zero (0) or more agents associated with it at any time. In addition, the `forall` statement supports the iteration through all of the agents associated with the current cell by specifying `agent` as the range of iteration. The assignment statement can also be used in conjunction with agents. In the case of the pinball program, the first `forall` statement is counting the number of agents associated with this cell. The implicitly declared variable `ball` is used to remember the

```
2 dimensions of
        const which of 0..1
        count of 0..6

        # The presence (1) of a particle traveling in the given direction.
        particle[] for 6 of 0..1
end

# Hexagonal neighborhood.
#
neighbor[] for 6
        := [-1, 1], [0, 1], [1, 0], [0, -1], [-1, -1], [-1, 0] when cell.which
        := [0, 1], [1, 1], [1, 0], [1, -1], [0, -1], [-1, 0] otherwise

# Find out how many particles are incident on this cell.
#
count := 0
first := -1
forall i
        if neighbor[i].particle[i+%3] then
                count := count + 1
                if first < 0 then first := i
                else last  := i
                end
        end
end
cell.count := count

if count = 2 & first+3 = last then
        # Rotate Clockwise or Counter-Clockwise
        #
        if random%2 then
                forall i cell.particle[i+%1] := neighbor[i+%3].particle[i] end
        else
                forall i cell.particle[i+%1] := neighbor[i+%3].particle[i] end
        end
else
        #
        # The incoming particles pass through unabated.
        #
        forall i cell.particle[i] := neighbor[i+%3].particle[i] end
end
```

Figure 10: Lattice gas automata using a hexagonal neighborhood

occurs during alternate time steps. The first phase determines which cell to move to, while the second phase actually performs the operation. Not only is programming complicated by this process, but the basic movement algorithm lies buried deep within the resulting code and is thus not easily accessible to the reader. Agents allow movement to be specified more clearly, concisely, and easily than is possible by using classical cellular automata techniques alone. The agents of *Cellang* are an adaptation of those found in the *Creatures*[14] system.

The program in Figure 11 shows a *Cellang* program for an odd sort of pinball game. The pinball game consists of balls and bumpers. The balls move around in the cell universe and split into two balls whenever

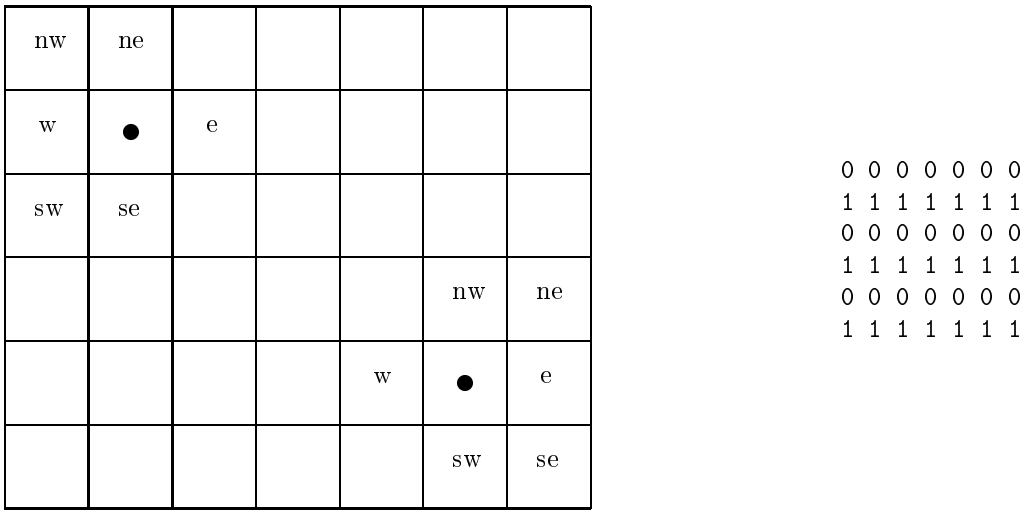


Figure 9: A hexagonal neighborhood and the pattern of `which` field values

always have a lower bound of zero). Finally, the modulo arithmetic operations +% and –% can be used in conjunction with index variables to perform both addition and subtraction operations modulo the range of that index variable.

