Complex Systems **5** (1991) 575–601

The Cellular Device Machine Development System for Modeling Biology on the Computer

Hans B. Sieburg*
Oliver K. Clay
Artificial Biological Systems Project,
HIV Neurobehavioral Research Center,
Department of Psychiatry, 0603-H,
University of California, San Diego, CA 92093-0603, USA

Abstract. This paper describes a development system for designing, implementing, and executing biologically motivated cellular automaton simulations. A new object-based programming language that is fundamental to the system is discussed in detail. Use of the language is illustrated in simple applications.

1. Introduction

The simulation and modeling approach described in this paper originally came about for two reasons. First, extracting blood or cerebrospinal fluid from a patient every five minutes over an extended period of time is difficult at best. Second, laws of disease progression are easily obscured by the simple mass of data collected in a long-term and diversified clinical cohort study. Consequently, computer-supported prediction becomes essential to fill gaps in, or extract features from, large databases by flexibly experimenting with hypotheses. As we have shown in our previous applications to HIV-related nervous and immune system disease [17–20], our cellular automaton (CA) based simulation approach responds to this need.

Given the broad spectrum of applications, the goal of this paper is to make our method more broadly available for further development.[1] Specifically, we would like to share our modeling language SLANG, which offers a new and expandable approach to programming lattice dynamical systems. After a brief introduction of basic terms, the second section describes the essential simulator kernel algorithms. The third section introduces our object-based

*To whom all correspondence should be addressed. Electronic mail address: `hsieburg@ucsd.edu`

[1]The software is available through anonymous FTP at `polaris.cognet.ucla.edu` in the directory ~ftp/pub/alife.

simulator language SLANG. In the fourth section, we present simple examples that illustrate the usage of SLANG. Finally, in section 5, we discuss work in progress and the modifications to traditional CA algorithms that we use in our present implementation.

2. Basic terms and algorithms

A *Cellular Device Machine* (CDM) is a soft architecture machine consisting of a nearest-neighbor lattice dynamical system of discrete *sites* (the *body*) in which (possibly mobile) finite-state automata (the *cell devices*) reside and interact according to local rules (figure 1). In this paper we describe a *CDM Development System* (CDM-DS) on which these soft architectures can be created and run as virtual machines on general-purpose hardware platforms.

Each cell device contains a pointer to an algorithm called *script*, which is composed of simple unnested "if...then...else..." statements. These *production rules* are each equivalent to a 2-arc subgraph of a hypergraph, thus allowing an entire device algorithm to be displayed graphically as well as sententially (see section 3 for more detail). Cell devices sharing the same script are identical if viewed as finite automata, and are thus said to belong to the same *species*.

The CDM-DS provides utilities to create, program, and run Cellular Device Machines. A CDM and its operating system are created by manipulating five global parameters via a dialog window:

1. Body size: The body size $B = A^2C$ is determined by the area A^2 of the lattice multiplied by the maximum occupancy *capacity* C per site. The latter parameter refers to our subdividing of sites into "slots" that can be filled dynamically from a selection of up to 256 species. On 1 megabyte of RAM we can support $A = 16$, 32, and 64, and $C \leq 16$. Larger sizes using $A = 128$ and 256 and $C \leq 256$ can be supported using additional RAM. To allow both the option of fast implementations requiring more RAM and slower implementations that use memory more efficiently, we require that no site may contain more than one member of the same species (Exclusion Principle).

2. Neighborhood topologies: In our present implementation, the local structure of the lattice world is represented by an ordered sequence of pointers to lattice sites. The definition of a cell device does not make any assumptions regarding this sequence. Using sets of pointers increases modeling flexibility as we are free to experiment with potentially different subsequences for the input to and the output from a center site C, for example of a 9-site Moore neighborhood {C, N, NE, E, SE, S, SW, W, NW}. Here, N denotes North, NE Northeast, and so forth. In some complex applications, such as our immune system models for example, we use the input neighborhood {C, N, E, S, W} and output neighborhood {C, NE, SE, SW, NW}. Another example

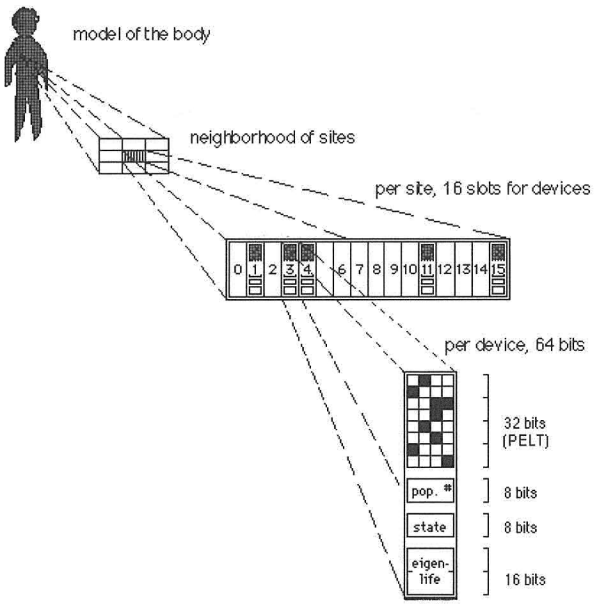


Figure 1: Overview of the CDM-DS data structures. The *body* is a representation of random access memory (RAM) as a nearest-neighbor lattice dynamical system with periodic boundary conditions. A *cell device* occupies 64 bits in the body. 32 bits are allocated for a *pattern element* (PELT), 8 bits each are dedicated to a species number and a *state* ($0 \ldots 255$), and 16 bits are used by an internal clock called *eigen-life* ($-32768 \ldots 32767$). The figure indicates 16 slots as the maximum occupancy capacity per lattice site, which is standard for 1-megabyte CDM implementations. More slots can be supported using additional RAM.

is the Game of Life, where the Moore set is the input neighborhood, but any Moore subset, the simplest being $\{C\}$, will function as an output neighborhood (see section 4). We are presently investigating the theoretical significance of these heuristical observations.

3. Percent scan: The "operating system" of each CDM that we construct uses an "attentional searchlight" to scan the lattice world. This searchlight can be programmed by the user to execute an arbitrary number of random prompts (with replacement) per update cycle. Each prompt temporarily passes control to a (possibly empty) cell device. The number of prompts is determined by

$$\#\ \text{prompts} := [(B * \%\text{Scan}/100)],$$

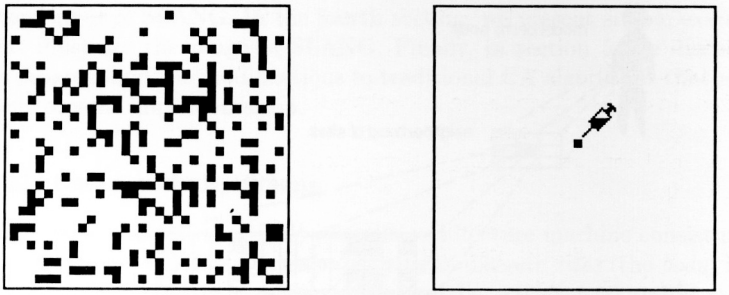


Figure 2: To create the initial configuration of an experiment we use a "boosting" dialog. This allows us to establish population characteristics of a species such as distribution and diversity. As the sample plots show, we can choose either to distribute a population randomly across the lattice world (left) or use a "syringe" tool to place single specimens or subpopulations at specific locations (right). If several species are involved, the syringe allows us to construct defined micro-environments. As a simulation progresses, we can therefore study the evolution of these micro-environments and their impact on the surrounding larger space. Naturally, the boosting procedure can be executed at any time during an experiment. To keep track of changes in the distribution and diversity of a population we implemented a "dump" command, which allows us to save an entire lattice for later analysis. Such dumps can also be used as initial start-up configurations. Here the modeler gets hands-on interactive experience of the effects that changes to a species script, for example, or to the input or output neighborhoods, would induce on the overall population dynamics. This results in very controlled and rapid hypothesis testing.

where $[x]$ defines the largest integer $\leq x$, and %Scan = 100 corresponds to a full update. A number less than 100 for %Scan indicates that only a fraction of the body is scanned by the random searchlight. The # prompts thus calculated determines one machine cycle of the CDM.

