

# A brief Cellang++ User Manual

Hao Zhu

Bioinformatics Section, Southern Medical University

Guangzhou, 510515, China

2018, 04, 20

1 Introduction .....	2
1.1 Cellular automata and biological modeling.....	2
1.2 Why object-oriented extensions .....	2
2 Language Summary.....	3
2.1 Lexical Elements .....	3
2.2 Syntax and Semantics.....	4
3 Input and Output .....	13
3.1 Input .....	13
3.2 Text output to screen and to files.....	14
3.3 Graphic outputs and hot keys .....	14
4 Commands and command line arguments .....	21
4.1 Model compilation .....	21
4.2 Running commands and arguments .....	22
4.3 Step-run mode .....	22
5 Programming features and issues .....	22
5.1 Cellular automata style multicellular modeling .....	22
5.2 Boundaries and boundary conditions .....	23
5.3 How to define signaling events .....	24
5.4 Capture of signaling events .....	25
5.5 Messages can be qualitative and quantitative .....	25
5.6 Heterogeneity and inhomogeneity .....	26
5.7 Runtime perturbation .....	26
5.8 Describe discrete (not continuous) molecule diffusion .....	26
5.9 Dell division .....	27
5.10 Numerical solution of differential equations .....	27
5.11 Modeling cells with multiple compartments .....	27
5.12 Display very small values of fields .....	28
6. Availability and bug reports .....	28

# 1 Introduction

## 1.1 Cellular automata and biological modeling

Cellular automata are a popular tool for the study of complex and dynamic systems. In addition to the classic rule-based cellular automata systems, there are other implementations, including language-based ones. A cellular automata language is a programming language; one can use it to realize various cellular automata rules. For a language-based cellular automata system, programs of encoded in the source language are usually translated into programs of an intermediate language (usually C or C++), and this feature makes language-based cellular automata systems flexible and extendable. *Cellang* was a cellular automata language developed by J. Dana Eckart in 1997 to realize various classic cellular automata rules.

A key feature of not only many physical systems but also many biological systems is that local simple interactions result in global complex patterns. For example, simple juxtacrine signaling among cells creates astonishing complex patterns on the tissue level. Potentially cellular automata are important for biological modeling, because in a metazoan the development and function of a cell depend on interactions with neighboring cells. Many researchers, by assuming a natural correspondence between a biological cell and an automata cell, have used cellular automata to model multicellular systems. However, compared with physical systems, biological systems have several unique features. The two most distinctive ones are heterogeneity and hierarchical structure. A tissue or organ contains cells of different types, these cells may change their types, states, and positions, and cells may undergo divisions. Within a cell, there are varied molecules and genes, which have different attributes and functions. Therefore, local simple interactions happen within and among cells on both the molecular and cell levels. These two features make it insensible to adopt a simple and direct mapping between biological cells and automata cells. Moreover, to examine biochemical reactions requires quantitative computing, which is overlooked by classic cellular automata. To build realistic multi-cellular biological models, extensions to classic cellular automata systems are needed.

## 1.2 Why object-oriented extensions

The attributes and functions of molecules and genes in a cell are highly heterogeneous. To facilitate describing them, it is reasonable to add object-oriented facilities into a cellular automata language to encapsulate these molecules and genes into objects. To simulate molecular signaling as message-passing is sensible for several reasons. First, in most situations molecular signaling is inferred upon epistasis analysis, and therefore indicates genetic interactions (including direct and indirect interactions between molecules) but not chemical reactions (direct interactions between molecules). For example, the true process of A activating B and A repressing B (symbolized as  $A \rightarrow B$  and  $A \vdash B$ ) may be via unknown intermediate players (such as C and D, making the full process to be  $A \rightarrow C \rightarrow D \rightarrow B$ ). Second, a direct activation of B by A can be implemented biochemically differently, including by phosphorylation, cleavage, or binding. Third, many biochemical reactions involve unknown quantitative parameters. Finally, interactions among molecules happen intermittently but

not constitutively. All of these make it helpful and necessary to use message-passing to model  $A \rightarrow B$  and  $A \rightarrow B$ .

Differential equations are widely used to implement quantitative computing in multicellular models, but equations actually make molecular interactions hardwired. In fact, molecular interactions are highly dynamic and context-dependent, a gene can be regulated by many transcription factors and a protein can interact with many other proteins. To faithfully describe the context-dependence, behaviors of molecules and cells should be flexibly programmed instead of hardwired. Upon these considerations, Cellang++ was developed; “++” means it is an extension of Cellang and it has object-oriented features. Cellang++ provides encapsulation of molecules, message passing between objects, quantitative computing of molecule concentration, function call, and rich, generic, and model-independent graphic outputs. In addition, cell division and movement can be implemented easily and simulated faithfully using the two new statements “copyto” and “moveto”. In the following parts, the natural correspondence between an automata cell and a biological cell and between an object and a molecule allows us simply to use “cell” and “object” indicate biological cells and molecules (including genes).

## 2 Language Summary

A Cellang++ model consists of four components: a 2D cell array, a Cellang++ program, an input, and some outputs. The 2D cell array and the input are usually combined in one input file, and the Cellang++ program is shared by all cells and consists of a description of cell attributes, a cell program, a message queue, and a set of object. Each object consists of a description of the molecule, a molecular program, and a message queue. The description of the cell and a molecule includes definition of the name, type, and value of attributes, and the cell program and each molecule program include loop, selection, assignment, send-messages, and so on. Variables (including simple variables and array variables) and their types (either integer or float) are implicitly declared when they firstly appear at the left side of an assignment. Many mathematics C functions can be called, and several new functions are implemented.

### 2.1 Lexical Elements

Lexical elements include: comments, reserved keywords, identifiers, operators, separators, and numeric literals.

Two C-type comments are allowed. The user can use a paired “/\*” and “\*/” to comment several lines or to use “//” to comment one line. A comment can appear anywhere within a program.

Keywords and identifiers are case insensitive, thus, **ELse** and **eLSe** both represent the same keyword **else**. The language has the following reserved keywords:

<b>const</b>	<b>copyto</b>	<b>dimensions</b>	<b>else</b>	<b>elsif</b>
<b>end</b>	<b>exit</b>	<b>for</b>	<b>forall</b>	<b>if</b>
<b>molecule</b>	<b>moveto</b>	<b>of</b>	<b>otherwise</b>	<b>sendmsg</b>
<b>then</b>	<b>when</b>			

Identifiers are names of variables, and must contain at least one character, and can be unlimited in length. Variables constructed from the first rule are ordinary ones, and variables begin with a “\_” are special ones. Special variables can only appear in messages and are not assignable.

identifier  $\rightarrow$  letter { underscore\_option letter\_or\_digit }  
 $\rightarrow$  underscore { letter underscore\_option letter\_or\_digit }

undersocre\_option  $\rightarrow$  underscore  
 $\rightarrow \lambda$

There are three predefined identifiers: “cell” (indicating the current cell), “time” (an integer, indicating the current time step), “rttime” (a floating point, indicating the current real time), and “msgq” (indicating the message queue). “time”, “rttime”, and “msgq” can only be used within expressions (in the right-hand side of an assignment and not assignable), and “cell” can be used within expressions (in the right hand side of assignments), be assigned new values (in the left hand side of assignments), and be used to refer to a cell with relative index [0,...,0]. The function “pos” (pos(1) and pos(2)) returns the position of the current cell in the 2D cell array.

The simple (one character) operators are:

\* / + - % = < > & | ! \$ (the message queue contains)

The compound (two characters) operators are:

<= >= != +% -% -> (object) !\$ (the message queue does not contain)

In some cases an explicit separator is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A separator is any of:

[ ] , : . := .. ( )

Space characters are also separators, except within a comment.

A numerical literal is expressed in the conventional integer decimal notation.

numerical\_literal  $\rightarrow$  digit { underscore\_option digit }

## 2.2 Syntax and Semantics

### (1) Constant Declarations

Constant declarations represent numeric values that do not change while the program runs. Their use supports improved readability and greater flexibility. A constant is either a numeric literal or an identifier which is the name of a previously declared simple constant. The size of an array constant is given by a constant, the value of which must be positive. The values of an array constant are given either explicitly as a list of signed constants, or implicitly as a default signed constant. If the array values are given explicitly, the number of values given must match the size of the array. The indexing of array elements begins with zero (0).

declare\_constant  $\rightarrow$  simple\_constant  
 $\rightarrow$  array\_constant

simple\_constant  $\rightarrow$  **const** identifier := signed\_constant

array\_constant → **const** identifier[] **for** constant  
                  := signed\_constant { , signed\_constant }

constant → numerical\_literal  
          → identifier

signed\_constant → sign\_option constant

sign\_option → +  
             → -  
             → λ

Examples:

**const** min\_value:=-15  
**const** max\_value:=-min\_value  
**const** size:=5  
**const** rates[] **for** 3 := 1,2,3  
**const** times[] **for** size:=1,2,3,4,5  
**const** param[] **for** 8:=1.1

## *(2) Cell Declaration*

The cell declaration describes the set of cells that make up the model. The declaration must appear only once in the program, and must precede any statements and object declarations, with the exception of constant declarations which may appear before the cell declaration.

cell\_declaration → constant **dimensions of** field\_list

field\_list → range  
           → { const\_option field } **end**

const\_option → **const**  
              → λ

field → const\_option field\_type **of** range

field\_type → array\_field  
           → ident\_list

array\_field → array\_name [] **for** constant  
             → λ

array\_name → identifier

range → signed\_constant .. signed\_constant

ident\_list → identifier { , identifier }

The constant appearing before “dimensions” specifies the number of dimensions (currently only 2D is used). The “field\_list” describes the attributes of each cell. An “ident\_list” allows a number of identifiers to be associated with a single range of value. When an “array\_field” is defined, the array name is used to collectively refer to a number of elements. The “constant” indicates the number of elements that the array field will contain. The indexing of array elements begins with zero (0).

