SpatKin
User manual

SpatKin version: 1.0.0.
May 17, 2017

1 Introduction

1.1 General description

This manual describes SpatKin, software capable of performing stochastic simulations of reaction–diffusion kinetics of biochemical systems on the membrane. The simulator offers rule-based modeling capabilities and tracks individual molecules accounting for excluded-volume effects. An application note describing SpatKin has been published in *Bioinformatics*, 2017.

1.2 Computational approach

The employed algorithm ensures exact state-to-state dynamics of the underlying time-continuous Markov process: competing events, such as biochemical reactions and diffusive moves, are selected from a catalog of all possible events and fired with propensities proportional to their respective rate constants. The catalog of possible events is always complete as, after simulating any event, it is updated by considering every possible new event that may happen in the updated system. Complete updates are feasible due to the fact that the space is discretized using a triangular lattice. In one step, a molecule can move to an adjacent empty lattice site, a unimolecular reaction can fire, or a bimolecular reaction can occur between molecules that are placed in adjacent lattice sites. Only events defined by rules are allowed to occur. The network of possible interactions is evaluated on-the-fly for existing molecular species according to the rules specified by the user (meaning that a complete molecular interaction network does not need to be generated prior to the simulation). The method is rejection-free unless there are special regions of diminished diffusivity defined on the lattice.

In the limit of infinite diffusion, the algorithmic approach is equivalent to the Gillespie algorithm used for simulations of well-mixed systems; for fast diffusion and larger reactors, where the grain size becomes irrelevant, simulation results correspond to that obtained with finite-element method-based solvers for partial differential equations (PDEs). SpatKin can be viewed by a computer scientist as a stochastic simulator of a cellular automaton or, by a physicist, as a Boltzmann lattice-gas simulator.

1.3 Package overview

SpatKin was developed to enable defining and studying computational models of cell signaling on the membranes. The name stands for *spat*ial *kin*etics. The software enables a user to write down rules which define possible reactions, reducing the combinatorial explosion of possible states inherent to definitions of many signaling systems (for an introduction to rule-based modeling see, e.g., Chylek *et al.*, 2014, *Wiley Interdiscip. Rev. Syst. Biol. Med* **6**, 13–36). SpatKin does not share code but is conceptually profoundly inspired by BioNetGen [Harris *et al.*, 2016, *Bioinformatics* **32**, 3366–68], and patterns heavily on syntactic conventions of the BioNetGen Language (BNGL).

SpatKin is a suite of programs:

- **Spatkin** – is a graphical user interface (GUI, Qt-based) for two command line tools:
- **spatkin-kernel** – which is the core simulation program, and
- **spatkin-mosaic** – which produces graphical snapshots out of binary trajectories.

The GUI provides a syntax-aware code editor and a convenient wrapper for command-line utilities, as well as tools for tasks such as trajectory viewing and preliminary result evaluation.

Source code, written in C++, is available under the terms of the GNU Lesser General Public License (LGPL) version 3.0 and, along with binary executables for Windows and Mac, can be retrieved from the project homepage: http://pmbm.ippt.pan.pl/software/spatkin (this permalink currently redirects to http://pmbm.ippt.pan.pl/web/Spatkin).

1.4 Published applications

Preliminary versions of SpatKin have been used to study simple models of stochastic reaction–diffusion kinetics of kinases and phosphatases reacting on the membrane described in:

- Nałęcz-Jawecki P, Szymańska P, Kochańczyk M, Miękisz J, Lipniacki T: Effective reaction rates for diffusion-limited reaction cycles *Journal of Chemical Physics* **143**(21), 215102 (2015).
- Szymańska P, Kochańczyk M, Miękisz J, Lipniacki T: Effective reaction rates in diffusion-limited phosphorylation–dephosphorylation cycles. *Physical Review E* **91**, 022702 (2015).
- Kochańczyk M, Jaruszewicz J, Lipniacki T: Stochastic transitions in a bistable reaction system on the membrane. *Journal of the Royal Society Interface* **10**(84), 20130151 (2013).
- Zuk PJ, Kochańczyk M, Jaruszewicz J, Bednorz W, Lipniacki T: Dynamics of a stochastic spatially extended system predicted by comparing deterministic and stochastic attractors of the corresponding birth–death process. *Physical Biology* **9**(5), 055002 (2012).

1.5 Validation

The simulator has been validated in the limit of infinite and in the limit of zero diffusion. In both these limits, the analytically calculated amounts of molecules in each molecular state in the steady state and reaction channel firings per unit time agree perfectly [Szymańska *et al.*, 2015, *Phys. Rev. E* **91**, 022702]. For finite diffusion, analytical estimates for effective macroscopic reaction rates and simulation results agree well [Nałęcz-Jawecki *et al.*, 2016, *J. Chem. Phys.* **143**, 215102].

2 Input file

Input files are plain-textual files with extension `.spatkin`. The contents of an input file is interpreted imperatively, meaning that statements in sections are treated as commands and are evaluated "eagerly." Technically, the commands are realized as semantic actions of a recursive-descent LL(k) parser with the side-effects-free look-ahead. This imposes some intuitive restrictions on the order of appearance of sections in the whole input file and of their constituent elements. The code is read in a free-format: what matters is the order of tokens. An input file can contain comments which are skipped by the parser. Wherever possible, SpatKin strives to follow conventions of BNGL, but with a handful of exceptions (described later).

2.1 Input file sections

Every `.spatkin` file must contain sections that define parameters, properties of the 2-D simulation box ("world"), initial configuration of molecules, rules of allowed events, observables, and simulation settings:

- parameters section,
- world section,
- regions section,
- molecule types section,
- seed species section,
- event rules section,
- observables section,
- simulation settings section.

The expected order of the sections is defined in the program grammar (in Appendix B). Several complete example "programs" are included in the Tutorial models section.

2.2 Parameters section

The parameters section specifies numeric values to be used as:

- reaction rate constants,
- numbers/occupancies of seed species,
- world dimensions,
- molecular weights, etc.,

in the world, molecule types, seed species, and event rules sections.

```
begin parameters
   k    10.
   k1   1./3.
   k2   k
   k3   3.14159*(k1/k2 + 2.71828)
   nA   100
   nB   1e2*nA
end
```

Listing 1: Parameters section example.

Parameter values can be written in the usual form (e.g., 100, 0.0025) or using exponential notation (e.g. 1e2, 2.5E-3). Parameters can be evaluated on-the-fly based on values of previously defined (and immediately evaluated) parameters. Parameters cannot be reassigned. Arithmetic types are always promoted to the real, double-precision type when performing arithmetic operations, so for example 22/7 is 3.142857… Infix operators +, –, *, / take normal precedence, brackets can be used to enforce the order of evaluation. Computed parameter values are printed to the standard output by the command-line kernel executable. Appendix B contains a formal grammar of the parameters section.

2.3 World section

This section primarily sets the size and shape of the lattice, that is the number of lattice nodes in both spatial dimensions.

```
begin world
   topology plane
   size 100 100
   random seed 42
end
```

Listing 2: World section example.

The planar topology entails periodic boundary conditions (left–right and top–bottom). Reflecting boundary conditions can be obtained simply by defining two stripes of zero diffusivity at two edges (see the regions section).