The `forall` statement is the preferred mechanism for indexing all of the values of an array.[7] A `forall` index variable can act as an index into one or more arrays that appear in the statement(s) of the body of the `forall`. An index variable can be indicated either simply as a variable name or as the variable name followed by a colon (:) and a range of values that the variable should have during consecutive iterations through the loop. When the range of values for an index variable has been specified by the programmer, the size of the range must be equal to the number of elements for any array for which it is used as an index. Furthermore, the successive values of the index correspond to the successive elements of the array it indexes (beginning with the first named variable given in its declaration) regardless of the value of the bounds given for the range of the index variable. In this way, different ranges can be specified to best suit the needs of the programmer for that particular loop. If no range was specified, then its lower bound is taken as 0 and the size of the range is inferred from its use in the statements contained in the loop body.

It is also necessary to provide a mechanism to look at values in the array at some offset from the the current value of the index. This is accomplished by using the +% and –% indicators which "circular shift" the array in the desired direction from position the current index value would otherwise indicate. More formally, a circular shift is an addition/subtraction modulo the size of the index variable range. Thus if the range of the index variable `i` is `0..5` and the value is `4`, then the result of `i+%3` is `1`. The modulo operations are extremely useful in combination with arrays.

Finally, the predefined identifier `random` provides a new uniformly distributed random number each time it is used. This mechanism could not be used in the gas diffusion example since it was necessary to guarantee that each cell in the block of the Margolus neighborhood had the same random value. Its use here, however, alleviates the need for a separate cell field for maintaining a random value.

5.3 Agents

Although it is possible to move a value group from one cell to another cell using classical cellular automata techniques, it is an error plagued process. These techniques invariably consist of two phases, each of which

[7] If only some values are desired, their individual names or an absolute reference (e.g. `array[3]`) should be used. Note that absolute references are based on a lower bound of zero (0).

```
2 dimensions of
        const which of 0..3
        rand  of 0..1
        gas   of 0..1
end

neighbor[] for 4

        := cell, [1, 0], [0, -1], [1, -1]
                when (cell.which=0 & time%2=1) | (cell.which=3 & time%2=0)

        := cell, [0, -1], [-1, 0], [-1, -1]
                when (cell.which=1 & time%2=1) | (cell.which=2 & time%2=0)

        := cell, [-1, 0], [0, 1], [-1, 1]
                when (cell.which=3 & time%2=1) | (cell.which=0 & time%2=0)

        := cell, [0, 1], [1, 0], [1, 1]
                otherwise

# Calculate the sum of the random fields in the neighborhood.
#
rand := cell.rand + neighbor[1].rand + neighbor[2].rand + opp.rand

cell.gas := neighbor[2].gas when rand%2 = 1  # Clockwise rotation
         := neighbor[1].gas otherwise        # Counter-clockwise rotation

# Stir the random values using a von neumann neighborhood.
#
cell.rand := ([1, 0].rand + [0, 1].rand + [-1, 0].rand + [0, -1].rand
                + cell.rand)%2
```

Figure 8: Gas diffusion using a Margolus neighborhood

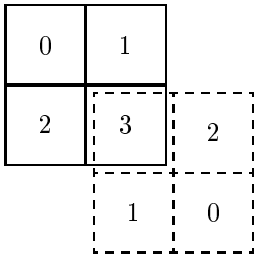
use of the array variable `neighbor`. The first lexical occurrence of an array must have a `for` clause indicating the number of elements that the array contains, with indexing of the array always starting at zero (0). The simplest way to assign initial values to an array is by using an empty index ([ ]) and giving the proper number of values, separated by commas, to assign to the corresponding array elements. If only one value is given, then all elements of the array are assigned that value.[6] In this program the elements of the array were always referenced using their individual names.