The “const” keyword may only appear at the beginning of a cell field. If the “const” keyword is given, then the “ident\_list” or “array\_name” declared by the field is constant. Constant cell fields cannot have their value changed by any statement of the program. The value of a constant field may only be altered by reading new values from input. A range defines an inclusive set of integer or float values. The first is the lower bound and must be less than or equal to the second which is the upper bound of the range. A range indicates the permissible values that a field can be assigned. It is an error to assign a field a value outside of its associated range.

Examples:

**2 dimensions of 0..255**

```
const dims := 2
dims dimensions of
    const x, y of -9..9
    distance of 0.0..999.0
end
```

```
const size := 4
2 dimensions of
    count of 0..3
    const value[] for size of -9.0..9.0
end
```

### *(3) Object Declarations*

None or multiple objects can be defined between cell declaration and cell statements. As is seen, object fields are defined akin to cell fields, except that there is not dimension definition and there should be at least one explicitly defined field.

```
object_list → object { object }
            → λ
```

```
object → Molecule “{“ field_list statement_list “}”
```

```
field_list → { const_option field } end
```

```
statement_list → statement { statement }
```

### *(4) Expressions*

An expression is a formula that defines a computation.

expression → expression { binary\_op expression }  
→ unary\_op primary  
→ primary  
→ check\_msg  
→ function

unary\_op → +  
→ -  
→ !

binary\_op → +  
→ -  
→ \*  
→ \  
→ %  
→ &  
→ |  
→ =  
→ <  
→ >  
→ <=  
→ >=  
→ !=  
→ \$  
→ !\$

primary → index  
→ identifier obj\_reference field\_reference  
→ relative\_index obj\_reference field\_reference  
→ array\_reference obj\_reference field\_reference  
→ array\_constant  
→ ( expression )

relative\_index → [ index { , index } ]

array\_reference → identifier [ expression ]

field\_reference → .identifier  
→ .array\_reference  
→ λ

array\_constant → identifier [ expression ]

index → signed\_constant shift\_ops

→ index\_variable shift\_ops

shift\_ops → +% shift\_amount shift\_ops  
→ -% shift\_amount shift\_ops  
→ λ

shift\_amount → constant  
→ index\_variable

obj\_reference → -> identifier  
→ λ

check\_msg → **msgq** \$ message  
→ **msgq** !\$ message

function → identifier ( parameter\_list )

parameter\_list → parameter { , parameter }  
→ λ

parameter → identifier  
→ digit

message → relative\_index.( identifier, identifier, msg\_tag )  
→ identifier.( identifier, identifier, msg\_tag )

msg\_tag → identifier  
→ digit

The predefined identifier “time” indicates the current step of the execution; its initial value of is 0 and is increased by 1 each time all of the cells in the model have been updated. The precedence for the unary, binary and shift operators is as below, and operators belonging to the same level are evaluated from left to right.

- (1) +%    -%
- (2) ! (logical negation)
- (3) (unary) +    -
- (4) \*    / (quotient) % (remainder)
- (5) (binary) +    -
- (6) =    <    >    !=    <=    >=
- (7) & (logical and) | (logical or)

The relational and logical operations return integer values, with a zero (0) being returned when the condition is false and a one (1) when the condition is true. Unlike all of the other binary operations, the results of binary relational operations must not be immediately used by another binary relational



operation.

The quotient (/) and remainder (%) operators, when applied to a and b (e.g. a % b), yield the following results.

- (1) The quotient has the same sign as a \* b and of the two integer values on either side of the real valued result, its magnitude is the one whose value is closest to 0.
- (2) The remainder is given by  $a - b * (a / b)$  when b is non-negative and by  $a + b * (a / b)$  otherwise.

If an array reference or an array constant is used, then the identifier naming the array must be a previously declared. The range of the array is assumed to be zero (0) based, and the value of the expression must be a non-negative integer. If the expression used to reference an element of the array contains an index variable, then if the range of the index variable was not given by the “forall” loop the range of the index variable will be inferred to be the size of the array it is being used to reference. Inferred ranges are always zero (0) based. If the range of an index variable was not given in the “forall” loop, all of the inferred ranges must be the same.

The shift operators can only be applied to index variables of “forall” loops. The associated simple constant or index variable indicates the amount to be shifted. The shift operation performs a circular shift taking the index range associated with the index variable as a cyclic group. Thus the +% operator performs an addition modulo the size of the group and the -% operator performs a subtraction modulo the size of the group. The magnitude of the combined amounts of a sequence of circular shifts must be less than or equal to the range of the index variable being shifted.

When referencing a neighboring cell using a relative index, the number of indices given must match exactly the number of dimensions indicated by the cell declaration. Such a reference is taken relative to the current cell of computation. Since it is possible that adding the index to the current cell location might give an index value outside the range of that dimension, “wrap around” is used to insure that the reference is a legal one. Thus neighboring cell references can “wrap around” the edge of the cells of the automata. Furthermore, each relative index must have an absolute value no larger than the size of the corresponding dimension (as determined by the implementation).

#### *(5) Statements*

A statement defines an action to be performed; the process by which a statement achieves its action is called execution of the statement. Cellang programs may contain any number of statements.

```
statement → declare_constant
          → assignment_statement
          → if_statement
          → forall_statement
          → exit_statement
          → sendmsg
          → moveto
          → copyto
```

All of the statements in a *Cellang++* program are performed for each cell, once during each “time”

value.

#### *(6) Assignment Statements*

An assignment statement permits the value of the variable on the left hand side of the first := to be set or changed.

assignment\_statement → variable := expression\_list  
→ rest\_expression

rest\_expression → rest\_when  
→ λ

rest\_when → **when** expression := expression\_list rest\_when\_other  
→ **when** expression

rest\_when\_other → rest\_when  
→ **otherwise**

expression\_list → expression { , expression }

variable → identifier field\_reference  
→ array\_name [] field\_reference array\_for\_option  
→ array\_reference field\_reference

array\_for\_option → **for** numeric\_literal  
→ λ

If the identifier has not previously appeared on the left hand side of an assignment statement, then the identifier is implicitly declared as a variable. The variable is (statically) declared as either a single or multiple cell field variable depending upon the value(s) of the expression list and whether or not a field reference was given. This implies that the expression list value(s) of the assignment statement must be either all single field or all multi-field expressions. An expression list should contain multiple expressions only when the left hand side of the assignment statement is an unindexed array. Furthermore, the variable may contain at most one (1) unindexed array.

If the left hand side of the assignment constitutes the declaration of an array (by being the first lexical occurrence) then a non-empty array for option must be given. If the left hand side does not constitute a declaration then the array for option should be empty. When the array for option specifies a numeric literal, the numeric literal given indicates the number of elements that the array will contain.

The range of values that can be assigned to the field(s) of a variable is determined by the implementation, but must be at least as large as the union of all field ranges given in the cell declaration. Programs which assign the field(s) of variable values outside of its associated range are erroneous, though compilers are not required to generate run-time checks to detect such occurrences.

If a single expression (or a single expression followed by “otherwise”) appears to the right of the “:=” then that expression is assigned to the entity on the left hand side of the “:=”. If one or more “when” clauses appear then the expression to the right of “when” is evaluated first, if it is non-zero the value of the expression to the left of “when” is assigned and execution of the assignment statement is complete. In the case that the expression to the right of “when” evaluates to zero (0), then successive “when” clauses are tried in order. If all expressions to the right of “when” in all such clauses yield a zero (0) value and if an expression to the left of “otherwise” appears, its value is assigned. If none of the “when” conditions yield a non-zero value and if no “otherwise” appears, then the value is unchanged.

Assigning a value to “cell” causes the value of the current cell to become that value. This is the only way that the value of cells can be altered. Note, however, that that assignment(s) to “cell” determine the value that the current cell will have beginning with the next “time” value. Thus an assignment to “cell” will NOT alter the value that “cell” yields in an expression, for computations performed during the same “time” value. The variable “cell” is the only variable that expresses this dual nature of present versus future value.

If an array is given on the left hand side of the assignment, then an expression list containing one (1) or more expressions may be given on any of the right hand sides. The right hand sides need not agree in the number of expressions given, though if more than one (1) the number of expressions given in the expression list must be the same as the size of the array. If multiple expressions are given, then each is assigned to the corresponding array entry. If only one (1) expression is given, then *all* elements of the array are assigned that value.

Examples:

```
value := max_value - 1
```

```
cell := blue when time = 0
      := orange otherwise
```

```
maybe := 1 when random = x
        := 2 when random = y
```

```
max := north
max := south when south > max
max := east when east > max
neighbor[] for 4 := [0, 1], [0, -1], [1, 0], [-1, 0]
```

### (7) If Statement

An “if” statement selects for execution one or none of the enclosed statement lists, depending on the value of one or more conditions.

```
if_statement → if expression then statement_list
              { elsif expression then statement_list }
              else_option end
```

else\_option → **else** statement\_list  
                   → λ

For the execution of an “if” statement, the condition specified after “if” and any conditions specified after “elsif” are evaluated in succession (treating a final “else” as “elsif non-zero then”), until one evaluates to a non-zero value or all conditions are evaluated and yield zero (0) values. If a condition evaluates to nonzero, then the corresponding statement list is executed; otherwise none of the statement lists are executed.

Example:

```
if neighbor_count = 4 then
  cell := 0
elsif neighbor_count = 3 then
  cell := 1
  count := count + 1
else
  cell := 0
end
```

#### (8) Forall Statement

A “forall” statement is used to iterate through a range of values, usually to examine or change the identifiers associated with an array.

forall\_statement → **forall** index\_variable statement\_list

index\_variable → identifier index\_range

index\_range → : range  
                   → λ

The index variable is implicitly declared, but is only visible and usable within the statement list of the loop body. Index variables cannot be assigned.