By optionally varying the seed for the random number generator of the "world" one can generate diverse initial configurations of molecules. By keeping the same seed for a series of simulations it is possible to start every time from identical initial conditions. If the seed is not specified, the random number generator is seeded based on the current time (at the single-second resolution)[1]. Appendix B contains a formal grammar of the world section.

[1] Currently, the random number generator for the initial placement of molecules is the standard system `rand()`.The random number generator of the simulation engine is Mersenne twister that is seeded separately (see the simulation settings section).

2.4 Regions section

Regions, shapes of which can be circular, rectangular, or consisting of a set of individually listed cells can be defined to locally modify diffusivity or to restrict initial placement of molecules to specific areas.

```
begin regions
   LeftRegion circle 10 25 8
   RightArea  rectangle 27 20 30 10
   CCC        cells 20,30; 40,20; 12,12
   rX         !RightArea
   rY         ((LeftRegion + !RightArea) * RightArea)
end
```

Listing 3: Regions section example.

Centers of circles and rectangles are provided as first two parameters of a region definition, and the radius or width and height, respectively, are given subsequently. There is also a special region type which allows for defining regular grids over the whole "world." By decreasing diffusivity in such a grid-like region one can easily delineate multiple semi-permeable compartments.

As in constructive solid geometry, it is possible to perform usual set-algebraic operations on regions (treated as sets of lattice nodes), see Table 1.

Operator	*Meaning*
$!a$	complement of set a
$(a + b)$	sum of sets a and b
$(a * b)$	intersection of sets a and b
$(a - b)$	subtraction of b from a
$(a \hat{} b)$	symmetric difference of a and b

Table 1: Algebraic operators for regions. Precedence is the same as of arithmetic operators; brackets must be used for two-argument operators and can be used to enforce the order of evaluation.

Definitions of regions cannot be changed during the simulation. Computed volumes of regions, with respect to both molecules and immobile binders (see a section on molecule types for an explanation of the distinction), are printed out on the standard output by the kernel executable. Regions can be referenced by their identifiers in the regions, seed species, and event rules sections. Appendix B contains a formal grammar of the regions section.

2.5 Molecule types section

In this section chemical entities are defined by specifying their names and names of their sites.

```
begin molecule types
   N()            # This molecule has no sites.
   A(a,b,c)       # This molecule has three sites.
   B(c) weight 5
   C(b) weight 3
   D(d1)[2]       # This molecule can engage 2 binders.
   L[3]           # This binder can engage 3 molecules.
end
```

Listing 4: Molecule types section example. In case the molecules C and D create a complex that is allowed to dissociate, molecule C has 37.5% chance of staying in the old lattice node after associationwith/dissociation from D (62.5% chance).

There are two kinds of chemical entities: *molecules*, which represent membrane-tethered proteins and occupy hexagonal tiles of the lattice (equivalently: triangular lattice nodes); and immobile *binders*, which occupy the dual lattice (i.e., the hexagonal lattice). The main difference between these two types of entities is that (regular) molecules can possess sites capable of assuming states dependent on, e.g., their phosphorylation level and capable of binding to sites of other molecules, whereas binders have no internal state (no sites) and can only bind regular molecules using 1, 2, or 3 chemically equivalent binding sites. The binders are inspired by immunogenic ligands that induce receptor dimerization/-clusterization and immobilization, and are thus immobile.

A single lattice node may contain: no molecules, or a single (whole) molecule, or one complex (comprising several bound molecules). Because of the steric constraints of the triangular lattice, any molecule can have up to 6 other molecules and 6 binders as neighbors, and any binder can be adjacent to maximally 3 molecules (see Fig. 1).

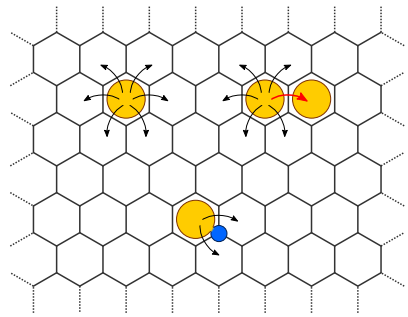

Figure 1: Lattice confinement of molecules and immobile binders. *Top left*: A molecule (orange) in a site of a triangular lattice can hop to one of unoccupied adjacent lattice sites. *Top right*: A molecule can jump to an occupied adjacent lattice site (red arrow) only when a complex formation reaction is allowed and such an event has been selected. *Bottom*: Movements of a molecule bound to an immobile binder (blue), which is a distinct kind of molecule, and as such is placed in a node of a dual lattice, are constrained so that the bond is not broken.

All sites in a single molecule must have *unique* names. This limitation disallows molecules having some symmetry, which appear usually in models of aggregation – SpatKin is not suitable for such systems, as, by convention, all the molecules that are connected by bonds (complexes) occupy just a single lattice node. After dissociation, one of reaction products is moved to an adjacent lattice site. The optional molecular weight property is used to specify the probability of staying in the old node vs. moving to an adjacent node: probability of not moving of a dissociation product is proportional to the weight fraction in the total weight of the molecular complex. Default weight of a single molecule is 1.

One doesn't have to specify possible phosphorylation states of sites in the molecule types section – they are inferred from reaction rules. SpatKin does not distinguish between phospo-sites and binding sites. Appendix B contains a formal grammar of the molecule types section.

2.6 Seed species section

This section defines initial conditions in the form of initial states of molecules: their phosphorylation states or bonds, abundance and, optionally, initial region (as defined in the world section).

```
begin seed species
   Y(d~U)            1000
   X(x)   10   # site x can be used for binding, but cannot change its phospho-state
   A(b!1) . B(a!1)    100
   A(x~pY)             nA    in region CompartmentA
   T(sh3~pY) occupancy 1.0  in region Region3
   L[@,@,@]            50   # `at' signs denote three unbound binder sites
end
```

Listing 5: Seed species section example. In the example, CompartmentA and Region3 are assumed to be defined in the regions section, and nA is assumed to be defined in the parameters section. Occupancy 1.0 causes the region Region3 to be filled completely with T molecules.

Phosphorylation states can be assigned with:

- ∼U (aliases: ∼u, ∼Y) – for sites which are not phosphorylated,
- ∼P (aliases: ∼p, ∼pY) – denoting a monophosphorylated site,
- ∼PP (aliases: ∼pp, ∼ppY) – to mark a bisphosphorylated site.

Inter-site bonds are specified by a pair of exclamation marks followed by a bond number.

Definition	*Pattern*		*Concrete*	
	phospho	*binding*	*phospho*	*binding*
A(a)	any	no	no	no
A(a∼P)	yes (single)	no	yes (single)	no
A(a∼U!1)	no	yes	no	yes*
A(a!?)	any	any	— incorrect —	
A(a∼PP!+)	yes (double)	yes	— incorrect —	

Table 2: Example molecule/site definitions and their semantics. *Correct if a molecule to which it is bound is defined nearby with a matching bond number.

Sites must be listed in the same order as in the seed species section. Complexes of molecules are placed in a single cell; it's one of SpatKin's inherent conventions. If the state of a site is not given, it is assumed to be unphosphorylated and unbound. Molecules can be initially placed in a predefined region (if there is not enough space to place molecules in a requested region, kernel will complain). Appendix B contains a formal grammar of the seed species section.