4. Aging: Aging is an important, but not well understood, aspect of biology. Naturally, different "life counters" have been used in simulations of biological systems. For example, the "Ulam death rule" erases all cells that are k generations old [25]. More subtly, Hebbian synapses that are left unused for a long time are discarded or assigned a lower weight, and strengthened if they are used frequently [11]. We have opted to use a mixture of both ideas to create the object-controlled internal clock that we refer to as *eigenlife* [18, 20]. The eigenlife of a cell device can be increased only by its own actions, that is, if it is prompted and a production rule in its script allows it to raise its eigenlife. Therefore, through its species script, each device is given a

```
read in initial configuration and seed of 'random'
sequence;
for (cycle=1;;cycle++) {
  for (i=0;i<prompts per cycle;i++) {
      pick a 'random' device;
      prompt it, giving it control until it finishes
      its tasks;
  }
  for (j=0;j<swaps per cycle;j++) {
      pick a site;
      pick a 'random' neighboring site;
      swap their contents;
  }
  if (automatic eigenlife decrementing requested)
      decrement all devices' eigenlives by 1;
  remove all devices with eigenlife ≤ 0;
  update screen plots and displays, output to files etc.;
  if (user requested stop) {
      pause;
      process user's run-time requests;
  }
}
```

Listing 1: Tasks of a CDM's operating system. In keeping with our view of an experiment as a run of a virtual machine composed of smaller machines called cell devices, we can speak of a machine cycle, also loosely referred to as a *timestep*. The operating system of each CDM that we construct can be programmed by the user to execute an arbitrary number of random prompts per cycle (where each prompt temporarily gives control to a possibly empty device), followed by any motion or eigenlife-decrementing tasks requested by the user during the design, and finally by "garbage collection" (any devices with eigenlife ≤ 0 are removed) and output to screen, files, and so forth. At this time the user has the choice of temporarily stopping the machine ("interrupt") to view populations, request plots, boost, dump, and so forth.

repertoire of strategies that it can use to increase its life expectancy beyond an initial value. Typically this is as a result of an interaction with its micro-environment. In our implementation, eigenlife *decreases* can be set to occur *globally* once per cycle as an operating system task if a universal decrementer is turned "ON". Alternatively, if the universal decrementer is "OFF", a cellular device may decrement its eigenlife through its script when it is prompted. These options for regulating eigenlife decreases correspond to different theories on aging through a systemic or individual "internal clock" (e.g., [9, 10, 1]).

```
on prompt, go to 'beginning' of species script;
while (there are production rules addressing the prompted
cell device) {
  execute next production rule in the script;
  if a '•' was encountered
      break;
}
```

Listing 2: Tasks of a cell device. When a cell device is prompted, the species' script file is searched for the first production rule that addresses its characteristics. A new line in the script file of a cell device is recognized as the beginning of a production rule if and only if its first printing character is an opening bracket ("("). Any other lines are assumed to be comments and will be skipped. Then the recognition/action conditions in the rule are executed, typically resulting in modifications of the prompted device and/or its neighbors. If the production rule is not terminated with a "break" command ("•"), the next rule addressing the (now possibly modified) cell device is executed. Script files are "circular", so interpreting continues until a "•" is encountered or until no more production rules address the device, after which another cell device is prompted.

5. Motion: At present the CDM-DS supports two kinds of motion. If the number N of blind swaps is chosen ≥ 1, a "blind swap" algorithm [24] creates local jitters in the lattice world. For $N = 0$, all cell devices will remain immobile unless their scripts contain instructions that allow random or directed local motion once per prompt. In a new version of the software we will implement a third kind of motion, a modified blind-swap algorithm that respects immobility.

Once these parameters are set, the Development System creates an "empty slate" CDM ready for experimentation. The first step in creating an experiment is to select or design the species that one would like to add to the project. The second step involves setting initial spatial configurations and specimen distributions for the added species (figure 2).

An experiment starts when the machine is "launched" on a pair (initial random seed, initial configuration). A CDM runs by prompting cellular devices using a "searchlight" algorithm, which scans the body asynchronously. The prompting process is guided by a hard-wired pseudo-random number sequence. Although this is not implemented in our current version, using a one-dimensional CA such as rule 30 [28] is conceptually compelling as it allows us to view as quasi-stochastic the evolution of the configurations of a CDM, yet at the same time the pair (next device prompted, current configuration) evolves as a fully deterministic CA. The "searchlight" algorithm determines the order in which cell devices are given a chance to interact with their local environment. The operating system tasks are explained in the pseudocode listing shown in listing 1. When prompted, a cell device will perform the tasks outlined in listing 2.

3. SLANG

The object-based CDM-DS *S*imulator *LANG*uage "SLANG" uses conditional "if...then...else..." production rules of the form

```
recognition condition
  -> (action when condition evaluates to true)
  :  (action when condition evaluates to false);
```

where

```
recognition condition
      := refObject.[ recMin,recMax] recObject
```

or unconditional "if...then..." production rules of the form

```
refObject -> (action);
```

Recognition occurs if and only if there are $\geq$ `recMin` and $\leq$ `recMax` recognizable devices `recObject` in the input neighborhood of the prompted *reference device* `refObject` (see Example 1 below for the precise meaning of recognition). `recMin` and `recMax` are integers whose ranges depend on the neighborhood type. For example, for the Moore neighborhood we have $1 \leq$ `recMin` $\leq$ `recMax` ≤ 9. The closed interval `[recMin,recMax]` will be referred to as the *recognition window*. If recognition occurs, the *true part* ("->") of the transition is interpreted, otherwise the *default part* is interpreted (":").

action refers to a combination of changes to the reference device `refObject` and optional changes to the contents of its neighborhood:

```
action := repObject;[put/delete commands];
                [[move]break commands]
```

Here the notation `repObject` (for *replacement device*) indicates potential changes to the reference device. The optional `[put/delete commands]; [[move]break commands]` are defined by

```
put command              := > putNumber putObject
delete command           := < delNumber delObject
[move]break command      := [X]•
```

The notation ";" separates the three types of changes in an action, the last two of which may be empty. The ">", or "put", instruction indicates that `putNumber` of `putObject` are to be placed into the output neighborhood if `putNumber` is less than or equal to the number `#free` of free sites. Otherwise, only `#free` sites will be filled. This convention prevents excessively long searches for available space. The "<", or "delete", instruction indicates that `delNumber` of `delObject` are to be removed from the input

neighborhood if `delNumber` is less than or equal to the number `#free` of free sites. `putNumber` and `delNumber` are integers whose ranges depend on the sizes of the input and output neighborhoods, respectively. The break command "•" is a jump condition that returns control from a production rule within a script to the operating system (see listing 2).

`move` instructions, which, for example, for the Moore neighborhood template are given by

```
X :=  C  := move one site to Center site (equivalent to
            'do not move')
      N  := move one site to North site
      NE := move one site to NorthEast site
      E  := move one site to East site
      SE := move one site to SouthEast site
      S  := move one site to South site
      SW := move one site to SouthWest site
      W  := move one site to West site
      NW := move one site to NorthWest site
      ®  := move one site in a random direction
```

NW	N	NE
W	C	E
SE	S	SW

can only be used in conjunction with the `break` command.