If the index range is a range the index variable will take on the successive values of the range (i.e. sequential execution). If the index range is not given, then the index variable must be used as the index for an array and the size of the index range is inferred from that usage. The index variable can only be used apart from an index for an array when the index range has been given or can be inferred from such a usage. If inferred, all of the inferred sizes of the index range must be equal in size. If the index range was given, then the specified index range must be the same size as all inferred sizes of the index range. If the index range was not given, then its lower bound is taken as 0 and it must be possible to infer the size of the range from an array usage in the body of the loop.

Examples:

```
forall i:0..3
forall j
  sum[i] := (neighbor[j] = i+% 1) + sum[i]
```

```

end
end
forall i
    in := in + 1 when neighbor[i].speed[i +% 3]
end

```

#### (9) Exit Statement

An exit statement may only be used within a “forall” statement.

```
exit_statement → exit
```

The effect of performing an exit statement is to cause the immediately enclosing loop to terminate, transferring control to the point just after the end of the enclosing “forall” loop.

## 3 Input and Output

### 3.1 Input

An input file provides not only the geometric description of a model but also the initial values of some or all cell fields. The format of the file is like

```

0
[1,1]=1,2,3
[1,2]=2,3,4
100
[1,1]=10,20,30
[1,2]=20,30,40

```

The “0” in the first line means the following inputs are given at time step 0. “[1,1]” means cell at location [1,1], and the “1,2,3” give the initial value of the first, second, and third cell field. The “100” in the fourth line means the following inputs are given at time step 100, which enables the user to give run time perturbation to particular cells. If a perturbation (value=20) is given only to the third field in the cell at [20,20], the input can be “[20,20] = , , 20”. If not specified, the initial value of all fields in all cells is zero at time step 0.

The input file can be either edited using an editor or generated using a program. For example, the C program that produces a 2D 256 × 256 cell array with values of the first cell field (the “type” field) is like:

```

for (x=0; x<256; x++)
for (y=0; y<256; y++) {
    if (y>=100 && y<=120)
        type = 1;
    else
        type = 0;
    printf("[%d,%d]=%d\n", x, y, type);
}

```

### 3.2 Text output to screen and to files

There are two ways to output running results: text output (to the screen and/or files) and graphic output (to screen windows only). The standard (and simplest) text output has the same format as the input file:

```
0
[0,0]=0
[0,1]=1
[0,2]=0
[0,3]=3
[0,4]=0
[0,5]=2
[0,6]=0
[0,7]=1
[0,8]=0
[0,9]=0
```

The “0” in the first line means time step 0.

Using the argument “-outfile X” in the running command line, the values of, say, 20 chosen fields, can be outputted to 20 files whose names share the same prefix letter “X”.

### 3.3 Graphic outputs and hot keys

Multiple graphic outputs have been developed. To output the values of a field in a graphic window, the user should define a color map file for the field to allow different values to be displayed in different colors. A typical color map file is:

```
0..0  65535  65535  0
1..1  65535  0      65535
2..9  35535  35535  35535
10..10 0  35535  35535
```

This means value 0 is displayed in the color “65535 65535 0”, value 1 is displayed in the color “65535 0 65535”, value 2 to 9 are displayed in the color “35535 35535 35535”, and value 10 is displayed in the color “0 35535 35535”. If values of two fields are very different, two color map files are needed to graphically display them, or the user can choose to rescale the value of a field. Graphic windows can be captured using the GIMP software. In addition to the main window, other graphic windows can be opened and closed using the same hot keys at any time. Note that before using hot keys to open graphic windows, to use the left mouse button to make a click in the main window is required. For windows displaying data in a cell or a line of cells, this click chooses a specific cell (windows opened using “e/E”, “n/N”, “1”, “2”, “a”, “b”, “c”, “d”, and “7”) or a specific line of cells (windows opened using “5” and “6”).

*(1) The “e/E” window.*

The hot key “e” opens a window displaying all fields and all signaling events in the cell the mouse click is made, and the hot key “E” opens 9 windows displaying the cell and its 8 neighboring cells

(Figure 1). After the “e/E” window(s) is opened, a mouse click in the main window changes the cell that is displayed.