2.7 Event rules section

This section provides definitions of molecular patterns and associated events in a convention nearly identical to that of BioNetGen language (BNGL) the only exception being that patterns of molecular complexes must have their bonds wired explicitly. Reactions can involve one or two molecules (or molecular complexes) and can be uni- or bidirectional. In a simulation, bimolecular reactions can occur only between molecules (or molecular complexes) localized in adjacent lattice nodes. Parameters for bidirectional reactions are given for the forward and then for the backward reaction.

```
begin event rules

   "Movement of any A":
   >> A()  m

   # This is a comment; below, we define an unnamed rule
   # (which will be referred to as rule "2" in the output).
   A(b!1) . B(a!1)  ->  A(b) + B(a)   k

   C() + A(x∼U)  <->
   C() + A(x∼P)  kfast,kslow

   K(m,k∼U) + M(k) <-> K(m!1,k∼U) . M(k!1)  kp1,km1

   "Binding of K & M", "Unbinding of K & M":
   K(m,k∼P) + M(k) <-> K(m!1,k∼P) . M(k!1)  kp2,km2

   ++ Lig[@]                 3
   -- Lig[@]                 3
```

```
  +!  A()[@]   & Lig[]      30
  +!  A()[@!+] & Lig[]     300
  -!  A()[@!+] & Lig[@!+]   10

end
```

Listing 6: Example event rules section.

Here, the syntax of BNGL has been extended to account additionally for molecular diffusion and interactions with binders. Molecules or molecular complexes that are bound to a single binder can move only in the way that does not break the bond (adjacently to the binder), see Fig. 1. Molecules or molecular complexes that are bound to two binders are immobilized. Movements, (dis)appearance of molecules and appearance of binders, and molecule–binder (un)binding rules are defined using a prefix notation whereas reactions are defined using a middle-arrow notation.

Each rule can be optionally named. Unnamed rules are assigned names from their ordinal numbers. After simulation, the number of times a given rule was used to generate an event is printed in a summary statistics file (optional rule naming helps reading this file). Appendix B contains a formal grammar of the event rules section.

2.8 Observables section

In this section, molecules in specific states ("observables"), which are of interest, can be defined. All molecules matching defined observable patterns are counted and their count is printed to a log file (of column-based textual format).

```
begin observables
  A_every       A()                                group A
  A_unbound     A(a)             color 0.85,0.0,0.0 group A
  A_bound       A(a!+)           color gold         group A,B
  A_maybe_bound A(a!?)           color lightgray    group A,B
  L_u           Lig[@,@,@]       color gray
  L_b           Lig[@,@,@!+]     color yellow
  L_bb          Lig[@,@!+,@!+]   color orange
  L_bbb         Lig[@!+,@!+,@!+] color red
end
```

Listing 7: Example observables section.

Defining observables that are parts of complexes can be attained using the notation for the bond to any molecule, `!+`.

Observables can be grouped; then trajectory snapshots can be displayed separately for each group. To further facilitate visual analysis of snapshots, optionally, a color can be assigned to a molecule or binder: any color from the RGB color space can be defined by giving its red, green, blue components (from the range [0,1] after keyword `rgb`) or by using one of predefined X11 or SVG color name (for a complete list of names see Appendix D). Appendix B contains a formal grammar of the observables section.

2.9 Simulation settings section

This section controls duration of the simulation and logging frequency.

Description (optional) will be literally copied to a log file. Halting condition can be specified by the total simulation time or the total number steps. Logging times are uniformly spaced over the simulation time in case of specifying the number of observer intervals or the interval time, and not in

```
begin simulation
   description "My model 101"
   time end           100  # a hundred Monte Carlo seconds
   observer intervals  50  # writing down observables in fifty time points
   snapshots  off          # disable dumping spatial trajectory
   random seed       4242
end
```

Listing 8: Example simulation settings section.

case of the number of steps. Writing out trajectory can be disabled (also automated post-simulation generation of images can be disabled in GUI, in menu Settings). The random number generator seed influences the order of simulated events but does not affect the way in which molecules are inserted into the lattice. Appendix B contains a formal grammar of the simulation settings section.

3 Hints on the workflow

3.1 Model development

SpatKin stochastic simulation algorithm is an exact method, meaning that in the infinite diffusion limit becomes equivalent to the Gillespie algorithm. In a single time step, SpatKin simulates one random event. In principle, it's impossible to parallelize the execution of the algorithm and at the same time preserve its exactness. If diffusivity of molecules is much faster than reaction rates or when the simulated system is dilute, a lot of compute power is devoted to simulating diffusion (hopping on the lattice). To alleviate high computational requirements for model development and tuning, it is recommended to develop first a non-spatial version of the model (e.g., in BioNetGen) and then port it to SpatKin, taking into account existing discrepancies between the two tools—see Table 3.

Feature	BioNetGen	SpatKin
1) States of molecular sites	Any declared by user	Limited set: `~U`, `~P`, `~PP`
2) Identical molecular sites	Allowed	Allowed in immobile binders
3) States in molecule types section	Required	Not required
4) General rule syntax	Infix	Infix (reactions), prefix (other events)
5) Implicit intermolecular bonds	Supported	Not supported
6) >1 reaction in one rule	Allowed	Not allowed
6) >1 protomer in complex observable	Allowed	Not allowed
7) Rule naming syntax	`MY_RULE_1:`	`"My rule 1":`
8) Grouping of observables	Not supported	Supported

Table 3: Juxtaposition of discrepant BioNetGen and SpatKin capabilities and syntactic conventions.

3.2 Selected features of GUI

The **main window** features a multiple document interface (MDI) and thus can have multiple model files open and multiple stochastic simulations running simultaneously. If a simulation is forced to stop, partial results (including trajectory) are amenable to analysis. The last state of the simulated system at the moment of interruption is saved to a file Final.spatkin. In menu Settings one can disable automated post-simulation generation of images from the trajectory. In the **code editor window**, which is an MDI child of the main GUI window, after typing `color` an in-place drop-down list of predefined color names should appear (for the full list of color names see Appendix D).

The **plot window** that shows how amounts of declared observables evolve in time can be zoomed in by dragging mouse pointer over the drawing area; the point when the left mouse button is pressed and the point when it is released are used to define a rectangle to zoom in. By clicking with the right mouse button, the original plot ranges are restored. Lines corresponding to individual observables can be hidden or shown by toggling legend captions in the top-right window corner. Similarly, in the **histogram window**, legend captions can be toggled to hide/unhide box-plots of choice. Clicking on a sector of the pie-chart displayed in the **statistics windows** highlights a corresponding rule in the table next to the pie-chart. The **trajectory window** can show observables in separate groups (iff such groups were defined in the observables section), which significantly aids visual analysis. Trajectory can be played forth and back using arrow keys (first the slider may need to receive focus by clicking on it; press Ctrl or Mac's ⌘ to skip frames).

3.3 Continuing a simulation

When a simulation is finished or interrupted, a Final.spatkin file is generated. The generated file has the structure of an input file, where in the seed species section locations and states of all molecules at the last time step are saved. Such a file with initial conditions taken from the end of one simulation can be used as an input file to run a new simulation.