The "reference device" `refObject`, the "recognition device" `recObject`, the "replacement device" `repObject`, and `putObject` and `delObject` are bundle data types declared through the following type definition:

```
typedef struct bundle{
  long           pelt;   /* pattern element */
  unsigned char  device; /* species denominator */
  unsigned char  state;  /* state denominator */
  short          life;}  /* eigenlife value */
bundle;
```

The pattern element or `pelt` is a 32-bit string that can be given *explicitly* in any of four different representations:

	decimal	(default; no denotational symbol)
$...	hexadecimal	(e.g., `$3B214AFC`)
b...	binary	(e.g., `b01110101011010110011111101111000`)
'...'	ASCII	(e.g., `'ASC2'`)

All representations should define a 32-bit integer; thus, a decimal representation must lie between −2147483648 and 2147483647, a hexadecimal representation may have at most 8 digits, a binary representation may have at most 32 digits, and an ASCII representation must have exactly 4 characters.

The species denominator `device` can be given *explicitly* by an ASCII string. The state denominator `state` can be given *explicitly* by a decimal

integer between 0 and 255 (inclusive). The eigenlife value can be given *explicitly* by a decimal integer between −32768 and 32767.

The richest and most complicated aspect of SLANG is *implicit* definition through special instructions to enforce inheritance or device modifications. The following is a list of these special instructions, where the term "corresponding bits" refers to an arbitrary section of the canonical bit-model of a cell device (figure 1):

`#`	wild card	("allow any")
`?`	inherit	(from corresponding bits in `refObject`)
`!`	complement	(of the corresponding bits in `refObject`)
®	mutate	(assign random value)
`%n`	block mutate	(same as pelt of `refObject`, but with a randomly chosen n-bit block replaced by a random n-bit block ("randomly mutate in n consecutive bits"))
©	copy	(from corresponding bits in the last `recObject`)
`¢n`	block copy	(same as pelt of `refObject`, but with a randomly chosen n-bit block replaced by the corresponding block in the last `recObject` ("copy n consecutive bits"))

Using "`?`", "`!`", "©", "®", "`%`", and "`¢`", pelts may be specified *implicitly*, or in *mixed implicit-explicit* form by substituting one or more digits of a hexadecimal, binary, or ASCII representation with "`?`", "`!`", and so forth. A "`?`" will "inherit" the corresponding bits (4 bits for a hex digit, 1 bit for a binary digit, 8 bits for an ASCII character) in the reference pelt. A "`!`" instruction will access the complements of the corresponding bits in the reference pelt.

Species and state denominators may be given *implicitly* by "`?`". The eigenlife value may be given *implicitly* by the inheritance instruction "`?`" or by increment or decrement functions of the form "`?+` integer", "`?-` integer".

The use of special instructions is restricted as follows:

```
refObject    #
recObject    #, ?, !, ©, ®
repObject       ?, !, ©, ®, %, ¢
putObject       ?, !, ©, ®, %, ¢
delObject    #
```

Therefore, the wild card "`#`" symbol is allowed in the bundle specifications of `refObject`, `recObject`, and `delObject`, but does not make sense anywhere in the bundle specifications of `repObject` and `putObject`. Otherwise, `putObject` is very similar to `repObject` since both can use "`?`", "`!`", "©", and "®". In `refObject` and `delObject`, "`?`", "`!`", "©", and "®" do not make sense and will therefore lead to syntax errors.

An example of a syntactically correct production rule is shown in listing 3.

```
(#,MP,1,#).[3,#](#,LPS,#,#)
  -> ((?,?,2,?);>1('TNF',TNF,1,9),<4(#,LPS,#,#);•)
  :  ((?,?,?,?);;•);
```

Listing 3: This production rule means that, if the activated reference device `(#,M,1,#)` recognizes, regardless of the pelt, at least three recognition objects `(#,LPS,#,#)` in its input neighborhood, it will change its state from 1 to 2, maintain its patterns element and eigenlife value, and produce one product `('TNF',TNF,1,9)`, while removing four `(#,LPS,#,#)` devices. If no recognition occurs because there are too few `(#,LPS,#,#)` devices in the input neighborhood, then the activated device will not change. The production rule as shown could be used, for example, to describe the secretion of tumor necrosis factor (TNF) by a macrophage (MP) stimulated by lipopolysaccharide (LPS).

To illustrate the typical usage of special instructions in the pelts, we give a few examples. Here, `refPelt`, `recPelt`, `repPelt`, `putPelt`, and `delPelt` denote the pelts of the `refObject`, `recObject`, `repObject`, `putObject`, and `delObject`, respectively.

Example 1. Combinations of the instructions "?", "!", and "#" can be used to define a large number of *recognition processes*. For example, the recognition condition `(#,X,1,#).[1,#](#,Y,#,#)` checks only for the presence of at least one Y in the neighborhood of X. As indicated by the wild cards, the specific pelt structure, the state, and the eigenlife of Y do not matter in this Boolean recognition process. The next two cases are different. In `(#,X,1,#).[3,#]('##??',Y,#,#)`, recognition occurs only if the last two ASCII characters of Y are the same as the corresponding ASCII characters in the pelt of X. The first two characters are inconsequential to the recognition process. Even more complicated, in `('ABCD',X,1,#).[3,#]('!!?#', Y,#,#)` recognition only occurs if the first two ASCII characters in the pelt of Y are the complement of the corresponding characters in the pelt of X, and the third characters are equal. The fourth character is inconsequential. Using recognition processes such as the ones just described has the advantage that *recognition* is a fully user-controlled event that is well defined during execution. The values of the eigenlife function $E(X;$ recognition condition$) :=$ `refLife` + `repLife` $- 1$ of a cell device X are then also well defined during execution. `refLife` and `repLife` indicate the eigenlife parameters of `refObject` and `repObject`, respectively.

Example 2. If `refPelt` = `$F0FF0001`, then `repPelt` = `$AB??00F!` will give us an actual `repPelt` of `$ABFF00FE`, since \$E = b1110 is the bitwise (4-bit) complement of the last hexadecimal digit \$1 = b0001. Therefore, the unconditional production rule

```
($F0FF0001,X,1,#) -> (($AB??00F!,?,?,?);;•);
```

states that, when prompted, a specimen of species X in state 1 and with pelt `$F0FF0001` will acquire the new phenotype `$ABFF00FE` using `$AB??00F!`, without changing its state or altering its internal clock. As indicated by the "•", control is immediately returned to the operating system following the transformation.

Example 3. The following production rule illustrates sexual reproduction where the offspring acquire properties from both parents using "©" and "?":

```
(#,X,#,#).[1,1](#,X,#,#)
  -> ((?,?,?,?);>1($?©??©©?©,X,0,1);•)
  :  ((?,?,?,?);;•);
```

The rule states that if any two members of the X population meet, they will produce offspring with mixed characteristics according to the genetic program `$?©??©©?©`. For example, if `refPelt` = `$F0FF00C1`, and `recPelt` = `$ABCDEF03`, then `putPelt` = `$FBFFEFC3`. The offspring will be born in state 0 and with a life expectancy of 1.

Example 4. For the binary pelt `b110111110®011111®®®®®1100011101` we have marked fixed areas where a point and a block mutation can occur using the "®" instruction. This means that wild bits occupy the indicated positions. An interesting application is in the study of retroviruses that require exogenous activation of their host for reproduction. Due to this condition, mutations are frequent, and the mutation rates are correlated with the rates at which the host cell population is activated by different stimuli. Cell internal processes triggered by the stimuli also seem to cause phenotype mixing between viral and host genomes, although probably at a lesser frequency. Therefore, in the theoretical study of quasi-species formation from an initially pure virus strain, we may use the "?" instruction together with the "®" instruction. For example,

```
(#,CELL,#,#).[1,#](#,STIM,0,1)
  -> ((?,?,?,?);<3(#,STIM,#,#),
       >1(b??????00110111®®®®????????????,VIRUS,0,1);•)
  :  ((?,?,?,?);;•);
```

indicates that an infected cell `CELL` of

`refPelt` = b11011111010111110[1001100011101]

that is activated by a stimulus `STIM` of

`recPelt` = b01110101011010110011111011111000

may produce a virus strain `VIRUS` of

`putPelt` = b??????0011011®®®®1[????????????]