5.2 Hexagonal Neighborhood

Hexagonal neighborhoods exhibit properties desirable in the simulation of many physical systems. Fortunately, they are quite easy to build in *Cellang*. Figure 9 depicts a method which uses alternating columns of Moore neighborhood subsets. The value of the `which` field determines which Moore subset to use.[1]

The lattice gas automata program in Figure 10 demonstrates several additional features of arrays and `forall` statements. First, array can also appear as fields of cells (i.e. `particle`). The second is that the index variables of `forall` statements can be used to index the different elements of the array, and that the range of an index variable used in this manner can be automatically inferred by its use (inferred ranges

[6] This is very useful when working with large arrays (e.g. `array[] for 256 := 0`).




```
0 1 0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3 2 3
0 1 0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3 2 3
```

Figure 7: Margolus neighborhood and the pattern of `which` field values

program. Filter programs must be compiled separately, and are linked in with the *Cellang* program when the `-F` option is used. Filter programs must contain a definition for the function `do_filter`, which is called at the end of each time step. In addition, filter programs can use functions to get the number of dimensions, the size of each dimension, and the value of a particular cell field. Filter programs are free to report their results in any manner they choose. An example of a filter program, written in the C programming language[8], appears in Figure 6. This filter prints out the value of the second field of the cell at location (33, 44) every fourth time step.

5 Advanced Examples and Additional Language Features

This section examines two example programs that demonstrate techniques for generating the Margolus and hexagonal neighborhoods. The Margolus neighborhood will be used in the simulation of the diffusion of a gas,[5] while the hexagonal neighborhood is used to implement a lattice gas automata.

5.1 Margolus Neighborhood

Gas diffusion can be simulated by implementing a random walk for each particle/molecule of gas. The random walk can be implemented by using non-overlapping blocks of four cells (two on a side) which are offset at alternate times (Margolus neighborhood) and having each block of cells randomly rotate their gas particles either clockwise or counter-clockwise.

To better understand the Margolus neighborhood, consider Figure 7. The four cell block with a solid outline, and the other with the dashed outline, would be used at alternate times. At any particular time, however, the universe of cells is completely covered by a non-overlapping set of blocks. To implement a Margolus neighborhood in *Cellang* requires use of the `time` variable and some way of distinguishing between different blocks of 4 cells. The latter is accomplished by introducing a field, `which`, that contains a value, 0 to 3, indicating the cells relative position in the block. A portion of the cell universe `which` values are also shown in Figure 7. The `which` field has been declared as a constant, which means that its value can only bet set or changed by reading its value from the input. Constant fields use slightly less memory and in most cases should also improve the execution speed of the automata.

Now that the four cell block can be identified, a random number, common to each cell in the block, must be generated. This is done by having each cell maintain a two value (0 or 1) `rand` field. By using `time` and the value of `which` a random number, "shared" by each cell of the block, can be calculated. This random number is used to determine whether the particles of gas, denoted by the value of the `gas` fields, are rotated clockwise or counter-clockwise. To maintain an ever changing `rand` field for each cell, a new value is calculated using the Von Neumann neighborhood.

The *Cellang* program that implements gas diffusion is shown in Figure 8. Note the implicit declaration and

[5] The program is modeled closely on the version described by [16] on page 156.

4 Compiling & Viewing

Suppose the *Cellang* program for the game of Life that appears in Figure 1 is in the file `life`. A compiled version of the code is obtained by doing "`cellc life`" which places the object code into a file called `avcam.out`. The object code is executed and displayed by the `avcam` abstract machine via:

```
avcam -c avcam.out -r times -x11 -map life.colors < life.data
```

where `life.data` is a data file containing the initial configuration of live cells and `life.colors` contains a suitable color map. An optional parameter, `-f`, is used to indicate which field (starting from 1 for the first field) to use as the value to display for the cells. The `-r` option indicates a file which contains information about which generations to report cell values for.