3.4 Backing up results

When a new simulation is initialized, to store trajectory and other generated files a new directory is created with a name of the input file without the file name extension (.spatkin). If such folder already exists, to prevent overwriting previous results, it will be renamed by adding a suffix that reflects its creation time (in this way, the folder containing files of the most current run has the simplest name devoid of time-stamps). When closing an input file in GUI, the presence of corresponding time-stamped directories is checked and then it's possible to remove them in bulk.

3.5 Trajectory visualization

The trajectory of a simulated system is written to a binary file (with a textual, user-editable header), that is internally compressed. The trajectory file extension is spt. A separate tool, spatkin-mosaic, is necessary to read a trajectory and generate corresponding images. The tool can generate both vector- and raster-based images. A list of supported formats and other options can be displayed by issuing the command `./spatkin-mosaic --help`.

4 Tutorial models

All "µmodels" presented in this section are shown in the form of complete, runnable input files. Raw .spatkin input files can be found in the source code distribution (in doc/examples/tutorial).

4.1 Tutorial µmodel 1: Heterogeneous initial location of molecules

This example (see Listing 9) shows how user-defined regions can be used to place molecules in a non-homogeneous manner on the lattice. One region is used to constrain the initial placement of molecules A(x∼U,y∼U) (drawn grey), at time=0; other regions are filled with molecules B (red) and C (blue) in the course of simulation according to a temporally constrained emergence rule. Molecules B modify molecules A(x∼U,y∼U) into A(x∼P,y∼U) – green. Molecules C modify molecules A(x∼U,y∼U) into A(x∼U,y∼P) – yellow. Molecules A and B degrade slowly. Simulation shows how polarization can be introduced to the system and how it vanishes due to diffusion.

```
begin parameters
  m 3
  m2 10   k 0.1    # This is a free-format text, meaning that line breaks
  r 10   q 0.01    #  do not matter.
end

begin world
  topology plane  size 200 200
end

begin regions
  # Two basic primitives for defining regions are circles and rectangles.
  CircRgn  circle    100 100 50      # centerX centerY radius
  RectRgn  rectangle 100 150 200 100 # centerX centerY width height

  # Typical constructive geometry operations are supported:
  RgnX  !CircRgn                    # !a =: complement of set a
  RgnY  (CircRgn * RectRgn)         # (a * b) =: intersection of a and b
  RgnZ  (CircRgn - RectRgn)         # (a - b) =: subtraction of b from a
end

begin molecule types
  A(x,y)   B()   C()
end

begin seed species                            # Number of molecules or frac-
  A(x∼U,y∼U)  occupancy 0.1  in region RgnX  #  tional (region) occupancy
end                                           #  should be given here.

begin event rules
  # Diffusion, molecule emergence and degradation rules have prefix syntax.
  >> A()  m    #  |
  >> B()  m2   #  >- diffusion
  >> C()  m2   # |

  # Following 2 rules for molecule insertion are both spatially and
  #  temporally constrained; effective rate of insertion is proportional
  #  to the number of unoccupied lattice nodes (here, in a region):
  ++ B()  k  in region RgnY  since 5  until 8
  ++ C()  k  in region RgnZ  since 5  until 8
```

```
  B() + A(x~U,y~U) -> B() + A(x~P,y~U)  r
  C() + A(x~U,y~U) -> C() + A(x~U,y~P)  r

  -- B()  q     -- C()  q    # Molecules B, C are degraded with rate q.
end

begin observables
    A  A(x~U,y~U)  color lightgrey          # All observables that
    Ax A(x~P,y~U)  color green              #  have assigned colors
    Ay A(x~U,y~P)  color gold               #  are recorded in the
    B  B()         color red                #  trajectory file.
    C  C()         color blue               #
end

begin simulation
    time end 300            # Total duration (in simulation time units).
    observer intervals 100  # No. time points at which logging occurs.
end
```

Listing 9: Tutorial input file 1 (doc/examples/tutorial/01-regions.spatkin).

4.2 Tutorial µmodel 2: Diffusion-limited aggregation

This example (Listing 10) demonstrates diffusion-limited aggregation in just 3 (effectively 2) rules (rules of zero rate are omitted).

```
begin parameters
  m        10
  kfast 10000
end

begin world
  topology plane  size 120 120
end

begin regions
  Seeds cells 20,30; 70,20; 82,82  # Region consists of single lattice nodes.
end

begin molecule types
  Particle(mobile)
end

begin seed species
  Particle(mobile~U)  occupancy 1.0  in region Seeds  # confined to a region
  Particle(mobile~P)  occupancy 0.2                   # distributed uniformly
end

begin event rules

  "Gogogo!":
  >> Particle(mobile~P)  m  # A named rule for diffusion.

  >> Particle(mobile~U)  0  # Anonymous rule (referred to as rule "2").

  # According to the above rules, diffusivity of a molecule depends on its state.

  Particle(mobile~P) + Particle(mobile~U)  ->
  Particle(mobile~U) + Particle(mobile~U)  kfast
end

begin observables
  frost Particle(mobile~P)  color lightgreen # An array of observables colors
  snow  Particle(mobile~U)  color orangered  #  has been predefined for user's
end                                          #  convenience (see Appendix D).

begin simulation
  time end 100         # If there are no more events, a warning will be issued.
  observer intervals 200
end
```

Listing 10: Tutorial input file 2 (doc/examples/tutorial/02-aggregation.spatkin).

4.3 Tutorial µmodel 3: State-dependent removal from the membrane

This example (Listing 11) is inspired by the fact that lipid modification, such as palmitoylation or farnesylation, can affect membrane attachment of proteins; for example, G protein α subunit is depalmitoylated upon stimulation and then translocates to cytosol.

```
begin parameters
  m1  10     b  1    d  10    u  10    x  0.1
end

begin world
  topology plane  size 100 100
end

begin regions
end

begin molecule types           # By defining relative molecular weights in
  Thioesterase(a)  weight 1    #  this manner, we assure that Thioesterase,
  AlphaS(palmito)  weight 0    #  which is assumed immobile, does not move
end                            #  upon binding/unbinding AlphaS.

begin seed species
  AlphaS(palmito~P)  1000      # site `palmito' defined explicitly as unbound
  Thioesterase(a)      10      # site `a'       defined explicitly as unbound
end

begin event rules
  >> Thioesterase(a)  0        # assumed immobile
  >> AlphaS(palmito) m1        # assumed mobile
  Thioesterase(a) + AlphaS(palmito~P)   -> Thioesterase(a!1).AlphaS(palmito~P!1) b
  Thioesterase(a!1).AlphaS(palmito~P!1) -> Thioesterase(a!1).AlphaS(palmito~U!1) d
  Thioesterase(a!1).AlphaS(palmito~U!1) -> Thioesterase(a) + AlphaS(palmito~U)   u
  -- AlphaS(palmito~U) x       # removal of depalmitoylated AlphaS from membrane
end

begin observables
  A_palmi   AlphaS(palmito~P!?) color lightpink  # '?!' means that the status of
  A_depalmi AlphaS(palmito~U!?) color red        #  binding is irrelevant here
  T         Thioesterase()       color darkblue
end

begin simulation
  description "Depalmitoylation v0.1"  # Descriptions are copied into results.
  duration 300                         # Bimolecular complexes can be occasionally
  observer intervals 100               #  seen in the trajectory as split hexagons.
end
```

Listing 11: Tutorial input file 3 (doc/examples/tutorial/03-depalmitoylation.spatkin).