The boxes are not part of the SLANG syntax, but indicate the invariant viral genes that are integrated into the host genome and that are also present in the new strain. The unboxed "?" mark an area of phenotype mixing that may result in strain adaptation. This is separated by a fixed pattern from an area of random mutations. According to the sample production rule, successful stimulation, which results in the uptake of a certain quantity of the stimulant, does not change the host.

Example 5. © allows for a unidirectional copy. The following sample production rule illustrates how a bidirectional crossover operation, as it is commonly used in genetic algorithms [12], can be constructed from ©:

```
(#,X,#,#).[1,#](#,Y,0,1)
  -> (('?©??',?,?,?);<1(#,Y,#,#),>1('©?©©',Y,0,1);•)
  :  ((?,?,?,?);;•);
```

Suppose the ASCII representation of `refPelt`, that is the pelt of the prompted X specimen, is `'ABCD'`, and that `recPelt` = `'EFGH'`. According to the script, `repPelt` = `'?©??'` = `'AFCD'`. This is the result of the unidirectional transfer of information from Y to X using ©. The transfer of information from X to Y is coded in the pair of put and delete instructions `<1(#,Y,#,#)` and `>1('©?©©',Y,0,1)`. We first remove one member of the Y species, and then immediately create a new one of the same state and eigenlife as the original `recPelt`, but with `recPelt` = `'©?©©'` = `'EBGH'`.

Example 6. The following sample rule shows how the "`%n`" instruction is used to create a `repPelt` and a `putPelt` that are the same as the `refPelt` except that a randomly chosen block of n bits is replaced by a block of n random bits:

```
(#,X,#,#).[1,#](#,Y,0,1)
  -> ((%10,?,?,?);<1(#,Y,#,#),>1(%5,Y,0,1);•)
  :  ((?,?,?,?);;•);
```

The effect is

```
refPelt = b11011111010111110010011000111101
repPelt = b11011[1010110101]1001001100011101
putPelt = b11011111010111110011[10001]0011101
```

where the boxed portions of the bit-strings indicate the replaced blocks.

Example 7. The following sample rule shows how the "`¢n`" instruction is used to create a `repPelt` and a `putPelt` that are the same as the `refPelt` except that a randomly chosen block of n bits is replaced by the corresponding n-bit block of `recPelt`:

```
(#,X,#,#).[1,#](#,Y,0,1)
  -> ((¢8,?,?,?);<1(#,Y,#,#),>1(¢16,Y,0,1);•)
  :  ((?,?,?,?);;•);
```

The effect is

```
refPelt = b1101111101011111001001100011101
recPelt = b0001001011001010111111111101000
repPelt = b11011[01011001]11100100110001110 1
putPelt = b1101111101011 11[0111111111101000]
```

where the boxed portions of the bit-strings indicate the replaced blocks.

The full language specification is summarized in Backus-Naur Form [27] in listing 4. It is easy to see that any SLANG production rule corresponds to a 2-arc subgraph of a hypergraph (figure 3). Therefore, it is possible to design a visual programming language equivalent for SLANG [2]. The use of graphics is particularly attractive for non-specialists interested in using the CDM-DS.

Another corollary of the correspondence between SLANG scripts and graphs is that we can write a SLANG debugger based on a circuit-checking algorithm. This algorithm derives from the following result [3]:

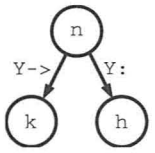
Theorem. *A directed graph G with transition matrix A contains a path of length λ iff $A^\lambda \neq 0$; it contains no circuits iff, for all λ sufficiently large, $A^\lambda = 0$.*

The need for circuit checking arises since any closed path ("circuit") in a cell device's graph can lead to an endless loop in which the active reference device becomes a "dictator" of the experiment, which forever controls the operating system. The introduction of the break command "•" avoids "dictatorship". However, quite often situations arise in which the user wants to have a cell device execute a chain reaction; that is, several production rules are processed in succession, and therefore selectively omit the terminator from the true or default actions. In this case, the circuit checker constitutes a last safeguard that alerts the user to the potential dangers of his or her intentions.

All production rules in a cell device script file using a reference bundle of the form `(refPelt,refPop,refState,#)`, where `refPelt` is either given explicitly or by "#", are included in a check for circuits. The checker first creates the matrix of all state transitions that are not terminated with a "•". If in this matrix of transitions that can contribute to a circuit there are no non-zero diagonal elements (= no self-loops), it then keeps calculating powers of this matrix until either the resulting matrix is 0—in which case there are no circuits—or until the Nth power is reached, where $N := \min\{\text{number of edges} + 1, \text{number of vertices}\}$. If this power matrix is still $\neq 0$, the transition graph contains a circuit since, if there is a path of length greater than the number of edges, an edge was traversed twice, and if there is a path of length greater than or equal to the number of vertices, we return to at least one vertex a second time. Furthermore, if the `refState` entry is a wildcard ("#"), an alert is posted whenever a "•" command is missing, as endless loops and other pitfalls happen very easily

```
sentence = unconditional_sentence | conditional_sentence.

unconditional_sentence = refOClass -> (repObject;IO;[direction•]);.
conditional_sentence =   refOClass.recWindow recOClass
                         ->(repObject;IO;[direction•])
                         :(repObject;IO;[direction•]);.

IO = io.
io = {io_op [ , ]}.
io_op = delete | put .

direction = C | N | NE | E | SE | S | SW | W | NW | ®

recWindow = [recMin,recMax] .
recMin = threshold.
recMax = threshold.

delete = < threshold object_class,.
put = > n object,.
refOClass = object_class. (additional restriction: NO ?, !, ®, ©, %, ¢ anywhere)
recOClass = object_class.
repObject = object.

object = object_class.     (additional restriction: NO #'s anywhere)
object_class = (pelt_class, device_class, state_class, eigenlife_class).

pelt_class = bB | $H | 'A' | N32 | ? | ! | # | ® | © | %N5 | ¢N5 .
device_class = X | ? | # .
state_class = N8 | ? | # .
eigenlife_class = N15 | ? | ?+N15 | ?-N15 | # .

threshold = n | # .

B = b[b][b]...[b]      (string of length ≥ 1 and ≤ 32)
H = h[h][h][h][h][h][h][h].
A = a[a][a][a].
Nk = 0 | 1 | 2 | 3 | ...      (decimal string representing any k-bit
                               integer 0...2^(k-1))
X = devicename1 | devicename2 | ... (ASCII name-string for any of
                               the devices)

b = 0 | 1 | e .
h = 0 | 1 | 2 | ... | 9 | A | ... | F | e .
a = A | ... | Z | a | ... | z | n | e .

e = ? | ! | # | ® | © .
n = 0 | 1 | 2 | 3 | ...
```

Listing 4: CDM-DS script language definition in Backus-Naur Form (BNF). Using this language specification format, the syntactically correct production rules are those that can be obtained from "sentence" by successively applying the equations or *substitution rules* in this listing. "...|..." means "... or ..."; "[...]" means "... is optional"; "{...}" means "... may be repeated as many times as we wish, but is optional". Strings in the standard text font (e.g., pelt_class and repObject) and bold capital letters in typewriter font (e.g., **H** and **A**) are used as *non-terminal symbols* to represent parts of the sentence. Bold lowercase letters in typewriter font (e.g., **h** and **a**) are non-terminal symbols for single characters in the sentence. Plain characters in typewriter font (e.g., (and 6) are *terminal symbols*, that is characters as they actually appear in the script files. Bold non-alphanumeric characters in typewriter font (e.g., | and {) are BNF metasymbols.