 .myexe — X	 .myexe — X	 .myexe — X	 .myexe — X	 .myexe — X
pouch =0.000000 FzB_Act_Htm_0_0 =0	pouch =0.000000 FzB_Act_Htm_0_0 =0	pouch =0.000000 FzB_Act_Htm_0_0 =0	pouch =0.000000 FzB_Act_Htm_0_0 =0	pouch =0.000000 FzB_Act_Htm_0_0 =0
randd =0.760737 FzB_Act_Vg_0_0 =0	randd =0.724901 FzB_Act_Vg_0_0 =0	randd =0.083564 FzB_Act_Vg_0_0 =0	randd =0.913336 FzB_Act_Vg_0_0 =0	randd =0.877660 FzB_Act_Vg_0_0 =0
birth =17.241150 FzB_Act_X_0_0 =0	birth =28.416296 FzB_Act_X_0_0 =0	birth2 =29.489582 FzB_Act_X_0_0 =0	birth =34.300579 FzB_Act_X_0_0 =0	birth =14.741676 FzB_Act_X_0_0 =0
birth2 =0.172412 Ntm_Rep_FzB_0_0 =0	birth2 =0.284163 Ntm_Rep_FzB_0_0 =0	birth2 =0.294896 Ntm_Rep_FzB_0_0 =0	birth2 =0.343006 Ntm_Rep_FzB_0_0 =0	birth2 =0.147417 Ntm_Rep_FzB_0_0 =0
D_all =0.974585 X_Rep_Vg_0_0 =0	D_all =0.974585 X_Rep_Vg_0_0 =0	D_all =0.974585 X_Rep_Vg_0_0 =0	D_all =0.974585 X_Rep_Vg_0_0 =0	D_all =0.974585 X_Rep_Vg_0_0 =0
De_all =0.974940 Vg_Act_Vg_0_0 =0	De_all =0.974940 Vg_Act_Vg_0_0 =0	De_all =0.974940 Vg_Act_Vg_0_0 =0	De_all =0.974940 Vg_Act_Vg_0_0 =0	De_all =0.974940 Vg_Act_Vg_0_0 =0
D_pol =0.000000 Vg_Act_Fj_0_0 =0	D_pol =0.000000 Vg_Act_Fj_0_0 =0	D_pol =0.000000 Vg_Act_Fj_0_0 =0	D_pol =0.000000 Vg_Act_Fj_0_0 =0	D_pol =0.000000 Vg_Act_Fj_0_0 =0
stress =0.000000 Vg_Rep_Ds_P_0_0 =0	stress =0.000000 Vg_Rep_Ds_P_0_0 =0	stress =0.000000 Vg_Rep_Ds_P_0_0 =0	stress =0.000000 Vg_Rep_Ds_P_0_0 =0	stress =0.000000 Vg_Rep_Ds_P_0_0 =0
divnum =0.000000 Vg_Rep_Ds_D_0_0 =0	divnum =0.000000 Vg_Rep_Ds_D_0_0 =0	divnum =0.000000 Vg_Rep_Ds_D_0_0 =0	divnum =0.000000 Vg_Rep_Ds_D_0_0 =0	divnum =0.000000 Vg_Rep_Ds_D_0_0 =0
divrate =0.000000 F7_P_Ubq_D_P_0_0 =0	divrate =0.000000 F7_P_Ubq_D_P_0_0 =0	divrate =0.000000 F7_P_Ubq_D_P_0_0 =0	divrate =0.000000 F7_P_Ubq_D_P_0_0 =0	divrate =0.000000 F7_P_Ubq_D_P_0_0 =0
Ug.v =0.000000 F7_P_Ubq_Ds_P_0_0 =0	Ug.v =0.000000 F7_P_Ubq_Ds_P_0_0 =0	Ug.v =0.000000 F7_P_Ubq_Ds_P_0_0 =0	Ug.v =0.000000 F7_P_Ubq_Ds_P_0_0 =0	Ug.v =0.000000 F7_P_Ubq_Ds_P_0_0 =0
Ug.v10 =0.000000 F7_P_Ubq_DBs_P_0_0 =0	Ug.v10 =0.000000 F7_P_Ubq_DBs_P_0_0 =0	Ug.v10 =0.000000 F7_P_Ubq_DBs_P_0_0 =0	Ug.v10 =0.000000 F7_P_Ubq_DBs_P_0_0 =0	Ug.v10 =0.000000 F7_P_Ubq_DBs_P_0_0 =0
Fz.v =0.020000 F7_D_Ubq_D_D_0_0 =0	Fz.v =0.020000 F7_D_Ubq_D_D_0_0 =0	Fz.v =0.020000 F7_D_Ubq_D_D_0_0 =0	Fz.v =0.020000 F7_D_Ubq_D_D_0_0 =0	Fz.v =0.020000 F7_D_Ubq_D_D_0_0 =0
FzB.v =0.000000 F7_D_Ubq_Ds_D_0_0 =0	FzB.v =0.000000 F7_D_Ubq_Ds_D_0_0 =0	FzB.v =0.000000 F7_D_Ubq_Ds_D_0_0 =0	FzB.v =0.000000 F7_D_Ubq_Ds_D_0_0 =0	FzB.v =0.000000 F7_D_Ubq_Ds_D_0_0 =0
FzB.v10 =0.000000 F7_D_Ubq_DBs_D_0_0 =0	FzB.v10 =0.000000 F7_D_Ubq_DBs_D_0_0 =0	FzB.v10 =0.000000 F7_D_Ubq_DBs_D_0_0 =0	FzB.v10 =0.000000 F7_D_Ubq_DBs_D_0_0 =0	FzB.v10 =0.000000 F7_D_Ubq_DBs_D_0_0 =0
Ntm.v =0.000000 D_P_Rep_Uts_0_0 =1	Ntm.v =0.000000 D_P_Rep_Uts_0_0 =1	Ntm.v =0.000000 D_P_Rep_Uts_0_0 =1	Ntm.v =0.000000 D_P_Rep_Uts_0_0 =1	Ntm.v =0.000000 D_P_Rep_Uts_0_0 =1
X.v =0.000000 Uts_Rep_Yki_0_0 =1	X.v =0.000000 Uts_Rep_Yki_0_0 =1	X.v =0.000000 Uts_Rep_Yki_0_0 =1	X.v =0.000000 Uts_Rep_Yki_0_0 =1	X.v =0.000000 Uts_Rep_Yki_0_0 =1
Vg.v =0.000000 Yki_Act_Vg_0_0 =1	Vg.v =0.000000 Yki_Act_Vg_0_0 =1	Vg.v =0.000000 Yki_Act_Vg_0_0 =1	Vg.v =0.000000 Yki_Act_Vg_0_0 =1	Vg.v =0.000000 Yki_Act_Vg_0_0 =1
Fj.v =0.000000 Yki_Act_Uts_0_0 =1	Fj.v =0.000000 Yki_Act_Uts_0_0 =1	Fj.v =0.000000 Yki_Act_Uts_0_0 =1	Fj.v =0.000000 Yki_Act_Uts_0_0 =1	Fj.v =0.000000 Yki_Act_Uts_0_0 =1
Ds.P.v =0.487377 cell_Sts_Uts_n1_n1 =0	Ds.P.v =0.487377 cell_Sts_Uts_n1_n1 =0	Ds.P.v =0.487377 cell_Sts_Uts_n1_n1 =0	Ds.P.v =0.487377 cell_Sts_Uts_n1_n1 =0	Ds.P.v =0.487377 cell_Sts_Uts_n1_n1 =0
Ds.D.v =0.487377 cell_Sts_Uts_n1_0 =0	Ds.D.v =0.487377 cell_Sts_Uts_n1_0 =0	Ds.D.v =0.487377 cell_Sts_Uts_n1_0 =0	Ds.D.v =0.487377 cell_Sts_Uts_n1_0 =0	Ds.D.v =0.487377 cell_Sts_Uts_n1_0 =0
Ft.P.v =0.489868 cell_Sts_Uts_n1_p1 =0	Ft.P.v =0.489868 cell_Sts_Uts_n1_p1 =0	Ft.P.v =0.489868 cell_Sts_Uts_n1_p1 =0	Ft.P.v =0.489868 cell_Sts_Uts_n1_p1 =0	Ft.P.v =0.489868 cell_Sts_Uts_n1_p1 =0
Ft.D.v =0.489868 cell_Sts_Uts_0_n1 =0	Ft.D.v =0.489868 cell_Sts_Uts_0_n1 =0	Ft.D.v =0.489868 cell_Sts_Uts_0_n1 =0	Ft.D.v =0.489868 cell_Sts_Uts_0_n1 =0	Ft.D.v =0.489868 cell_Sts_Uts_0_n1 =0
F7.P.v =0.489868 cell_Sts_Uts_0_0 =0	F7.P.v =0.489868 cell_Sts_Uts_0_0 =0	F7.P.v =0.489868 cell_Sts_Uts_0_0 =0	F7.P.v =0.489868 cell_Sts_Uts_0_0 =0	F7.P.v =0.489868 cell_Sts_Uts_0_0 =0
F7.D.v =0.489868 cell_Sts_Uts_0_p1 =0	F7.D.v =0.489868 cell_Sts_Uts_0_p1 =0	F7.D.v =0.489868 cell_Sts_Uts_0_p1 =0	F7.D.v =0.489868 cell_Sts_Uts_0_p1 =0	F7.D.v =0.489868 cell_Sts_Uts_0_p1 =0
F4F7.P.v=0.010130 cell_Sts_Uts_p1_n1 =0	F4F7.P.v=0.010130 cell_Sts_Uts_p1_n1 =0	F4F7.P.v=0.010130 cell_Sts_Uts_p1_n1 =0	F4F7.P.v=0.010130 cell_Sts_Uts_p1_n1 =0	F4F7.P.v=0.010130 cell_Sts_Uts_p1_n1 =0
F4F7.D.v=0.010130 cell_Sts_Uts_p1_0 =0	F4F7.D.v=0.010130 cell_Sts_Uts_p1_0 =0	F4F7.D.v=0.010130 cell_Sts_Uts_p1_0 =0	F4F7.D.v=0.010130 cell_Sts_Uts_p1_0 =0	F4F7.D.v=0.010130 cell_Sts_Uts_p1_0 =0
D.P.v =0.487198 cell_Sts_Uts_p1_p1 =0	D.P.v =0.487198 cell_Sts_Uts_p1_p1 =0	D.P.v =0.487198 cell_Sts_Uts_p1_p1 =0	D.P.v =0.487198 cell_Sts_Uts_p1_p1 =0	D.P.v =0.487198 cell_Sts_Uts_p1_p1 =0
D.D.v =0.487198 cell_Sts_cell_n1_n1=0	D.D.v =0.487198 cell_Sts_cell_n1_n1=0	D.D.v =0.487198 cell_Sts_cell_n1_n1=0	D.D.v =0.487198 cell_Sts_cell_n1_n1=0	D.D.v =0.487198 cell_Sts_cell_n1_n1=0
DDs.P.v =0.010049 cell_Sts_cell_n1_0 =0	DDs.P.v =0.010049 cell_Sts_cell_n1_0 =0	DDs.P.v =0.010049 cell_Sts_cell_n1_0 =0	DDs.P.v =0.010049 cell_Sts_cell_n1_0 =0	DDs.P.v =0.010049 cell_Sts_cell_n1_0 =0
DDs.D.v =0.010049 cell_Sts_cell_n1_p1=0	DDs.D.v =0.010049 cell_Sts_cell_n1_p1=0	DDs.D.v =0.010049 cell_Sts_cell_n1_p1=0	DDs.D.v =0.010049 cell_Sts_cell_n1_p1=0	DDs.D.v =0.010049 cell_Sts_cell_n1_p1=0
Uts.v =0.489808 cell_Sts_cell_0_n1 =0	Uts.v =0.489808 cell_Sts_cell_0_n1 =0	Uts.v =0.489808 cell_Sts_cell_0_n1 =0	Uts.v =0.489808 cell_Sts_cell_0_n1 =0	Uts.v =0.489808 cell_Sts_cell_0_n1 =0
Uts.sts =0.000000 cell_Sts_cell_0_0 =0	Uts.sts =0.000000 cell_Sts_cell_0_0 =0	Uts.sts =0.000000 cell_Sts_cell_0_0 =0	Uts.sts =0.000000 cell_Sts_cell_0_0 =0	Uts.sts =0.000000 cell_Sts_cell_0_0 =0
Yki.v =0.489848 cell_Sts_cell_0_p1 =0	Yki.v =0.489848 cell_Sts_cell_0_p1 =0	Yki.v =0.489848 cell_Sts_cell_0_p1 =0	Yki.v =0.489848 cell_Sts_cell_0_p1 =0	Yki.v =0.489848 cell_Sts_cell_0_p1 =0
cell_Sts_cell_p1_n1=0	cell_Sts_cell_p1_n1=0	cell_Sts_cell_p1_n1=0	cell_Sts_cell_p1_n1=0	cell_Sts_cell_p1_n1=0
cell_Sts_cell_p1_0 =0	cell_Sts_cell_p1_0 =0	cell_Sts_cell_p1_0 =0	cell_Sts_cell_p1_0 =0	cell_Sts_cell_p1_0 =0
cell_Sts_cell_p1_p1=0	cell_Sts_cell_p1_p1=0	cell_Sts_cell_p1_p1=0	cell_Sts_cell_p1_p1=0	cell_Sts_cell_p1_p1=0
<41,40>(<1,0> T=0, 0208	<39,40>(<-1,0> T=0, 0208	<38,40>(<-2,0> T=0, 0208	<42,40>(<2,0> T=0, 0208	<36,40>(<-4,0> T=0, 0208

Figure 1. The E windows opened using the hot key “E” to display all fields and all signaling events in 9 cells.

## (2) The “n/N” window.

The hot key “n” opens a window displaying values of the chosen fields in the cell the mouse click is made and in X layers of neighboring cells, and the hot key “N” opens a set of windows displaying values of all chosen fields in the cell the mouse click is made and in X layers of neighboring cells (Figure 2). After the “n/N” window(s) is opened, make a mouse click in the n window or the leading window of N windows (the window displaying the value of the “-f” field) following the hot key “1” to “9”, the layer of neighborhood (thus the size of the windows) can be changed. The “e/E” and “n/N” windows are useful for model debugging.