4.4 Tutorial µmodel 4: Rule-based capabilities (1)

This example (Listing 12) demonstrates rule-based capabilities.

Each molecule S ("substrate") can be independently phosphorylated on 10 residues, meaning that S may assume one of $2^{10} = 1024$ phosphorylation states. Residues A, B, C, D, E can be phosphorylated by kinase K1 which is recruited and remains tethered in the circular region RgnL; residues F, G, H, I, J can be phosphorylated by kinase K2 that is recruited and remains tethered in the circular region RgnR (in this way, occurrence of several second-order reactions is constrained spatially). To become phosphorylated on all residues (dark red observable), S must visit both regions.

In the absence of phosphatase activity (parameter ku = 0), all S are ultimately phosphorylated but even a weak activity of uniformly distributed phosphatases (parameter ku = 0.01) prevents simultaneous phosphorylation of S on all residues.

```
begin parameters
  m    10.     # diffusivity
  kadd  0.1    # insertion rate
  kp   10.     # kinase activity
  ku    0.0    # phosphatase activity  <-- CHOOSE: ku=0 or ku=0.01
  occuS 0.1    # substrate occupancy
  occuP 0.03   # phosphatase ocupancy
end

begin world
  topology plane  size 200 100
end

begin regions
  RgnL  circle  50 50 30
  RgnR  circle 150 50 30
end

begin molecule types
  S(A,B,C,D,E,F,G,H,I,J) # a multi-site substrate
  K1()                   # a kinase
  K2()                   # another kinase
  P()                    # a phosphatase
end

begin seed species
  S(A~U,B~U,C~U,D~U,E~U,F~U,G~U,H~U,I~U,J~U)  occupancy occuS
  P()  occupancy occuP
end

begin event rules

  >> S() m  # By omitting diffusive rules for K1 and K2, they are
  >> P() m  #  made immobile.

  K1() + S(A~U) -> K1() + S(A~P) kp
  P()  + S(A~P) -> P()  + S(A~U) ku
```

```
  K1() + S(B~U) -> K1() + S(B~P) kp
  P()  + S(B~P) -> P()  + S(B~U) ku

  K1() + S(C~U) -> K1() + S(C~P) kp
  P()  + S(C~P) -> P()  + S(C~U) ku

  K1() + S(D~U) -> K1() + S(D~P) kp
  P()  + S(D~P) -> P()  + S(D~U) ku

  K1() + S(E~U) -> K1() + S(E~P) kp
  P()  + S(E~P) -> P()  + S(E~U) ku

  K2() + S(F~U) -> K2() + S(F~P) kp
  P()  + S(F~P) -> P()  + S(F~U) ku

  K2() + S(G~U) -> K2() + S(G~P) kp
  P()  + S(G~P) -> P()  + S(G~U) ku

  K2() + S(H~U) -> K2() + S(H~P) kp
  P()  + S(H~P) -> P()  + S(H~U) ku

  K2() + S(I~U) -> K2() + S(I~P) kp
  P()  + S(I~P) -> P()  + S(I~U) ku

  K2() + S(J~U) -> K2() + S(J~P) kp
  P()  + S(J~P) -> P()  + S(J~U) ku

  ++ K1() kadd  in region RgnL  since 1.0  until 3.0
  ++ K2() kadd  in region RgnR  since 5.0  until 7.0
end

begin observables
  S_10u     S(A~U,B~U,C~U,D~U,E~U,F~U,G~U,H~U,I~U,J~U) color blue
  S_5pNterm S(A~P,B~P,C~P,D~P,E~P,F~U,G~U,H~U,I~U,J~U) color gold
  S_5pCterm S(A~U,B~U,C~U,D~U,E~U,F~P,G~P,H~P,I~P,J~P) color pink
  S_10p     S(A~P,B~P,C~P,D~P,E~P,F~P,G~P,H~P,I~P,J~P) color darkred
  K1  K1() color black
  K2  K2() color dimgrey
  P   P()  color green
end

begin simulation
  time end 500
  observer intervals 100
end
```

Listing 12: Tutorial input file 4 (doc/examples/tutorial/04-rule_based_1.spatkin).

4.5 Tutorial µmodel 5: Rule-based capabilities (2)

This example (Listing 13) is a further demonstration of rule-based capabilities and extends the previous example. Additionally here, independently of their phosphostate, molecules S can form homodimers, so there are $(2^{10})^2/2 \simeq$ over half a million potential homodimer species (more than molecules in the simulation). Dimers in which one protomer is phosphorylated on A, B, C, D, E and the other is phosphorylated on F, G, H, I, J are stabilized (not allowed to dissociate).

```
begin parameters
  m     10.    # diffusivity
  kadd   0.1   # insertion rate
  kp    10.    # kinase activity
  ku     0.0   # phosphatase activity  <-- CHOOSE: ku=0 or ku=0.01
  occuS  0.1   # S occupancy
  occuP  0.03  # phosphatase ocupancy
  b      1     # S homodimerization
  d      1     # S-S un-dimerization
  stb  100     # S-S homodimer stabilization
end

begin world
    topology plane  size 200 100
end

begin regions
  RgnL  circle  50 50 30
  RgnR  circle 150 50 30
end

begin molecule types
  S(A,B,C,D,E,F,G,H,I,J,dim,stable)
  K1()   K2()   P()
end

begin seed species
  S(A~U,B~U,C~U,D~U,E~U,F~U,G~U,H~U,I~U,J~U,dim,stable~U) occupancy occuS
  P()  occupancy occuP
end

begin event rules
  >> S() m
  >> S(dim!1).S(dim!1) m  # diffusion of S-S dimer
  >> P() m

  K1() + S(A~U) -> K1() + S(A~P) kp
  P()  + S(A~P) -> P()  + S(A~U) ku

  K1() + S(B~U) -> K1() + S(B~P) kp
  P()  + S(B~P) -> P()  + S(B~U) ku

  K1() + S(C~U) -> K1() + S(C~P) kp
```

```
  P()  + S(C~P) -> P()  + S(C~U) ku

  K1() + S(D~U) -> K1() + S(D~P) kp
  P()  + S(D~P) -> P()  + S(D~U) ku

  K1() + S(E~U) -> K1() + S(E~P) kp
  P()  + S(E~P) -> P()  + S(E~U) ku

  K2() + S(F~U) -> K2() + S(F~P) kp
  P()  + S(F~P) -> P()  + S(F~U) ku

  K2() + S(G~U) -> K2() + S(G~P) kp
  P()  + S(G~P) -> P()  + S(G~U) ku

  K2() + S(H~U) -> K2() + S(H~P) kp
  P()  + S(H~P) -> P()  + S(H~U) ku

  K2() + S(I~U) -> K2() + S(I~P) kp
  P()  + S(I~P) -> P()  + S(I~U) ku

  K2() + S(J~U) -> K2() + S(J~P) kp
  P()  + S(J~P) -> P()  + S(J~U) ku

  S(dim) + S(dim) -> S(dim!1).S(dim!1) b

S(A~P,B~P,C~P,D~P,E~P,dim!1,stable~U).S(F~P,G~P,H~P,I~P,J~P,dim!1,stable~U) ->
S(A~P,B~P,C~P,D~P,E~P,dim!1,stable~U).S(F~P,G~P,H~P,I~P,J~P,dim!1,stable~P) stb

  S(dim!1,stable~U).S(dim!1,stable~U) -> S(dim,stable~U) + S(dim,stable~U) d

  ++ K1() kadd  in region RgnL  since 1.0  until 3.0
  ++ K2() kadd  in region RgnR  since 5.0  until 7.0
 end

 begin observables
  S_10u      S(A~U,B~U,C~U,D~U,E~U,F~U,G~U,H~U,I~U,J~U) color blue
  S_5pNterm  S(A~P,B~P,C~P,D~P,E~P,F~U,G~U,H~U,I~U,J~U) color gold
  S_5pCterm  S(A~U,B~U,C~U,D~U,E~U,F~P,G~P,H~P,I~P,J~P) color pink
  S_10p      S(A~P,B~P,C~P,D~P,E~P,F~P,G~P,H~P,I~P,J~P) color darkred
  SS_dim_stable S(dim!+,stable~P)  color red  # observing S-S dimer
  K1 K1() color black
  K2 K2() color dimgrey
  P  P()  color green
 end

 begin simulation
  time end 500
  observer intervals 100
 end
```