⇔

```
(#,X,n,#).[#,#](#,Y,#,#)
-> ((?,X,k,?);;●)
:  ((?,X,h,?);;●);
```

Figure 3: The one-to-one correspondence between a SLANG production rule and a small hypergraph is established using dialog windows for each edge and vertex. Specifically, a vertex dialog requests information regarding bundle data structures of the species. Also covered are motion instructions and breaks. The vertex dialog also takes into account that, in larger graphs, particular vertices may pose as both a replacement and a reference device. An edge dialog requests information regarding the transition conditions. In the example shown, successful recognition is encoded in the edge labelled "`Y->`" and requires thresholds for the recognition window and the settings of the bundle data structure of species Y. The edge labelled "`Y:`" is the default edge. Defining the default involves only two user responses: entering "`Y`" as the edge name and setting the "default" radio button to "ON", which disables all other responses. Not shown in the figure is that, in our present implementation of the interface, we use rectangular vertices to represent any action that may result from a transition. The connecting edge between a rectangular "action vertex" and a circular "replacement vertex", for example "`k`" or "`h`" above, is automatically characterized as an "action edge" and therefore requires no further user information.

in this case. For example,

```
(#,A,#,#) -> ((?,A,2,?+1);;);
```

leads to an endless loop as soon as a member of population A with state 2 is prompted. Also,

```
(#,A,#,#) -> ((?,A,?,?+1);;);
```

will always lead to an endless loop as soon as a member of population A is prompted.

4. Explicit examples

We chose examples that are well established in the literature to encourage comparisons and thereby facilitate the learning of the language. A more elaborate application to HIV infection is discussed in [19, 20]. All examples were run on an Apple Macintosh IIcx™ personal computer with 8 megabytes of memory.

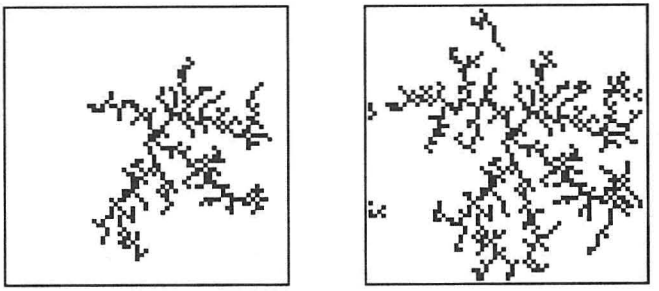


Figure 4: The plots show two intermediate clusterings in the lattice world of "B". Global motion and automatic eigenlife decrement are turned "OFF" in this experiment. 100% of the body is scanned per cycle. We used a random initial distribution for "A", which occupies 25% of a 64^2 sites body. For "B" we used a single initial seed, which was placed approximately at the center of the lattice world. Since there is no automatic eigenlife decrementing, we set the initial eigenlife for both species to 1. We used the full Moore neighborhood for the input and output neighborhoods. Note that, due to the periodic boundary conditions, wrap-around clustering occurs on the left and upper sides of the plot on the right.

1. *Cluster Formation on a Torus*: One of the simplest examples leading to complex behavior, namely fractal spatial structure, is dendritic growth through diffusion-limited aggregation. In our present implementation, two species are required to visualize this process. One, denoted by "A", consists of randomly mobile specimens described by the following script:

```
(#,A,1,#).[1,#](#,B,1,#)
  -> ((?,A,1,?);>1(?,B,1,?),<1(?,A,1,?);•)
  :  ((?,A,1,?);;®•);
```

This means that any "A" specimen that recognizes at least one "B" specimen will transform itself into a member of species "B" according to the self-elimination `(?,A,1,?); ..., <1(?,A,1,?)` and the reappearance statement `...;>1(?,B,1,?),...`. The new "B" specimen inherits the pelt and the eigenlife value from the departing member of "A". By default, if an "A" does not see a "B" specimen, it will move one step in a random direction. Note that the "®" instruction functions as the diffusion term in an environment where global motion is turned "OFF". The second species, denoted by "B", is immobile and passive, and therefore has the *trivial* script

```
(#,B,1,#) -> ((?,?,?,?);;C•);
```

The function of the "B" species is purely auxiliary since it carries the initial gluing seed and the growing cluster.

As shown by this example, dendritic growth through diffusion-limited aggregation does not seem to be dependent on the CA update paradigm; that is, the asynchronously updated CDM CA shows the same overall dynamical behavior as a synchronously updated CA [24, 16, 29]. This raises the interesting question whether asynchronicity should be the "normal" mode of CA implementation from which synchronicity can evolve as a model property. Also, we noticed that varying the width of the recognition window—using `(#,A,1,#).[1,k](#,B,1,#)` where $1 \leq k \leq 9$, instead of just `(#,A,1,#).[1,#](#,B,1,#)`—will result in different cluster densities. Whether this is reflected in variations of the fractal dimension is currently under investigation.

2. *Ising Models.* We included this example to illustrate the power of the SLANG language to encode lengthy verbal model descriptions in very few production rules. In [21], the authors describe how "One can calculate the spontaneous magnetisation on the computer [program ISING]. We set a 'spin' IS (atomic magnetic dipole moment) at each lattice position of a square lattice; IS = 1 or −1 according to whether the spin is up or down. Neighboring spins want to be parallel in this 'Ising'-ferromagnet of 1925. The energy will thus be conserved in a reversal of spin, if as many of the spins are up as are down, i.e. if the sum over the four neighboring IS is zero. In this sense the program reverses a spin [IS(i) = −IS(i)] if, and only if, the sum over the neighboring spins vanishes." The lattice in the demonstration program ISING is an $L \times L$ lattice with helical boundary conditions and two "passive" (unchanging) boundary rows added on top and bottom (therefore an $(L+2) \times L$ lattice). The input neighborhood used is the peripheral part of the von Neumann neighborhood, namely "N E S W". Conservation of energy must hold in the full von Neumann neighborhood "C N E S W", so the sum of the spins in "C N E S W" must be +1 or −1, and therefore the sum of the spins in "N E S W" must be 0. Since every site is occupied by a spin (+1 or −1) this can only happen if exactly two in "N E S W" have spin −1 and two have spin +1. In this case, the center spin will flip. Initially the spins are randomly oriented: IS = 1 with probability p, otherwise IS = −1. In other words, $p * 100\%$ of the spins are 1, the rest are −1. The program is demonstrated using $L = 40$ and $p = 0.2$, and run for 100 time steps on a fully updated CA.

This model is ported to the CDM-DS under the following conditions: (1) toroidal boundary conditions instead of helical boundary conditions, (2) lattice sizes 32 × 32 or 64 × 64 instead of 40 × 40, (3) Spin = −1 ⇔ pelt = $00000000, spin = +1 ⇔ pelt = $FFFFFFFF, (4) flipping of spin ⇔ inverting (complementing) of pelt. The latter demonstrates the usage of the "`!`" complement instruction in the `repPelt` of the replacement object. When translated into SLANG, program ISING is just one production rule (`!`):