If the `-r` option is not given then the cell values which have changed will be reported for each generation. A sample time file appears in Figure 5. A time file is a list of time indications given one per line. A number indicates a time when the values should be reported, while a "+" entry causes these values to be produced at the indicated time intervals. Finally a "!" entry determines the time at which the last set of values should be produced and the simulation stops. Note that time files need not contain a "!" entry. In the example, The first set of values (after time 0) is produced at time 50, then at each time thereafter until time 100 at which point no values are reported until time 500 and the simulation stops.

Alternatively, `avcam` can just output the cell values which can be filtered and/or viewed with `cellview`. An example of this style of use is:

```
cat  life.data | avcam -r times -o |
                          cellview -x11 -map life.colors -dim 0..63,0..63
```

Where the `-o` option indicates that cell values be written to the standard output (as opposed to being viewed directly by using one of the `-x11`, `-iris`, or `-ogl` options). The "-c avcam.out" has been left out, since the the codefile is assumed to be "avcam.out" by default. Note that now, however, the dimensions of the cell universe must be known. By convention, the cell universe begins numbering at 0 for each dimension. The size and number of dimensions is implementation dependent (and both can be configured at compile time in the current release of the software).

The `cellc` compiler can also be used to generate a trace of the computation on a particular cell when executed by `avcam`. For example, by giving the option "`-trace 20,30`", a report of each stage of the computation, during each time step, will be printed for the cell located at position (20, 30).

A filter program can also be compiled into the cellular automata using the `-F` option to the `cellc` compiler. Filter programs provide a faster and more efficient way to generate composite data using from *Cellang*

```
#include <stdio.h>
extern int num_dims(void);
extern int dim_size(int dim);
extern long int get_field(int field, int dim_0_index, int dim_1_index);
extern unsigned long int get_time(void);

int do_filter(void) {
        if (get_time()%4 == 0) printf("%ld\n", get_field(2, 33, 44));
        return 0;  /* Returning a non-zero causes the CA to quit. */
}
```

Figure 6: Simple filter program written in C.

```
0
[3, 3] = 1
100
[24, 56] = 0
[50, 50] = 5, 6
[2, 32] = , , 3
```

Figure 4: An example input/output for a *Cellang* program

```
50
+1
100
500!
```

Figure 5: An example time file for reporting cell values

cell at absolute location 24, 56 has its first field set to 0, the cell at absolute location 50, 50 has its first field set to 5 and it second field set to 6, and the cell at location 2, 32 has its third field set to 3. This cell value input process continues for as many time blocks as are in the input. The output of a *Cellang* program has the same form and can thus appear as input to another automata. If each cell has several values which must be set, then the multiple values are separated by commas and are assigned in the order in which they were declared in the associated *Cellang* program. When only some of the possible field values are given, then the values are assigned in order until no more are available, the unspecified cell fields retaining their previous values.

3 The Viewer

The viewer, which is used to examine the cell values in a graphic manner, is independent of the *Cellang* language and compiler. The input to the viewer is identical in form to the output (and thus the input) of *Cellang* programs. The viewer allows the viewing of a single field along contiguous portions of any two dimensions. The values can be displayed as colored squares (via either X-Windows, the IRIS Graphics Library library, or OpenGL under Unix, or Windows NT). Limited debugging aids (available only with the Unix viewing options) also allow the numeric values of a neighborhood of cells (as opposed to the color associated with this value) and the values of each field of a particular cell to be viewed.

The idea of separating the language (and thus the cellular automata simulator) and the viewer allows greater flexibility both in the viewing of data and in the creation of cellular automata. There is no restriction that the input to the viewer be generated by a *Cellang* program, and it is possible that programs of other sorts will be developed to examine, gather and report statistics concerning the results of *Cellang* programs. "Valve" filters could be inserted between the output generated by a *Cellang* program and the input to the viewer. Such a valve program could allow the presentation speed to be tailored to the researcher's needs. Since the output of *Cellang* programs can be sent to the standard output, it can be piped directly into the viewer with no need for intermediate storage files. The use of a tpipe filter would enable all or part of the produced cell values to be captured in a file during viewing.

```
const max_value := 255

2 dimensions of
        x, y of 0..max_value
end

cell.x := ([1, 0].y + [0, 1].y + [-1, 0].y + [0, -1].y) % (max_value + 1)
                when time % 2

cell.y := ([-1, 1].x + [1, 1].x + [-1, -1].x + [1, -1].x) % (max_value + 1)
                when (time + 1) % 2
```