Listing 13: Tutorial input file 5 (doc/examples/tutorial/05-rule_based_2.spatkin).

4.6 Tutorial µmodel 6: Gradient formation

This example (Listing 14) shows how a gradient of phosphorylated substrate can be formed between enzymes tethered to different "compartments" of the reactor.

```
begin parameters
  W  200    # Arithmetic expressions and previously defined parameters
  H  W/3    #  can be used to define parameters (evaluations are always
  A  8      #  performed in double floating-point precision).
  m  3    p  100    q  p
end

begin world
  topology planar size W H
end

begin regions
  reservoirK  rectangle W*(    1)/(2*A) H/2 W/A H  # in-place arithmetics
  reservoirP  rectangle W*(2*A-1)/(2*A) H/2 W/A H  #
end

begin molecule types   K()  S(s)  P()  (* kinase, substrate, phosphatase *)  end

begin seed species
  S(s~P)  occupancy 1/8    # not inserting: S(s~U), S(s~PP)
  K()     occupancy 1/32 in region reservoirK
  P()     occupancy 1/32 in region reservoirP
end

begin event rules
  >> S() m                # K(), P() are immobile
  K() + S(s~U) -> K() + S(s~P)  2*p
  K() + S(s~P) -> K() + S(s~PP)   p
  P() + S(s~PP)-> P() + S(s~P)  2*q
  P() + S(s~P) -> P() + S(s~U)    q
end

begin observables
  K    K()     color darkmagenta
  S_u  S(s~U)  color yellow    group S  # Grouping observables may aid
  S_p  S(s~P)  color orange    group S  #  trajectory visualization;
  S_pp S(s~PP) color red       group S  #  group "all" is always created.
  P    P()     color green
end

begin simulation
  duration 10000
  observer intervals 100
end
```

Listing 14: Tutorial input file 6 (doc/examples/tutorial/06-gradient_formation.spatkin).

4.7 Tutorial µmodel 7: Steady state controlled by diffusion

In this example (Listing 15), lattice is divided into two regions; molecules located in one of them have significantly reduced diffusivity. By visual inspection of the trajectory it can be observed that diffusion controls the steady state (i.e., the fraction of phosphorylated S molecules – black). This case has been analyzed by Szymańska *et al.*, 2015 [see Figure 2(b) in *Phys. Rev. E* **91**, 022702].

```
begin parameters
  a     120       # -- geometric parameter
  rhoK    0.1     # \
  rhoP  rhoK/10   # -> parameters used to set up initial conditions
  rhoS    0.25    # /
end

begin world
  topology plane  size 2*a a  # width==2*height
end

begin regions
  Left  rectangle a/2 a/2 a a  diffusivity 0.01
end

begin molecule types
   K()  P()  S(s)   # kinase, phosphatase, and their substrate
end

begin seed species
  K()     occupancy rhoK
  P()     occupancy rhoP
  S(s~U)  occupancy rhoS
end

begin event rules
  >> K() m   >> P() m   >> S()  100
  K() + S(s~U) -> K() + S(s~P)    1
  P() + S(s~P) -> P() + S(s~U)  100
end

begin observables
  K K()      color yellow
  P P()      color red
  Su S(s~U)  color lightgrey
  Sp S(s~P)  color black
end

begin settings
  time end 20
  observer intervals 200
end
```

Listing 15: Tutorial input file 7 (doc/examples/tutorial/07-diffusion_controlled_steady_state.spatkin).

4.8 Tutorial µmodel 8: Ligand-induced receptor dimerization

This is an example (Listing 16) of excitable system where introduction of trivalent ligands that collocalize bivalent receptors facilitates their activatory autotransphosphorylation. This activity is further enhanced by recruitment and activation of autotransphosphorylating kinases and opposed by phosphatases which bind to and act on both activated receptors and kinases. This example is inspired by early events in B cell receptor signaling.

```
(*
 *  A note on reproducibility:
 *  This is a stochastic simulation of an excitable system, which means that
 *  the system occasionally can get activated spontaneously (due to a fluctuation).
 *  In such case, one can change random generator seed(s) and re-run the simulation.
 *)

begin parameters
  m     10.
  occuR  0.03   # receptor occupancy
  occuK  0.1    # kinase ocupancy
  occuP  0.05   # phosphatase ocupancy
end

begin world
  topology plane  size 64 64
  random seed 12345  # This seed influences initial locations of molecules.
end

begin regions
  patch  circle 32 32 10
end

begin molecule types
  K(P,K,R,A)  # kinase
  P(R,K)      # phosphatase
  Ag[3]       # extracellular antigen, trivalent
  R(P,K,A)[2] # bivalent receptor with 2 additional sites for binding K, P
end

begin seed species
  R(P,K,A~U)[@,@] occupancy occuR  # Molecules are inserted unbound.
  P(R,K)          occupancy occuP
  K(P,K,R,A~U)    occupancy occuK
end

begin event rules

  >> R() m    >> P() m    >> K() m

  # Immobile binders appear unbound in a lattice dual to the regular lattice.
  "Antigen appearance":
  ++ Ag[@,@,@]  0.01  in region patch  since 200  until 220
```

```
  "Receptor-ligand binding", "Receptor-ligand unbinding":
  +-!  R() &  Ag[]  10, 0.01

  # Receptors activation in trans:
  R()    + R(A~U) -> R()     + R(A~P)  0.01
  R()    + R(A~P) -> R()     + R(A~PP) 0.01
  R(A~P) + R(A~U) -> R(A~P)  + R(A~P)  0.02
  R(A~P) + R(A~P) -> R(A~P)  + R(A~PP) 0.02
  R(A~PP)+ R(A~U) -> R(A~PP) + R(A~P)  0.05
  R(A~PP)+ R(A~P) -> R(A~PP) + R(A~PP) 0.05

  R(P,K,A~PP) + K(P,K,R,A~U) -> R(P,K!1,A~PP).K(P,K,R!1,A~U)     1
  R(K!1,A~PP).K(P,K,R!1,A~U) -> R(K!1,A~PP).K(P,K,R!1,A~P)      10
  R(K!1).K(R!1) -> R(K) + K(R)                                   1

  K(P,K,R,A~P) + K(P,K,R,A~U)  -> K(P,K!1,R,A~P).K(P,K!1,R,A~U)  1
  K(P,K!1,R,A~P).K(P,K!1,R,A~U)-> K(P,K!1,R,A~P).K(P,K!1,R,A~P) 10
  K(K!1).K(K!1) -> K(K) + K(K)                                   1

  P(R,K) + R(P,K,A~PP) -> P(R!1,K).R(P!1,K,A~PP)                 3
  P(R,K) + R(P,K,A~P)  -> P(R!1,K).R(P!1,K,A~P)                  3
  P(R!1,K).R(P!1,K,A~PP) -> P(R!1,K).R(P!1,K,A~P)               10
  P(R!1,K).R(P!1,K,A~P)  -> P(R!1,K).R(P!1,K,A~U)               10
  P(R!1).R(P!1) -> P(R) + R(P)                                   1

  P(R,K) + K(P,K,R,A~P)   -> P(R,K!1).K(P!1,K,R,A~P)             1
  P(R,K!1).K(P!1,K,R,A~P) -> P(R,K!1).K(P!1,K,R,A~U)            10
  P(K!1).K(P!1) -> P(K) + K(P)                                   1

end

begin observables
  Ag_tot          Ag[@!?,@!?,@!?]  color black
  Receptor_U      R(A~U)  color lightgrey
  Receptor_P      R(A~P)  color grey
  Receptor_PP     R(A~PP) color brown
  Kinase_inactive K(A~U)  color gold
  Kinase_active   K(A~P)  color red
  Phosphatase     P()      color green
end

begin simulation
  time end 500
  observer intervals 1000
  snapshots on       # If you do not want snapshots, write: snapshots off
  random seed 12345  # This seed influences the order of events.
end
```