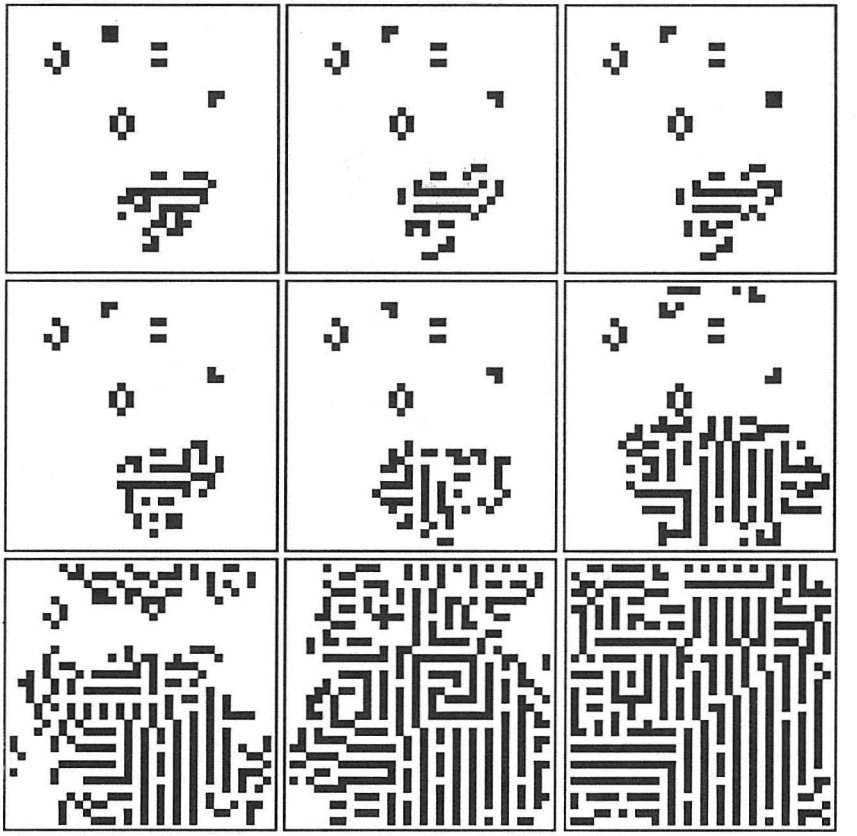


Figure 5: This figure shows 9 screen snapshots of live cells, namely cell devices of the bundle class (#,L,1,#), taken at irregular intervals during a simulation. Time evolution is shown from left to right. We chose regular start-up patterns, among them a glider shape placed in the lower half of the lattice world. As shown in the first plot, this glider explodes early on in the simulation, and the remnants of this explosion eventually fill all available space. For this demonstration we chose a body size of 32^2 lattice sites all of which are filled with members of the population L. Those members of L representing live cells are shown as black squares. The dead cells of bundle class (#,L,0,#) are invisible, represented by white squares. We chose the full Moore set as our input neighborhood. Since any subset of the Moore set can serve as the output neighborhood, we selected the center site "C". Neither pelts nor eigenlives are significant for this model. Global motion is turned "OFF" and individual motion does not occur.

```
(#,IS,#,#).[2,2](0,IS,#,#)
  -> ((!,IS,?,?);;•)
  :  ((?,IS,?,?);;•);
```

In our experiment we used the input neighborhood "N E S W" and the output neighborhood "C". Automatic decrementing of eigenlives and global motion were turned "OFF". 100% of the body was scanned per cycle. Initially, 20% of the body was randomly filled with pelt $FFFFFFFF and the rest with pelt $00000000. The dynamical behavior is the same as in the examples shown in [21].

3. *Game of Life.* We have ported this rule into SLANG because we wanted to see which patterns are generated by the asynchronous updating. The Game of Life [8] is based on the following verbal rule: "A dead cell (spontaneously) comes to life, if 3 live cells are in its neighborhood; if there are less than 2 live cells in the neighborhood of a live cell, the cell will die of loneliness; if more than 3 live cells are in the neighborhood of a live cell, this cell will die of overcrowding." This description easily translates into the following SLANG script, where a cell device "L" in state 0 represents a dead cell, and a cell device in state 1 represents a live cell:

```
(#,L,0,#).[3,3](#,L,1,#)
  -> ((?,L,1,?);>1(1,L,1,1);•)
  :  ((?,L,0,?);;•);

(#,L,1,#).[2,3](#,L,1,#)
  -> ((?,L,1,?);;•)
  :  ((?,L,0,?);<1(#,L,#,#);•);
```

As shown in figure 5, the long-term results of an asynchronous Game of Life simulation are irregularly structured patterns reminiscent of an electronic chip layout. As a simulation progresses, the microscopic connectivity of this layout continues to shift and rearrange itself. We are presently investigating the significance of these observations.

4. *Turing-Gierer-Meinhardt (TGM) Chemistry.* In this example taken from [4] we investigate a model system of pattern-generating chemical reactions. Four species are involved, denoted S, X, Z, and C. Species Z is capable of differentiation, which is induced by the activator X. The differentiation process is irreversible and C is the resulting differentiation product. Production of X requires a substrate S (figure 6).

For our demonstration experiment we initially seeded the S and X layers at 20%, and Z at 5% random uniform distribution of 32^2 sites (in other experiments we also tried 15%, 10%, 1%, and a single-site seeding for Z).

```
(#,S,1,#) -> ((?,?,?,?);®•);

(#,X,1,#).[1,#](#,S,1,#)
 -> ((?,?,?,?);>1(?,X,?,?),<1(#,S,1,#);•)
 :  ((?,?,?,?);®•);

(#,Z,1,#).[1,#](#,X,1,#)
 -> ((?,Z,2,?);>1(?,Z,1,?),<1(#,X,1,#);•)
 :  ((?,?,?,?);®•);

(#,Z,2,#) -> ((?,?,?,?);>1('C',C,1,1);•);

(#,C,1,#) -> ((?,?,?,?);®•);
```

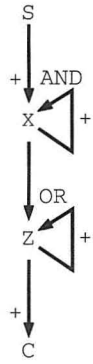


Figure 6: This figure shows the basic scheme of the sample TGM model on the right. To the left we have the corresponding SLANG production rules. This example demonstrates how easily graphical models typically employed by natural scientists to communicate ideas can be ported into SLANG and run on the CDM-DS. No particular pattern elements are used in this example, hence all pelts involved are wildcards "#". Both S and C have trivial scripts except that they are capable of random diffusion. The conditional rule for X states that, if S occurs at any non-zero concentration (`[1,#]`) in the neighborhood of an X specimen, S will be taken up and the X specimen will reproduce. If the concentration of S is 0, the specimen will continue its random motion. The production rules for Z state that any non-zero concentration of X will trigger differentiation of the Z specimen into a terminal state (`?,Z,2,?`), which inherits pelt and eigenlife. Also, Z will produce a state 1 offspring, and in the whole process will ingest 1 unit of the activator X. If no activation occurs, the Z specimen will continue random motion. A terminally differentiated Z specimen, as indicated by the self-inheritance (`#,Z,2,#`) `-> ((?,?,?,?);...)` rule segment, follows an unconditional rule, which states that each time the specimen is prompted, it will produce 1 unit of the differentiation product C. As an aside, the example shows some of the basic production rules and reaction schemes from which large-scale immune or nervous system models are typically composed.

The input and output neighborhoods were both the full Moore neighborhood. Automatic eigenlife decrement and global motion were set to "OFF". 100% of sites were prompted per time step. The spatial dynamic of the model is shown in figure 7.