.m - □ ×	.m - □ ×	.m - □ ×	.m - □ ×	.m - □ ×
0.2860 0.2860 0.2860	0.0000 0.0000 0.0000	0.1815 0.1815 0.1815	0.5628 0.5628 0.5628	0.0000 0.0000 0.0000
0.2860 0.2860 0.2860	0.0000 0.0000 0.0000	0.1815 0.1815 0.1815	0.5628 0.5628 0.5628	0.0000 0.0000 0.0000
0.2860 0.2860 0.2860	0.0000 0.0000 0.0000	0.1815 0.1815 0.1815	0.5628 0.5628 0.5628	0.0000 0.0000 0.0000
<33,21> Yki.v T=	<33,21> Vg.v T=	<33,21> Ftf7_D.v T=	<33,21> D_a11 T=	<33,21> pouch T=
.m - □ ×	.m - □ ×	.m - □ ×	.m - □ ×	.m - □ ×
0.0000 0.0000 0.0000	0.0000 0.0000 0.0000	0.2802 0.2802 0.2802	0.3185 0.3185 0.3185	0.2848 0.2848 0.2848
0.0000 0.0000 0.0000	0.0000 0.0000 0.0000	0.2802 0.2802 0.2802	0.3185 0.3185 0.3185	0.2848 0.2848 0.2848
0.0000 0.0000 0.0000	0.0000 0.0000 0.0000	0.2802 0.2802 0.2802	0.3185 0.3185 0.3185	0.2848 0.2848 0.2848
<33,21> diorate T=	<33,21> Fj.v T=	<33,21> D_P.v T=	<33,21> Ft_P.v T=	<33,21> Ds_D.v T=
.m - □ ×	.m - □ ×	.m - □ ×	.m - □ ×	.m - □ ×
0.0000 0.0000 0.0000	0.2848 0.2848 0.2848	0.2802 0.2802 0.2802	0.1490 0.1490 0.1490	0.1490 0.1490 0.1490
0.0000 0.0000 0.0000	0.2848 0.2848 0.2848	0.2802 0.2802 0.2802	0.1490 0.1490 0.1490	0.1490 0.1490 0.1490
0.0000 0.0000 0.0000	0.2848 0.2848 0.2848	0.2802 0.2802 0.2802	0.1490 0.1490 0.1490	0.1490 0.1490 0.1490
<33,21> diorate T=	<33,21> Ds_P.v T=	<33,21> D_D.v T=	<33,21> DDs_D.v T=	<33,21> DDs_P.v T=

Figure 2. The N windows opened using the hot key “N” to display the value of all fields in a local area.

### (3) The single cell windows.

The hot keys “1”, “2”, “a”, “b”, “c”, and “d” can open up to 6 “cell window” to display values of chosen fields and all signaling events in a time period (from the time step the window is opened to the time step it is closed). First, use the left mouse button to make a click at a specific position in the main window; second, press “1” to open a pair of cell window, one displaying values of chosen fields and the other displaying all signaling events. After 30 seconds, one can use the left mouse button to make a click at another position in the main window, and press “2” (and “a”, “b”, “c”, and “d”) to open another pair of cell window (Figure 3AB).



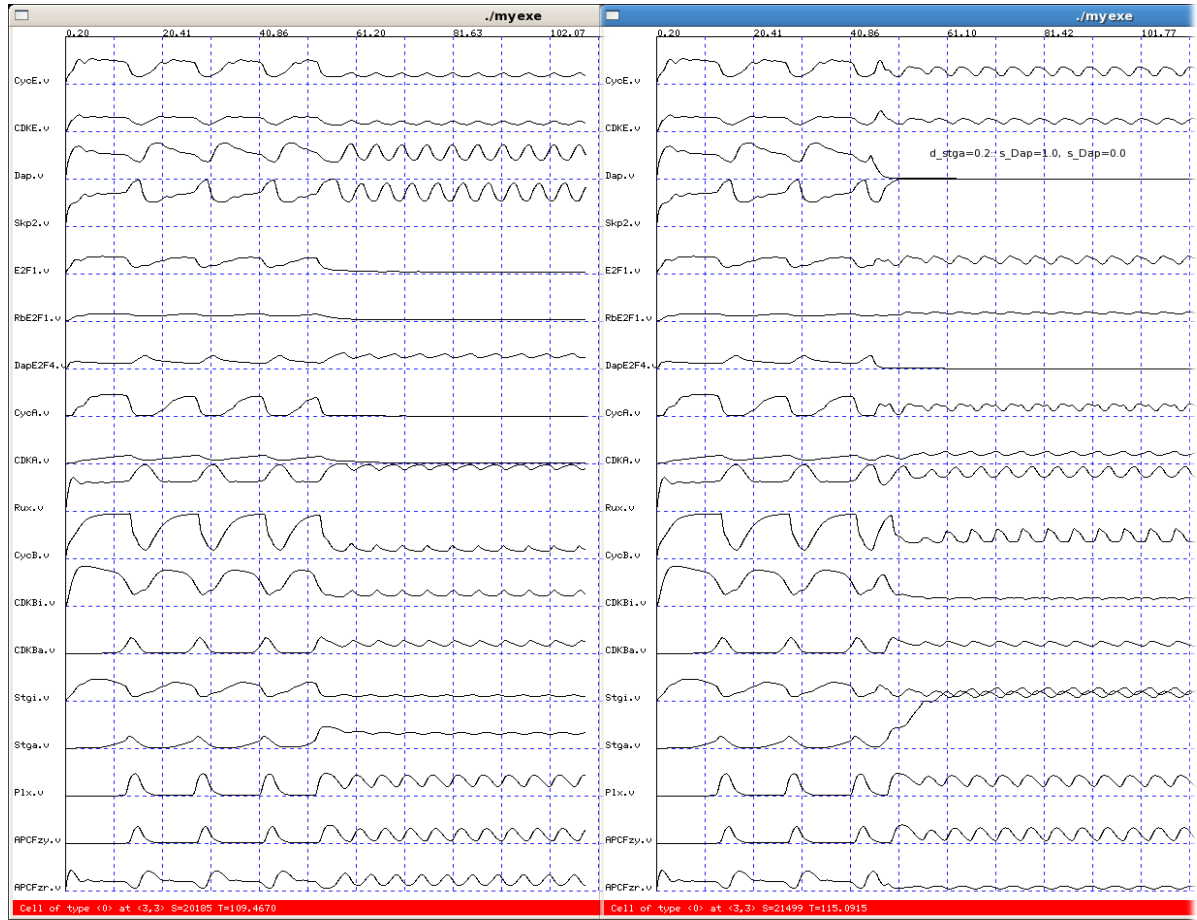


Figure 3A. The window opened using the hot keys “1”, “2”, “a”, “b”, “c”, or “d” to display the value of chosen fields from time  $i$  to time  $j$  in a specific cell (shown here are two windows).

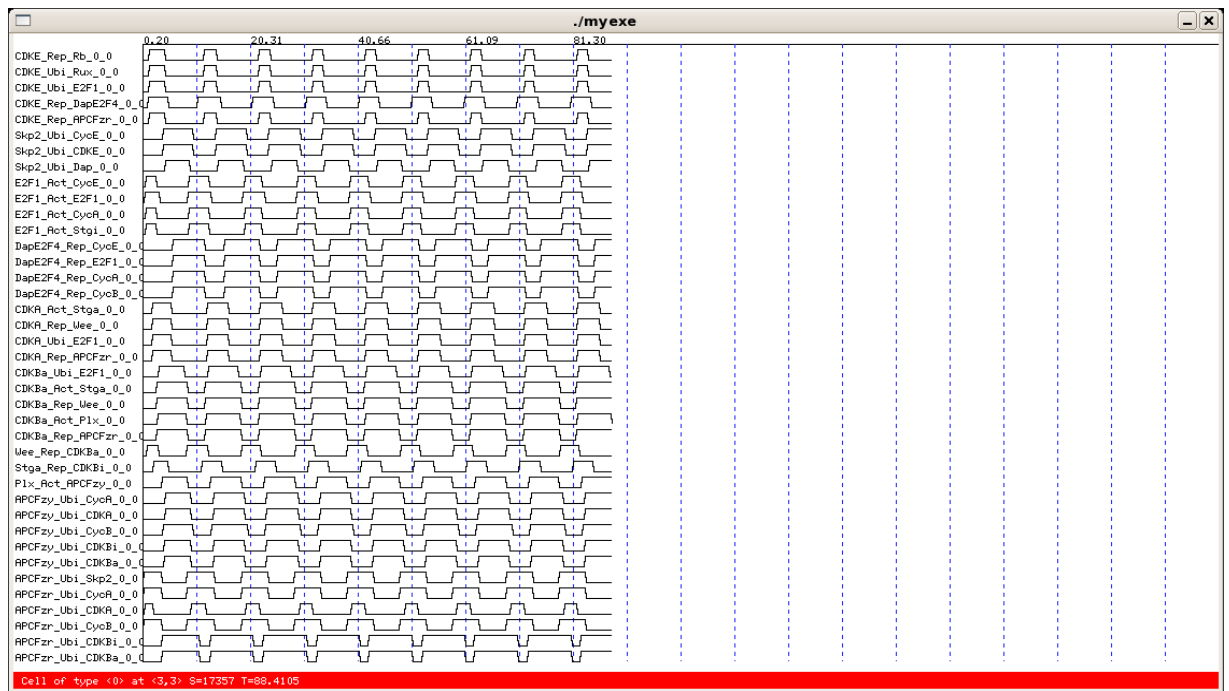


Figure 3B. The window opened using the hot keys “1”, “2”, “a”, “b”, “c”, or “d” to display all signaling

events from time  $i$  to time  $j$  in a specific cell.

(4) *The 2D tissue windows.*

The hot key “3” opens a set of windows displaying the current values of all chosen fields in the whole cell space (Figure 4).

(5) *The 2D tissue window.*

The hot key “4” opens a set of windows displaying the incidence/absence of all signaling events in the whole cell space (Figure 5). If  $M$  events are defined in all, the hot key “4” opens  $M$  windows.