Listing 16: Tutorial input file 8 (doc/examples/tutorial/08-receptors_and_ligands.spatkin).

4.9 Tutorial µmodel 9: Crowding-facilitated switch in a bistable system

In this example (Listing 17), kinetics of a bistable system is simulated. In a defined instant, chemically inert molecules ("crowders") are introduced which leads to reduction of reactants' diffusivity. In this system, the presence of crowders favors processive rather than distributive phosphorylation, and in this way favors the steady state of a high amount of doubly phosphorylated kinases. Initially the system is in the steady state of a low amount of phosphorylated kinases; upon recruitment of crowders to the membrane, a transition to the other steady state is observed.

The effect of the presence of crowding molecules and their diffusivity on the effective diffusivity of (other) molecules on the lattice has been analyzed by Szymańska *et al.*, 2015 [see Figure 9 in *Phys. Rev. E* **91**, 022702].

```
(*
 *  A note on reproducibility:
 *  This is a stochastic simulation of a bistable system, which means that
 *  the system may occasionally switch to another state (though it is very
 *  implausible). In such case, one can change random number generator
 *  seed(s) and re-run the simulation.
 *)

begin parameters
  rhoK  0.4
  rhoP  0.1
  d     1    / (6 * rhoP)
  c1    0.02 / (6 * rhoK)
  c2    0.15 / (6 * rhoK)
  c3    4    / (6 * rhoK)
  m     300
end

begin world
  topology planar
  size 50 50
  random seed 123
end

begin regions
end

begin molecule types
  K(a)  # kinase
  P()   # phosphatase
  C()   # crowder
end

begin seed species
  P()     occupancy rhoP
  K(a~U)  occupancy rhoK
end
```

```
begin event rules

  ## Rules for reactants:
  #
  >> K() m   >> P() m

  K(a~U) + K(a~U) -> K(a~U) + K(a~P)   2*c1
  K(a~U) + K(a~P) -> K(a~U) + K(a~PP)    c1

  K(a~P) + K(a~U) -> K(a~P) + K(a~P)   2*c2
  K(a~P) + K(a~P) -> K(a~P) + K(a~PP)    c2

  K(a~PP) + K(a~U) -> K(a~PP) + K(a~P) 2*c3
  K(a~PP) + K(a~P) -> K(a~PP) + K(a~PP)  c3

  P() + K(a~P)  -> P() + K(a~U)  d
  P() + K(a~PP) -> P() + K(a~P)  2*d

  ## Rules for crowders:
  #
  ++ C()  0.15  since 36  until 40
  >> C()  m/10
end

begin observables
  K    K(a~U)  color yellow
  K_p  K(a~P)  color orange
  K_pp K(a~PP) color red
  P    P()     color lime
  C    C()     color dimgray  # crowder
end

begin simulation
  duration 100
  observer intervals 1000
  random seed 123
  snapshots on
end
```

Listing 17: Tutorial input file 9 (doc/examples/tutorial/09-crowding_facilitated_switch.spatkin).

4.10 Tutorial µmodel 10: Traveling wave

The considered system contains phosphatases and auto-phosphorylating kinases reacting in a long cylindrical domain. This prototypical bistable system is the subject of the analysis described by Zuk *et al.*, 2012 [*Phys. Biol.* **5**, 055002] and Kochańczyk *et al.*, 2013 [*J. R. Soc. Interface* **10**, 20130151], where the concordance of particle-based simulations in SpatKin and finite-element method-based simulations of a corresponding partial differential equation system is demonstrated.

```
(*
 *  Please note that the simulation can take about two hours. Occasionally, the
 *  stochastic traveling wave may fail to propagate or spontaneous self-activation
 *  may occur in another part of the lattice -- in such case one can change random
 *  number generator seed(s) and re-run the simulation.
 *)

begin parameters
  n_stat_K    183  #  |
  n_stat_Kp   571  #  > high-phospholevel steady state (calculated from ODEs)
  n_stat_Kpp  439  # |
  rhoK  0.4
  rhoP  0.1
  d     1    / (6 * rhoP)
  c1    0.02 / (6 * rhoK)
  c2    0.18 / (6 * rhoK)
  c3    4    / (6 * rhoK)
  m   1000
end

begin world
  topology planar size 404 30
  random seed 123456789
end

begin regions
  Barrier   rectangle  402  15    4 30  diffusivity 0  # reflective boundary
  Ignition  rectangle   50  15  100 30
  Rest      rectangle  250  15  300 30
end

begin molecule types
  K(a) (* self-activating kinase *)     P() (* phosphatase acting on the kinase *)
end

begin seed species
  P()      occupancy  rhoP   in region Ignition
  K(a~U)   n_stat_K          in region Ignition
  K(a~P)   n_stat_Kp         in region Ignition
  K(a~PP)  n_stat_Kpp        in region Ignition
  P()      occupancy  rhoP    in region Rest
  K(a~U)   occupancy  rhoK   in region Rest
end
```

```
begin event rules
  K(a~U)  + K(a~U) -> K(a~U)  + K(a~P)   2*c1
  K(a~U)  + K(a~P) -> K(a~U)  + K(a~PP)    c1
  K(a~P)  + K(a~U) -> K(a~P)  + K(a~P)   2*c2
  K(a~P)  + K(a~P) -> K(a~P)  + K(a~PP)    c2
  K(a~PP) + K(a~U) -> K(a~PP) + K(a~P)   2*c3
  K(a~PP) + K(a~P) -> K(a~PP) + K(a~PP)    c3
  P() + K(a~P)  -> P() + K(a~U)   d
  P() + K(a~PP) -> P() + K(a~P) 2*d
  >> K() m
  >> P() m
end

begin observables
  K    K(a~U)   color yellow
  K_p  K(a~P)   color orange
  K_pp K(a~PP)  color red
  P    P()       color lime
end

begin simulation
  description "Induced chemical travelling wave"
  duration 100
  observer intervals 200
  random seed 987654321
end
```

Listing 18: Tutorial input file 10 (doc/examples/tutorial/10-traveling_wave.spatkin).