As an afterthought, it is worth pointing out that this example also demonstrates an explicit relationship between SLANG production rules and systems of ordinary differential equations (modulo parameter functions) of the general form

$$F' = \text{production} - \text{decay} + \text{diffusion}$$

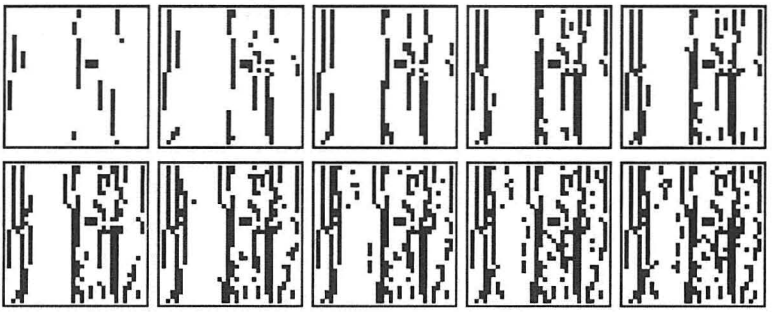


Figure 7: This figure shows 10 snapshots of the emerging patterns in the Z species layer taken at irregular time steps. We used a start-up density of 1% randomly placed initial seeds. Time evolution is shown from left to right. Characteristic for the dynamics of this system is the formation of long subtly branched fibers, which eventually form tangled network structures. Depending on the initial seeding density these structures become more, or less, pronounced. The predominantly "upward" growth direction is due to the algorithm that we use for placing products in the clockwise order C N NE E SE S SW W NW. Therefore, if C is occupied, the product will be placed at N; if C and N are occupied the object will with first preference be placed at NE; and so forth. We are presently extending this algorithm to include flexible placement by combining the "put" instruction with the "motion" instruction. This would make `>1®('Y',Y,3,12)` a syntactically correct statement where a member of species Y is placed into a random neighborhood site if this site is not already occupied. If, for example, `>2NE('X',X,1,2)` is used, then 2 units of species X are placed into any free location starting at NE. The statement `>2C('X',X,1,2)` is equivalent to the old `>2('X',X,1,2)`. In our present implementation, it is possible to program a directional put using a combination of non-directional ">" and "<" instructions. The idea is that "`>2...`, `<1...`" will place a product at NE, "`>3...`, `<2...`" at E, and so on. Unfortunately, placement using this approach will be precise if and only if all sites except the center are unoccupied (occupied sites will randomize placement). To obtain precision, which for certain models may be desirable, it is necessary to extend the language as suggested. It is clear that whichever type of placement the user decides to apply, the spatial dynamics of the model will always be a network-like pattern.

```
F:
  hunt:
  (#,F,1,#).[1,#](#,R,#,#)
    -> ((?,F,2,?+2);<1(#,R,#,#);)
    :  ((?,F,2,?);;) ;
  mate:
  (#,F,2,#).[1,#](#,F,#,#)
    -> ((?,F,1,?+1);>2(?,F,1,8);•)
    :  ((?,F,1,?);;•) ;

R:
  feed:
  (#,R,1,#) -> ((?,R,2,?+1);;);
  flight:
  (#,R,2,#).[1,#](#,F,#,#)
    -> ((?,R,3,?+2);;)
    :  ((?,R,3,?);;);
  mate:
  (#,R,3,#).[1,#](#,R,#,#)
    -> ((?,R,1,?+1);>4(?,R,1,20);•)
    :  ((?,R,1,?);;•);
```

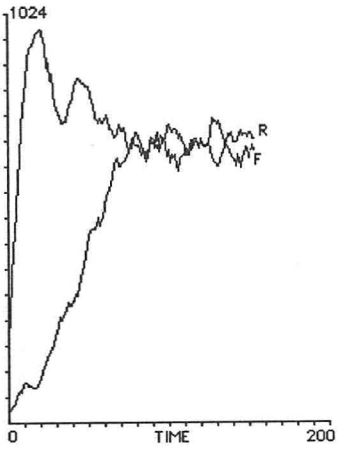


Figure 8: On the left of this figure we have listed the SLANG scripts for 2 quasi-single-sex species (F = "foxes" = predators, R = "rabbits" = prey). We allocated 2 behavioral states for the predator species, the first of which corresponds to "hunting" and the second to "mating". The prey species uses 3 states: the first corresponds to "feeding", the second to "flight", and the third to "mating". Successful execution of a behavior results in eigenlife increases in both species, where "attack" and "flight" are awarded with "$?+2$", and "feeding" and "mating" with "$?+1$". We did not go so far as to represent successless behaviors by a private form of stress that leads to an eigenlife decrease "$?-1$". Instead, the eigenlife value remains unchanged "?". The global eigenlife decrementer is set to "ON". In this demonstration experiment pelts are never used, and states are only used "internally" by the devices to order their own production rules, not "externally" by other devices to recognize them. If a fox or rabbit meets another member of its species, offspring are born. If there are as many males as females, the number of offspring will correspond roughly to 1/2 of the biological litter count, as we are pooling males and females in this model. Here, the rabbits' litter size is chosen to be twice that of the foxes. In a more elaborate set-up, we might use pelts to distinguish between female and male members of a species. This is interesting because we can pick up more dynamical fine structure than in the time plot shown on the right of the figure, since the survival of a species depends on the survival of sufficient numbers of both sexes. At the beginning of the experiment shown, rabbits (R) were introduced as a randomly distributed population with life expectancy of 20 per specimen populating 20% of their lattice world. 30 foxes (F) with life expectancy 8 were released from one location into this population. The time plot shows the dynamics of both populations for 160 time steps. The population densities are plotted on the vertical axis versus time on the horizontal axis.

which are frequently used in mathematical biology and biochemistry. In the system shown in figure 6, the equation for the species X, for example, would derive production terms from the concentration of S and from its own concentration, which is related to its replication rate. Decay occurs through the uptake by Z, and diffusion comes from the "®" instruction. Rate changes in the dynamics are obtained by changing the width of the recognition window, and/or the put and the delete rates in the production rule. It was previously indicated that cellular automata represent an alternative to, rather than an approximation of, differential equations in modeling physics [23]. Whether the relationship between differentiable systems and asynchronous CAs programmed with SLANG rules indicates a similar alternative for modeling biological systems requires further exploration.

5. *Predator/Prey Ecology.* In this example we consider a modified Volterra-Lotka model of a 1 preditor ("Foxes"), 1 prey ("Rabbits") ecological system. The general model definition is as follows. Reproduction is between any two specimens of the same species that are in the mating state. We model feeding through increases in the eigenlife value of a specimen. We assume, as in the classic Volterra-Lotka model [14, 26], that the rabbits have access to an unlimited food supply, and therefore their eigenlife is always increased. A fox, in turn, must hunt, and therefore must have a rabbit in its neighborhood if it is to prolong its eigenlife by feeding. The model assumes that a fox can only eat one rabbit "per prompt". Eigenlife also is increased if a fox or a rabbit mates, or if a rabbit escapes a hunting fox. This last case can be interpreted as an increase in fitness when a rabbit survives a chase. In this very simple model, the site selection algorithm determines the outcome of such a chase. If a fox is selected, one rabbit is caught; if a rabbit is selected, the rabbit escapes and its eigenlife is increased. Therefore the selection algorithm's numerical parameter(s) must thus encode the ratio of captures versus escapes. Each species is given more than one state (figure 8).

It should be noted that if the global eigenlife decrementer is turned "ON", preditor and prey specimen alike will survive only for the length of their initial eigenlife value (given upon birth). To counteract this global aging process we give each species in this CDM model a *repertoire of strategies*, which it can use to increase its eigenlife. For example, the rabbits can prolong their lives by feeding, by interacting with other rabbits (members of the opposite sex), or by surviving a chase by the predator.

5. Discussion

The Cellular Device Machine Development System is an example of a new breed of biologically motivated simulation software that has recently begun to emerge at the interface of mathematics, computer science, genetics, and cellular physiology. It was first conceived as a way of combining the advantages of autonomous, self-modifying, and mobile, finite automata ("devices")

that can be programmed using a dedicated object-based language, with those of a globally updated cellular automaton environment ("body"). In closing this paper, we would like to remark on the modifications to traditional cellular automaton (CA) algorithms that we use in our implementation, and on improvements to our system that are presently in progress.