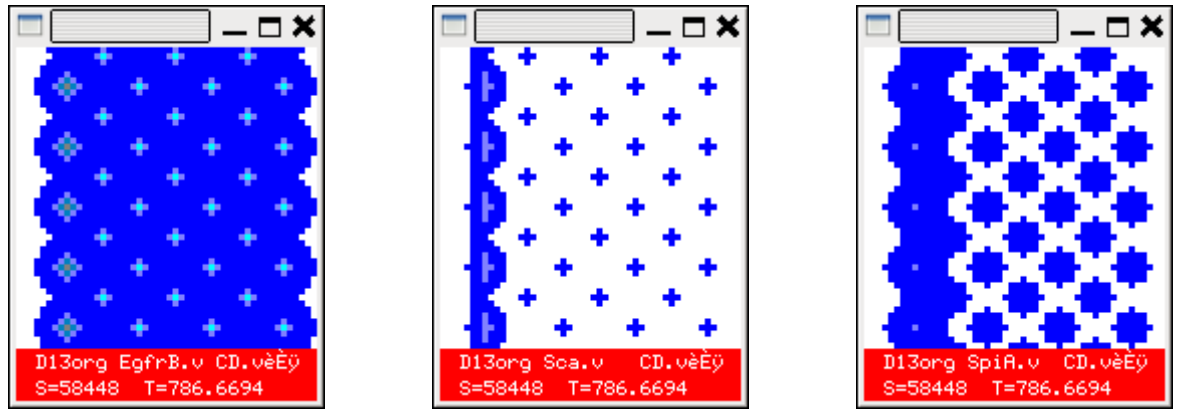


Figure 4. The window opened using the hot keys “3” to display the distribution of values of chosen fields at a time step in all cells (shown here are the concentrations of three molecules). For example, the command “./myexe -s 2 -f 10 -fields 1, 4, 5, 6, 7, 8, 9, 11, 18, 19, 21, 23, 25, 27, 29, 32, 34 -map colormap -dispstep 0.3 < cellarray164” allows this hot key to open 17 windows displaying the value of field 1, 4, 5, 6, 7, 8, 9, 11, 18, 19, 21, 23, 25, 27, 29, 32, and 34 (automatically and implicitly numbered from the first field to the last field).

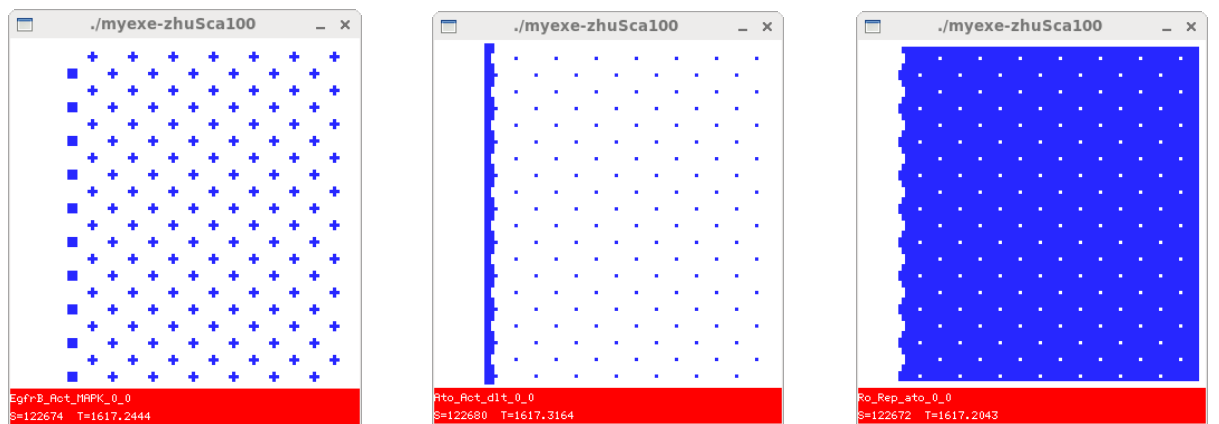


Figure 5. The window opened using the hot keys “4” to display the distribution of all signaling events at a time step in the cell space (shown here are the distributions of three signaling events).

*(6) The 1D tissue window.*

First, use the left mouse button to make a click at a specific position in the main window; second, use the hot key "5" opens a window displaying the current values of the chosen fields in the chosen line of cells ([Figure 6](#)).

*(7) The 1D tissue window.*

First, use the left mouse button to make a click at a specific position in the main window; second, use the hot key "6" opens a window displaying the occurrence/absence of all signaling events in the chosen line of cells ([Figure 7](#)).

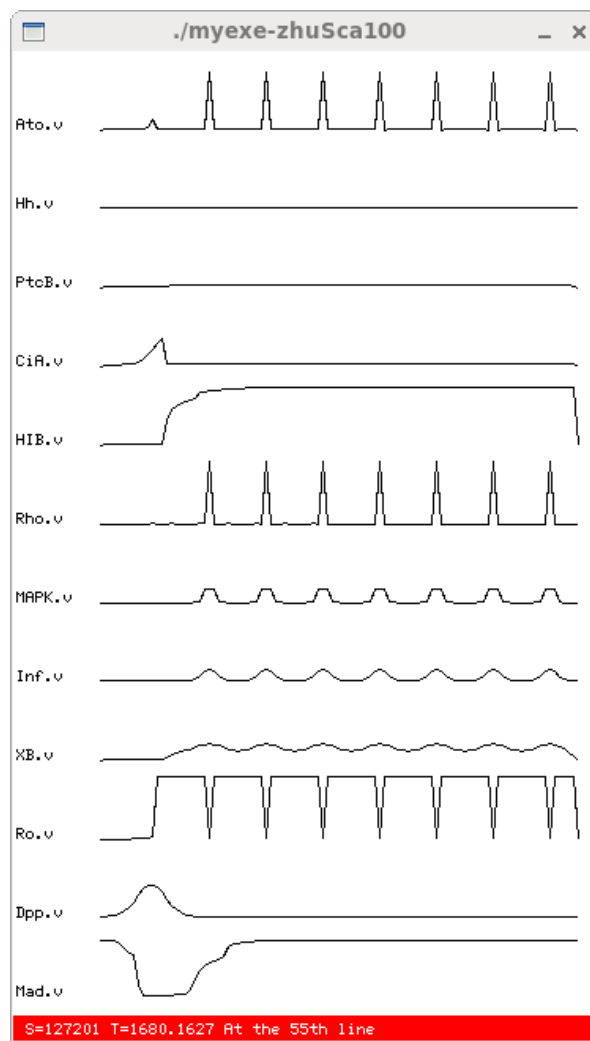


Figure 6. The window opened using the hot key "5" to display the value of chosen fields at a time step in a line of cells.

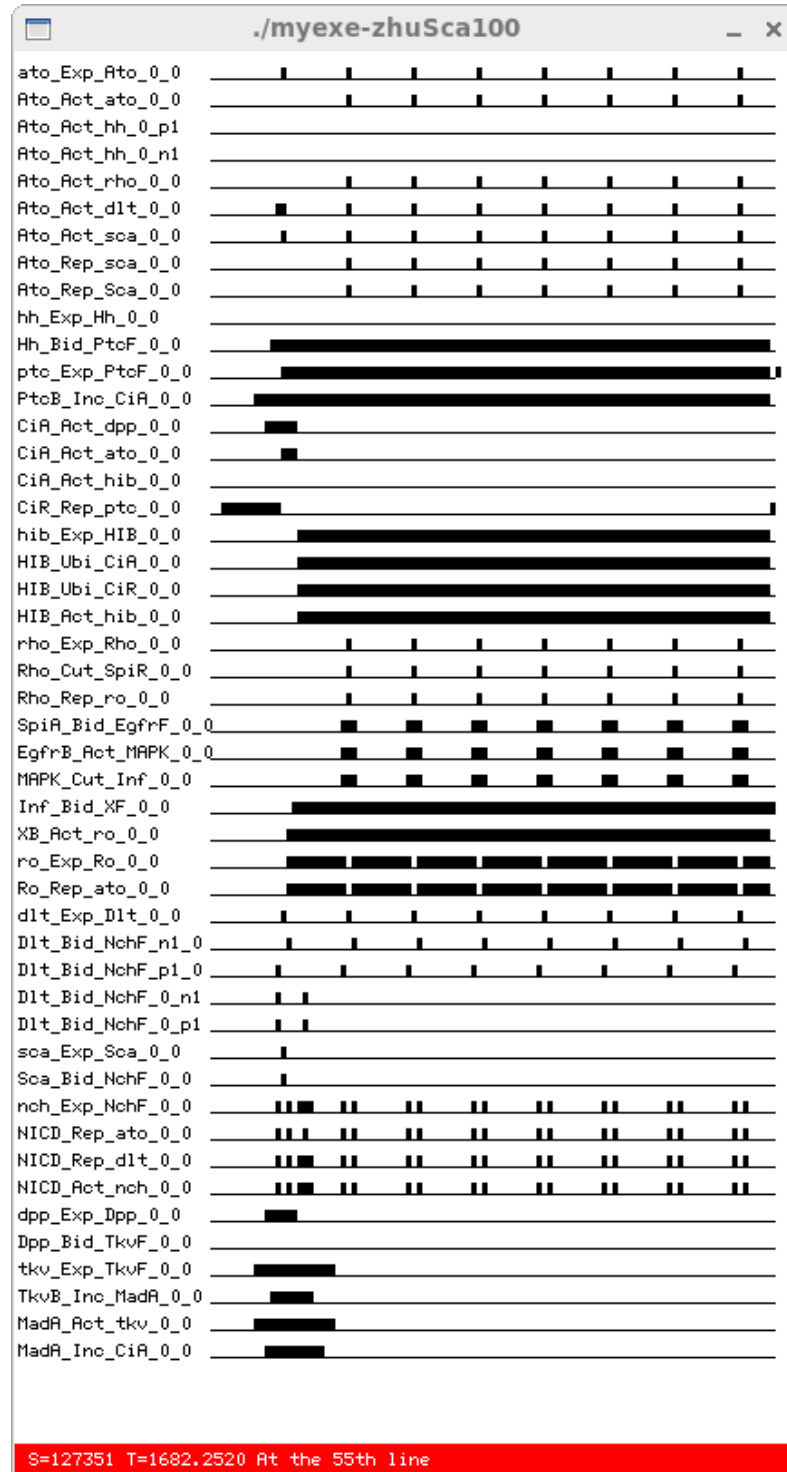


Figure 7. The window opened using the hot key “6” to display all signaling events at a time step in a line of cells. Shaded parts indicate the occurrence of the event.