Figure 3: Multi-field cells with time dependent neighborhoods.

doesn't take effect until the beginning of the next cycle. Notice that the standard Moore neighborhood used by the game of Life must be given by relative indexing.[4] While relative indexing may seem cumbersome, it provides great flexibility. The only practical restriction on relative indices is that they be integer constants or `forall` statement index variables (discussed below).

Because there are no language specific limits on the size of relative indices, it is possible to explore a wide range of neighborhoods. Neighborhoods which use field values n cells away are possible for any value of integer value n. In addition, there is no restriction that neighborhoods be symmetrical or have any other special topological properties. This is exemplified by the program in Figure 2, which uses `forall` statements to successively reference the neighboring cells. The single index variable takes on the successive values given in the range of values specified after the colon (:). The lower and upper bound of the range must be integer constants.

The predefined variable, `time`, indicates the number of computational time steps that have been executed thus far. In order to prevent corruption of its value, `time` can only appear within expressions, where it can be used to influence computation. The initial value of `time` is 0 and it is incremented by 1 each time all the cells of the universe have been updated. This feature allows neighborhoods which are time dependent. Similarly it is also possible to have cell field values change with dependence upon time. The time dependent neighborhoods can be used to represent both hybrid neighborhoods or situations where the physical system behaves according to different rules/neighborhoods as it ages.

Cellang programs may also contain more than a single field per cell. Figure 3 shows a program with 2 fields per cell which also uses a time dependent neighborhood. Note that the dot notation familiar from many other languages for use with records/structures is used here to determine which of several cell fields is being denoted. If, on the other hand, the field name is not present, then all fields of the cell are used in assignment and for tests of equality and inequality. Figure 3 also demonstrates the use of simple constants (`max_value`). Simple constants can be used anywhere a number can be used: to specify the number of dimensions, the bounds of a range, as part of a relative index, or as part of an arithmetic expression. As the name suggests, however, the value of a constant cannot be changed.

Finally, both the input and output of *Cellang* programs allow the user to manipulate cellular automata data in ways that many systems do not permit. Both input and output are of identical form, thus allowing automata to be chained together. Both input and output are given as a sequence of time blocks. A time block begins with a time (a non-negative integer value) followed by zero or more cell location and value pairs. When multiple time blocks are given, successive times must be strictly greater than all previous times. An example set of time blocks for a 2–dimensional automata is given in Figure 4. It specifies that although all the cells of the universe have a default value of 0 at time 0, the cell at absolute location 3, 3 will have a value of 1. Furthermore, it specifies that at the beginning of time 100 (before any cells have been updated), the

[4] Relative indexing specifies a cell relative to the current cell. Thus the relative index "[-1, 1]" refers to the bordering northwest cell.

```
2 dimensions of 0..1

sum := [0, 1] + [1, 1] + [1, 0] + [-1, 1] + [-1, 0] +
       [-1, -1] + [0, -1] + [1, -1]

cell := 1 when (sum = 2 & cell = 1) | sum = 3
     := 0 otherwise
```

Figure 1: The game of Life

```
2 dimensions of 0..1

sum := 0
forall i:-2..4
    forall j:-3..1
        sum := sum + [i, j] when i != 0 | j != 0
    end
end

cell := 1 when (sum = 2 & cell = 1) | sum = 3
     := 0 otherwise
```

Figure 2: The game of Life with a large asymmetric neighborhood.

2 Language Basics

Programs written in *Cellang* have two main components, a cell description and a set of statements.[3] The cell description determines how many dimensions there are, what field(s) each cell contains, and the range of values that can be associated with each field. Statements occur in three varieties: assignment, if and forall. The assignment statement is unusual in that it provides conditional assignment; the if statement is of the conventional nature; and the forall causes an index variable to iterate through a range of values.