The traditional CA implementation requires strict quasi-parallelism; that is, any update algorithm has to be equivalent to a fully parallel algorithm. In practice this translates to a procedure reminiscent of the "screen-swapping" technique used in computer games. No content of a cell may be altered until its value has been checked by all the other cells of which it is a neighbor. If memory space is not a problem, this is done by maintaining two or more copies of the lattice. The program reads the values of the selected cell's neighbors from the "old" copy and enters the new value for that cell into a buffer, or "new", copy. Only after each cell has been selected exactly once does the "new" array become the "old" array, and processing starts over. Since all updates thus become valid simultaneously at the end of a sequential "searchlight sweep", the order in which the cells are selected does not affect the result, and if the host processor allows parallel computation they could equally well be selected all at once ("floodlit"). While still using quasi-parallel updating, the CDM simulator program departs from the CA tradition by not selecting all of the cells in every time step but, more generally, by selecting a subset. While the subset is implicitly defined by prompting lattice sites according to a random sequence until a user-selected percentage of the lattice space is covered, the updating itself is deterministic.

Originally cellular automata were introduced as synchronously updated nearest-neighbor lattice dynamical systems (see [7, 5, 22] for overviews). Consequently, in most cellular automaton models studied elsewhere, some form of synchronous updating is applied. It was demonstrated previously that changing the update scheme can change the pattern-forming capacity of a cellular automaton [13]. As illustrated by our asynchronous Game of Life, apparent self-organization such as that found in gliders is critically dependent on synchronous updating. Another example is the "land developers' rule", which erodes soil only in the synchronous version [24, 6]. As we showed in section 4, other CA systems display similar patterns under either mode of updating (see Examples 1, 2, and 5). From a theoretical point of view, neither update scheme is "wrong", nor will it produce "wrong" dynamics; rather, if each is taken by itself, it may conceal structure-forming behaviors from the other. Based on this consideration and the observation that, in CA models of large biological systems, it is more realistic to think of synchronicity as a locally emerging non-stationary property of rule populations not initially locked in simultaneous conversation, we use strict asynchronous prompting of cell devices in the CDM simulator program. This choice is also consistent with our aim of retaining an object- and event-driven dynamic with as few arbitrary interventions from a global controller as possible. A variety of strategies for running efficient parallel simulations of asynchronous CAs are outlined in [15].

Going back to the future, we are now working on several improvements of the development system. First, a revision of the main data structures will enable us to keep all scripts in RAM. Second, we are developing some language extensions that reflect our past working experiences with SLANG, such as the use of rule nesting, of logical operators for connecting reference objects, of constants, and of macros. Third, we are working on a version of the CDM-DS that runs on electronic networks so that the sharing of SLANG script libraries and therefore of unambiguous and reproducible experiments becomes possible between remote users. The non-ASCII instruction symbols ®, ©, "¢", and "•" will be replaced by ASCII-compatible symbols. Fourth, development of more intrinsic data-analysis capabilities will enable the simulator to evaluate much of its own data. Finally, Using the flexible neighborhood representation by pointer sets we will be able to run a larger spectrum of user-defined local topologies on our system than just the input and output subsets of the Moore neighborhood.

Acknowledgments

This work is partially supported by NIMH grant R29 MH45688 to HBS. The authors are grateful for suggestions and criticism by Christa Müller-Sieburg, Ruth Williams, Hubert Halkin, Scott Kelley, and Cristoval Baray.

References

[1] A. O. Anderson, "Structure and Organization of the Lymphatic System," in *Immuno-physiology*, edited by J. J. Oppenheim and E. M. Shevach (Oxford, Oxford University Press, 1990).

[2] C. Baray and H. B. Sieburg, "An Intelligent Interface for a Cellular Automaton Simulator," research report funded under the Howard Hughes Medical Institute Academic Enrichment Program for Women and Minorities (1991); paper in preparation.

[3] C. Berge, *Graphs*, second edition (Amsterdam , North-Holland, 1985).

[4] J.-P. Boon and A. Noullez, "Development, Growth, and Form in Living Systems," in *On Growth and Form*, edited by H. E. Stanley and N. Ostrowsky (Dordrecht, Martinus Nijhoff Publishers, 1986).

[5] A. W. Burks, editor, *Essays on Cellular Automata* (Urbana, University of Illinois Press, 1970).

[6] O. K. Clay and H. B. Sieburg, "Generalized Asynchronous Cellular Automata"; paper in preparation.

[7] E. F. Codd, *Cellular Automata* (New York , Academic Press, 1968).

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

[9] L. Hayflick, "The Limited In Vitro Lifetime of Human Diploid Cell Strains," *Experimental Cell Research*, **37** (1965) 614–636.

[10] L. Hayflick and P. S. Moorhead, "The Serial Cultivation of Human Diploid Cell Strains," *Experimental Cell Research*, **25** (1961) 585–621.

[11] D. O. Hebb, *The Organization of Behavior* (New York, Wiley, 1949).

[12] J. H. Holland, *Adaptation in Natural and Artificial Systems* (Ann Arbor, The University of Michigan Press, 1975).

[13] T. E. Ingerson and R. L. Buvel, "Structure in Asynchronous Cellular Automata," *Physica D*, **10** (1984) 59–68.

[14] A. J. Lotka, *Elements of Physical Biology* (Baltimore, Williams and Wilkins, 1925). (Reissued as *Elements of Mathematical Biology* by Dover, 1956.)

[15] B. D. Lubachevsky, "Efficient Parallel Simulations of Asynchronous Cellular Arrays," *Complex Systems*, **1** (1987) 1099–1123.

[16] P. Meakin, "Computer Simulation of Growth and Aggregation Processes," in *On Growth and Form*, edited by H. E. Stanley and N. Ostrowsky (Dordrecht, Martinus Nijhoff Publishers, 1986).

[17] H. B. Sieburg, "A Logical Dynamic Systems Approach to the Regulation of Antigen-driven Lymphocyte Stimulation," in *Theoretical Immunology*, edited by A. S. Perelson, Santa Fe Institute Studies in the Sciences of Complexity, Proceedings volumes 2–3 (Reading, MA, Addison-Wesley, 1988).

[18] H. B. Sieburg, "The Cellular Device Machine: Point of Departure for Large-Scale Simulations of Complex Biological Systems," *Computers and Mathematics with Applications*, **20** (1990) 247–267.

[19] H. B. Sieburg, J. A. McCutchan, O. K. Clay, L. Caballero, and J. J. Ostlund, "Simulation of HIV Infection in Artificial Immune Systems," *Physica D*, **45** (1990) 208–227.

[20] H. B. Sieburg, "Physiological Studies in Silico," in *1990 Lectures in Complex Systems*, edited by L. Nadel and D. Stein, Santa Fe Institute Studies in the Sciences of Complexity, Lectures volume III (Reading, MA, Addison-Wesley, 1991).

[21] D. Stauffer and H. E. Stanley, *From Newton to Mandelbrot: A Primer in Theoretical Physics* (New York, Springer-Verlag, 1990).

[22] T. Toffoli, "Cellular Automata Mechanics" (Doctoral dissertation, The University of Michigan, 1977).

[23] T. Toffoli, "Cellular Automata as an Alternative to (Rather Than an Approximation of) Differential Equations in Modeling Physics," *Physica D*, **10** (1984) 195–204.

[24] T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modelling* (Cambridge, MA, MIT Press, 1987).

[25] S. M. Ulam, "On Some Mathematical Problems Connected with Patterns of Growth of Figures," *Proceedings of Symposia in Applied Mathematics (AMS)*, **14** (1962) 215–224.

[26] V. Volterra, "Variazioni e Fluttuazioni del Numero d'Individui in Specie Animali Convivienti," *Mem. Academia Lincei*, **2** (1926) 31–113.

[27] E. W. Dijkstra, *A Discipline of Programming* (Englewood Cliffs, NJ, Prentice-Hall, 1976).

[28] S. Wolfram, "Random Sequence Generation by Cellular Automata," *Advances in Applied Mathematics*, **7** (1986) 123–169.

[29] T. A. Witten and L. M. Sander, "Diffusion-limited Aggregation: A Kinetic Critical Phenomenon," *Physical Review Letters*, **47** (1981) 1400–1403.