#### (8) The network window.

Captured signaling events at any time step  $t$  can be reconstructed into an emergent signaling network in the form of adjacency matrix. In the graph  $G = (V, E)$  that represents a network,  $V$  indicates molecules and  $E$  indicates message passing among molecules.  $V$  is alphabetically ordered on the name of molecules as  $V = \{v_1, v_2, \dots, v_n\}$  and  $E = \{(v_i, v_j) \mid i, j \in V\}$ . Thus in the adjacency matrix  $\mathbf{M} = [m_{ij}]$ ,



zhuvview.o zhux11.o

Here “-g” means the codes allow debugging using a tool such as DDD.

## 4.2 Running commands and arguments

### (1) Run without output to files

```
./myexe -s 2 -f 10 -fields 1, 2 -map colormap [-dispstep 0.3] < cellarray124
```

“-s 2” indicates to use 2x2 screen pixels to show a cell, “-f 10” indicates the main window displays values of the 10th field, “-fields 1, 2” indicates values of the 1st and 2nd field are outputted, “-map colormap” indicates the file “colormap” is the color map, “-dispstep 0.3” is optional and indicates that values of the chosen fields are sampled at the time interval 0.3 (rttime) (without this argument, values of the chosen fields are sampled at every time step), “<” is a pipeline to accept the input file “cellarray124” that describes the 2D cell array and initial values in each cell.

To enable graphic output of more fields, the command can be:

```
./myexe -s 2 -f 10 -fields 1, 2, 4,5,6,7,8,9,11,18,19,21,23,25,27,29,32,34 -map colormap  
< cellarray124
```

### (2) Run with output to files

```
./myexe -s 2 -f 10 -fields 1,4,5,6,7,8,9,11,18,19,21,23,25,27,29,32,34 -map colormap  
-outfile X [-outstep 0.3] < cellarray124
```

“-outfile X” indicates that values of the chosen fields are outputted to files whose names begin with the prefix “X” (the field names are file names), “-outstep 0.3” is optional and indicates that values of the chosen fields are sampled at the time interval 0.3 (rttime) (without this argument values of the chosen fields are sampled at every time step). Note that the argument “-outstep 0.3” should be used, otherwise the output files will be too large.

### (3) Run in debugging model

```
DDD myexe  
(in DDD) run -s 2 -f 10 -map colormap < cellarray124
```

## 4.3 Step-run mode

The hot key “Shift + s” can be used to make running pause and to continue. In the “pause” state, the user can capture screen windows, and can also use the “s” key to make step running (one press of “s” key performs one time step).

## 5 Programming features and issues

### 5.1 Cellular automata style multicellular modeling

To build multicellular models using C/C++ or Matlab, the codes describing computation and model geometry are intertwined. Since the codes of data array operations scatter everywhere in a model, they are an important source of bugs and errors, especially if a model undergoes frequent revisions. The Cellang++ system provides a cellular automata style modeling paradigm that separates computational modeling from geometric modeling; this feature greatly facilitates investigating

models of different sizes and shapes.

The Cellang++ system can be used to build multiple kinds of models, including (1) discrete models without message passing, (2) discrete model with message passing, (3) quantitative models without message passing, and (4) quantitative models with message passing. The three examples belong to the fourth class.

## 5.2 Boundaries and boundary conditions

If the 2D array is DIM\_1\_SIZE=N and DIM\_2\_SIZE=N, the C language “for(i=0; i<DIM\_1\_SIZE; i++)” and “for(i=0; i<DIM\_2\_SIZE; i++)” can be used to generate the array, and the array is [0] - [N-1]. If there are PDE in a model, the effective domain is at most [1] - [N-2], so the size of the domain is defined by

```
for(i=1; i<DIM_1_SIZE-1; i++).
```

```
for(i=1; i<DIM_2_SIZE-1; i++).
```

Multiple boundary conditions of PDE can be used to describe molecule diffusion at boundaries. If cells in a boundary can be assumed to be identical to cells in the opposite boundary, the periodic boundary condition (“P:=0”) can be used, otherwise the Neumann, Dirichlet, or Robin boundary condition should be used. The general form of Neumann/Robin boundary condition is “dudn + a\*u = g”. It is Neumann BC if a==0, but is Robin BC if a<>0. A special form is the zero-flux BC “dudn = 0”, where a==0 and g==0. Neumann/Robin can be handled together, if “a” is represented by “Nd” (meaning the Dirichlet term) and “g” is represented by “Nn” (meaning the Neumann term), so the form is “dudn + Nr \* u = Nn”. The “Nn” and “Nd” appear in PDEs, with “NnVal” being a parameter.

In a quantitative model, the user can assign initial concentrations to molecules in the input file, but it is more convenient to assign them in the Cellang++ program. The assignment is made together with BC in the first line of an equation in the form:

```
cell->object.field:=opde( (BC)(BC)(BC)(BC), := IC,
```

An example of boundary condition and initial condition is:

```
cell->Wg.v:=opde((Nn:=NnVal*cell->Wg.v Nd:=0.0) (Nn:=NnVal*cell->Wg.v Nd:=0.0)
(P:=0) (P:=0), :=1.0, //NnVal is a constant
```

An example of initial condition is:

```
cell->Fz.v:=opde(, :=sFz, //sFz is a constant
```

Intercellular message passing should also consider the boundary issue, to make them handled in a consistent way with the handling of PDE boundary conditions. There are two ways to define “buffer cells” (in the periphery) and “usable cells” (in the center of a 2D cell array). One is made in the C program that generates the input file, such as:

```
for (x=0; x<200; x++)
for (y=0; y<200; y++) {
if (y>=195 && y<=5) || ((x>=195 && x<=5)
type = 0;
else
type = 1;
printf("[%d,%d]=%d\n", x, y, type);
```

}

And the other is made in the Cellang++ program, such as:

```

if (pos(1) >= 5.0 & pos(1) <= 195.0) & (pos(2) >= 5.0 & pos(2) <= 195.0) then
    cell.type := 1
else
    cell.type := 0
end

```

### 5.3 How to define signaling events

In a quantitative model, how A nonlinearly activates/represses B can be described by a Hill function, and thus three kinds of events can be defined as follows.

(1) *Events of molecular interaction or gene transcription controlled by the threshold concentrations of proteins*

These events, including transcriptional activation/repression of genes and activating/repressive protein-protein interactions, are defined upon the half-maximal activation/repression coefficient in the Hill functions. When the concentration of protein A reaches the half-maximal activation/repression coefficient ( $A > ta_{A\_B}$  or  $A > tr_{A\_B}$ ), A sends the message *activation* or *repression* to B, which is captured as  $A\_Act\_B$  or  $A\_Rep\_B$  in B.

(2) *Events of molecular interaction not controlled by threshold concentrations of proteins*

Binding of receptors by ligands is computed according to the mass-action rule and shows lesser nonlinearity. The lack of evidence on at what concentrations a ligand binding to a receptor may make binding events unreasonably defined. In some situations, a bound receptor (for example, bound Tkv, Ptc, EGFR, and Notch) causes the production of a downstream signal molecule (for example, MadA, CiA, MAPK, and NICD) in a nonlinear way, and one can use the production of the downstream molecule to indicate the ligand/receptor binding. For example, if  $k_{SpiEgfr} \cdot SpiA \cdot EgfrF > pa_{Egfr\_MAPK}$ ,

then SpiA sends a binding signal to EgfrF (to indicate the occurrence of the binding event). In such situations an extra parameter can be used to properly set binding conditions. For example, since the Hh level is very low, one can use  $k_{HhPtc} \cdot Hh \cdot PtcF > 0.085 \cdot pr_{Ptc\_CiAR}$  to indicate the binding of Hh to PtcF.

PtcF.

(3) *Derived events that are dependent on multiple other events*

Derived events (mostly in the situations of combinatorial control of gene expression) occur upon the occurrence of multiple simple events and are defined using the logic relationship between these simple events. For example, the expression of the gene *atonal* is regulated by four proteins at two enhancers of the gene with the relationship:

$$\left( \overline{NICD\_Rep\_ato} \cap \overline{Ro\_Rep\_ato} \cap CiA\_Act\_ato \right) \cup Ato\_Act\_ato$$

Thus, the event  $ato\_Exp\_Ato$  is defined upon the events  $NICD\_Rep\_ato$ ,  $Ro\_Rep\_ato$ ,  $CiA\_Act\_ato$  and  $Ato\_Act\_ato$ :



```

if msgq $ cell.(Ato, Act, _) |
  (msgq $ cell.(CiA, Act, _) & msgq !$ cell.(Ro, Rep, _) & msgq !$ cell.(NICD, Rep, _)) then
    sendmsg(cell.(Ato, Exp, _))
  end
end

```

Here the operator “\$” means the message queue “contains” the message.

## 5.4 Capture of signaling events

Message passing events are buffered in the message queue “msgq” and are captured in the next time step. Signaling between same molecules located in different cells is treated as different signaling event. Thus, the sending of message “Act” from molecule A to molecule B in the same cell is captured and assembled as “A\_Act\_B\_0\_0”, with “\_0\_0” indicating the current cell. But the sending of the message from A to molecule A in the left-top neighboring cell is captured and assembled in the target cell as “A\_Act\_A\_p1\_n”, with “\_p1\_n1” indicating the relative position of the sender cell at [+1, -1]. At the end of each time step, all message queues are cleared.