The cells of the automata reside at the integer lattice points of an n–dimensional Euclidean space. Due to the limitation of finite memory, only a finite number of cells can be declared. To prevent the creation of a discontinuous boundary at the edge of the space, the lattice of cells folds back upon itself to form a closed surface. Thus, for $n = 2$ the universe would form a torus. The field values of cells neighboring the current cell may be accessed by placing their relative position within square brackets ([ ]), followed by an optional dot (.) and field name.

There is only one primitive data type in *Cellang*: integer. Variables are declared by their first usage as the left hand side of an assignment statement. Conditional expressions follow the rules of *C*[8], thus a 0 value corresponds to false and non-0 values indicate true. The arithmetic and logical operators available are: +, –, *, /, % (remainder), +% (modulo addition), –% (modulo subtraction), ! (logical not), & (logical and), | (logical or), =, !=, <= and >=.

The game of Life, created by Cambridge mathematician John Horton Conway and popularized by Martin Gardener[7], demonstrates many of the features of *Cellang*. Figure 1 shows a *Cellang* program for this problem. Note that the universe of cells is 2–dimensional, each having either a 0 or 1 value. Comments begin with a sharp (#) and continue to the end of that line. The predefined variable `cell` is special as it refers to the current cell of the universe under consideration. Assignment to the variable `cell` is the only way in which the value of the current cell (or one of its fields) can be altered. Furthermore, the new value of `cell`

[3] The language assumes a uniform transition function over all the cells of the automata.

A *Cellular* Automata Simulation System

J Dana Eckart
Computer Science Department
Radford University
Radford, VA 24142
dana@runet.edu

10 August 1998

1 Introduction

Cellular automata were first introduced by Stanislaw Ulam and John von Neumann in the late 1940's.[17][11] A cellular automata consists of a possibly infinite, n–dimensional lattice of cells. Each cell has a state chosen from a finite alphabet. The state of all cells in the lattice are updated simultaneously and independent of one another in discrete time steps. Cells update their values by using a transition function. The transition function takes as its input the current states of the local cell and some finite collection of nearby cells that lie within some bounded distance, collectively known as a neighborhood. With the growing popularity of cellular automata in both recreation [7][4][6][5] and the modeling of physical systems[16][3][10][2][15], the need for an easy-to-use system for cellular automata programming is greater than ever. *Cellular* is a system designed and implemented by the author to meet these needs.

The *Cellular* system consists of: a programming language, *Cellang*, and associated compiler, `cellc`; an *a*bstract *v*irtual *c*ellular *a*utomata *m*achine[1] for execution, `avcam`; and a viewer, `cellview`. Compiled *Cellang* programs can be run with input provided at any specified time during the execution. The results of an execution can either be viewed graphically, as an output stream of cell locations and values, or passed through a custom filter before being reported. The output stream, or a similar stream produced by a custom filter, can also be fed into `cellview` for viewing, or it may be passed through other programs that compile statistics, massage the data, or merely act as a valve to control the flow of data from the cellular automata program to the viewer.

This simple Unix[2] toolkit view of the simulation process provides greater control than systems which combine the language and viewer (i.e. Cellsim[9] and CAM-6[16]). *Cellang* provides greater flexibility, particularly in the formation of neighborhoods, than does *Cal* (part of the *Scamper* system)[12] and more safely than can be accomplished in *Cellsim*[9]. The use of a deterministic rule set (as opposed to probablistic rules supported by the *SLANG*[13] system), specified using an imperative programming paradigm allows *Cellang* to be a small, simple, and easy to learn language. Finally, *Cellang* combines the classic cellular automata programming paradigm with that of agents (as exemplified by the *Creatures*[14] system). This combination allows a richer solution space to many of the interesting problems in modeling and simulation.

[1] This gives the system the flexibility to be adapted to support a variety of platforms.
[2] Unix is a registered trademark of AT&T.