## 5.5 Messages can be qualitative and quantitative

An operation such as “msgq \$ cell.(CiA, Act, \_)” checks a qualitative message. Message passing can also be used to transfer quantitative information. As the example of Notch/Delta binding shows, using message passing and two *forall* statements one can easily describe the signal *bind* sent out by the object Notch1 in the current cell to the object Delta1 in eight direct neighboring cells:

```

forall x:-1..1
forall y:-1..1
  sendmsg([x,y].(Delta1, Bind, amount_of_bound_Delta1))
end
end

```

The scope “-1..1” (the relative addresses of the target cells) for the index variables x and y means the first layer neighbors; “Delta1” is the target molecule; “bind” is the message; and “amount\_of\_bound\_Delta1” specifies the amount of bounded Delta1. When Delta1 is bound to Notch1, the bound portion should be removed from the free “Delta1” in the Delta1-providing cell. To do this, the code responding to the “Bind” message passing can simply be:

```

if msgq $ _.(Notch1, bind, _bound_amount) then
  value:=value - _bound_amount
end

```

Here the “\_bound\_amount” is an unbound variable that is to get the value of “amount\_of\_bound\_Delta1” in the message, and “value” is the local Delta1 value in the cell. If the amount of Delta1 is not concerned, “amount\_of\_bound\_Delta1” can be replaced by the anonymous variable “\_”.

Apparently, compared with hardwired molecular interactions by differential equations, message passing can be used to explore rich semantics of molecular signaling in a quantitative way or qualitatively way and to examine event-driven emergent behaviors and properties of molecular and cell signaling ([Figure 9](#)).

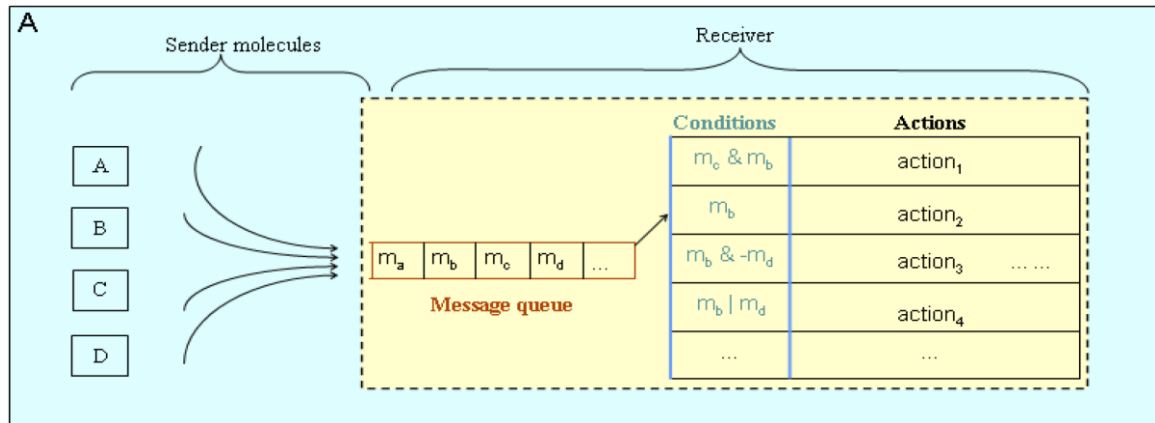


Figure 9. Event-driven molecular signaling.

## 5.6 Heterogeneity and inhomogeneity

Heterogeneity (different cell types) and inhomogeneity (different cell states) can be described in two ways, either in the input file or in the Cellang++ program. To build a heterogeneous model, the first cell field can be defined as cell type, and to build an inhomogeneous model a cell field can be defined to indicate the cell state. In the Cellang++ program, it is easy to modify the value of cell type and cell state upon defined conditions during running.

## 5.7 Runtime perturbation

Runtime perturbation, usually given to specific molecules in specific cells or directly to specific cells, can be made in two ways, either in the input file or in the Cellang++ program.

Example:

```
100
[1,1]=10,20,30
[1,2]=20,30,40
```

The more convenient way is to implement them in the Cellang++ program using an assignment statement.

## 5.8 Describe discrete (not continuous) molecule diffusion

There are two ways to describe molecule diffusion, either using partial differential equations or using message passing. Here the second is explained, assuming that the diffusion follows a first order decay process in cells. In the following case where a molecule M can freely diffuse into two layers of neighboring cells, the gradient of M in cells is controlled by A and B:

```
//In object M in the sender cell.
forall i:-2..2
forall j:-2..2
  if i!=0 | j!=0 then
    layer:=max(i,j)
    value:=A*exp(-B*layer)
    sendmsg([i,j].(M, Diff, amount))
```

```

        value:=value - amount
        //"amount" is the amount of M in this cell
    end
end
end
//In object M in the receiver cells.
    if msgq $ _.(M, Diff, _amount)
        value:=value + _amount
    end

```

Note that the anonymous variable “\_” in “if msgq \$ \_.(M, Diff, \_amount)” indicates that M from any neighboring cells can be accepted, and the unbound variable “\_amount” stores the amount of the molecule from 24 neighboring cells.

## 5.9 Dell division

Cell division can be simulated conveniently using the statement “copyto”, which enables cell proliferation in a 2D space to be modeled naturally (see Zhu and Mao. PCP and Hippo signaling encode two intercellular feedbacks controlling growth and growth arrest of *Drosophila* wing. Manuscript under review). It also enables features of lattice models and vertex models to be combined. Multiple conditions can be used to control the execution of “copyto”. When cell division is simulated, one should define only part of automata cells as biological cells at time step 0 so as to allow the remaining automata cells to be occupied by daughter cells.

## 5.10 Numerical solution of differential equations

The method of lines (MOL) method is used to solve partial differential equations. All differential equations are solved using the second-order Runge-Kutta method with adaptive time steps controlled by the two error thresholds “relerr” and “abserr”. For any molecule  $U$  satisfying  $dU/dt=f(t, U)$ , in the error control term “ $|err| < |U| \cdot relerr + abserr$ ”,  $err=(k_2 - k_1)/2$ ,  $k_1 = \Delta t \cdot f(t, U)$ ,  $k_2 = \Delta t \cdot f(t + \Delta t, U + k_1)$ , and  $relerr = abserr = 0.0001$ . In simulations, as long as “ $err \geq |U| \cdot relerr + abserr$ ”, the time step is halved. If overflow happens, it may be needed to manually reduce relerr and abserr, either in semantic.c or in the generated C codes.

## 5.11 Modeling cells with multiple compartments

It is most natural to model one biological cell as one automata cell. If the biological cells have a particular shape or multiple compartments, then a “multi-compartment cell” model is needed, in which one biological cell is represented by multiple automata cells. A specific case is the hexagonal epithelial cells, which can be represented by 6 automata cells (Zhu H, Owen M. Damped propagation of cell polarization explains distinct PCP phenotypes of epithelial patterning. Scientific Reports 2013, 3:2528) (Figure 10). In such situation, an important issue is to let every automata cell knows which of its neighboring cell belongs to the same biological cell and which does not. To do this, an identity number specific for each biological cell is generated and shared by all related automata cells. A method to implement this is to let an automata cell as the key compartment in which the random number generator (a C function) is used to generate an identity number. Then, using message passing, the automata cell sends the identity number to its partner cells. In intra- and inter-cellular message

passing, the identity number is used to judge the identity of neighboring cells. Note that “copyto” cannot be used in such models.

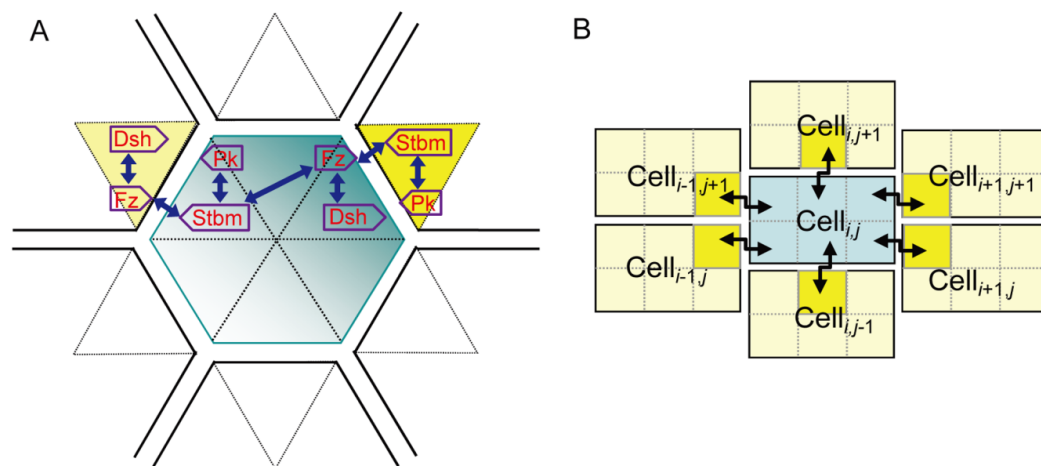


Figure 10. A hexagonal epithelial cell is represented by six automata cells.

A specific situation is that the biological cell is polarized and has two compartments; the growth of the *Drosophila* wing disc is a case. In this situation, the molecular concentrations in the two compartments can be described by two equations, and there is no need to model a cell using two automata cells.

## 5.12 Display very small values of fields

The values of some molecules may be very small. To properly display them, it is advisable to amplify their values by, say, 10 times using an extra field.

Example:

```
molecule Wg {
  v of 0.0..10.0
  v10 of 0.0..10.0
end
...
cell->Wg.v10:=cell->Wg.v*10.0
}
```

## 6. Availability and bug reports

The source code of the Cellang++ system (together with the codes of multiple models soon) is available at <http://lncRNA.smu.edu.cn/OtherCodes>. Please send bug reports and comments to [zhuhao@smu.edu.cn](mailto:zhuhao@smu.edu.cn) or [hao.zhu@ymail.com](mailto:hao.zhu@ymail.com).