A Compilation and deployment

A.1 Dependencies

The following external libraries are required to build the targets:

- **Spatkin**: Qt{Core,Gui,OpenGL,Xml,Svg}, qwt;
- **spatkin-kernel**: boost_{system,filesystem};
- **spatkin-mosaic**: boost_{system,filesystem,program_options}, cairomm, sigc++, freetype, png12, zlib.

Two of these software pieces are required to be provided in rather outdated versions:

- Boost version must be =1.55.0,
- Qwt version must be =6.1.2.

To alleviate this dependency issue, build scripts can retrieve, build, and internally deploy these libraries in expected versions. Other libraries SpatKin depends upon are expected to be available through your open source software package manager.

Satisfying dependencies on Linux. In order to install all required packages under Debian/Ubuntu Linux, it should be sufficient to[2]:

```
$ sudo apt-get install cmake
$ sudo apt-get install libbz2-dev
$ sudo apt-get install libcairomm-1.0-dev libsigc++-2.0-dev
```

The GUI component would require additionally:

```
$ sudo apt-get install qtbase5-dev qtbase5-dev-tools qttools5-dev
$ sudo apt-get install libqt5svg5-dev libqt5opengl5-dev
```

Satisfying dependencies on the Mac. Under Mac OS X/macOS with MacPorts installed, one can install required software with the command[3]:

```
$ sudo port -cuv install cmake qt5 cairomm
```

Satisfying dependencies on Windows. Using MSYS (and CMake's target build system 'MSYS Makefiles') one can compile a 32-bit version of SpatKin; using MSYS2/MinGW64 (and correspondingly CMake target build system 'MinGW Makefiles') one can compile a 64-bit SpatKin (this alternative is recommended). Under MSYS2, all prerequisites can be installed easily as follows:

```
$ pacman -S mingw64/mingw-w64-x86_64-binutils            \
            mingw64/mingw-w64-x86_64-gcc                 \
            mingw64/mingw-w64-x86_64-make                \
            mingw64/mingw-w64-x86_64-cmake               \
            mingw64/mingw-w64-x86_64-extra-cmake-modules \
            mingw64/mingw-w64-x86_64-pkg-config

$ pacman -S mingw64/mingw-w64-x86_64-qt5     \
            mingw64/mingw-w64-x86_64-cairomm \
            mingw64/mingw-w64-x86_64-gtkmm

$ pacman -S make patch git unzip
```

[2]This was tested last time under Ubuntu Linux 16.
[3]This was tested last time under macOS 10.12 "Sierra".

A.2 Building

The source code uses CMake which assists in its compilation on Linux, Mac, and Windows (with MinGW64/MSYS2). To compile the code, one can directly run the script build_release.sh located and intended to be launched in directory build, or follow the step-by-step instructions provided below.

In the first stage, non-standard third-party software should be retrieved and built. After changing to the top-most source directory, in the command line type:

```
$ cd build
$ cmake ../contrib  &&  make
```

If GUI is not intended to be built, one may disable building external components that are required by GUI with cmake's `-DSPATKIN_CONTRIB_NO_GUI_COMPONENTS=True`.

In the second stage, to perform a proper SpatKin build in the same directory, remove the generated CMakeCache.txt file, and then type:

```
$ cmake .. -DCMAKE_BUILD_TYPE=Release
```

At this point, an environmental variable pointing to a specific C++ compiler can be set prior to CMake invocation, and build generator and a custom install path prefix can be selected by passing appropriate CMake parameters with the above invocation, e.g. on MinGW64/MSYS2:

```
$ CXX="g++-4.8.1.exe" cmake .. -G "MinGW Makefiles" -DCMAKE_INSTALL_PREFIX=$HOME/local
```

If you want to configure and build only selected components, they should be listed explicitly using CMake variable `BUILD_SPATKIN_COMPONENTS` (e.g., `-DBUILD_SPATKIN_COMPONENTS=kernel` or `-DBUILD_SPATKIN_COMPONENTS="kernel;mosaic"`).

After configuring with CMake, depending on the previous choice of components, several build targets should become available for the proper build tool:

- all – that is default, will build all the main targets:
  - spatkin-kernel,
  - spatkin-mosaic,
  - spatkin-gui,
  - Spatkin (alias to spatkin-gui),
- install – installs to a default location (unless specified explicitly, as described above).

To have a regular build, run:

```
$ make
```

(or `mingw32-make.exe` under MinGW64/MSYS2). As the implementation of the parser makes heavy use of generative programming, please expect long compile times of grammar-defining source files.

If the build process is successful, you may want to issue

```
$ make install
```

(`mingw32-make.exe install` under MinGW64/MSYS2) and then make the built third-party libraries (stored in build/contrib) accessible by editing paths in the `LD_LIBRARY_PATH` (on Linux) or `DYLD_LIBRARY_PATH` (on the Mac) environmental variable, or just by copying the libraries to the directory, in which installed binary executables are located (Windows).

A list of tested compilers and additional hints on compilation can be found in file COMPILING.txt included in the source code distribution.

A.3 Deployment

On Windows, built software components and required libraries can be easily packaged into an installer with Nullsoft Scriptable Install System (NSIS; a suitable script is provided in source code distribution in packaging/windows). Detailed instructions on how to build and deploy a Mac executable bundle (an ".app") are contained in file INSTALL.txt.

B Syntax

Typographically, bold face is used for **sections**, sans-serif font for keywords and italics for *user-defined contents*. All user-defined variables are indicated in grammars by their type and some are also endowed with a short mnemonic (in normal serif font), forming a colon-joined pair. Variables are of one of the following four types:

- *id* – identifier,
- *string* – sequence of any characters (delimited by quotation marks),
- *integer* – unsigned integer (natural number),
- *real* – real number, always used in double precision.

All keywords are case-insensitive. Syntactically, a variant of Backus–Naur form is used to describe the structure of SpatKin programs—see Table 4.

Notation	*Meaning*
$[a]$	optional
$[a]+$	optional, one or more
$\{a\}$	set
$a\ \ b$	sequence
$(a \mid b)$	alternative
$a\ :=\ b$	assignment
$a = v$	default value of a

Table 4: Backus–Naur form used to describe SpatKin grammars (in order of decreasing precedence). Sequences and alternatives accept 2 or more arguments; sets can be empty.

B.1 Identifiers

Identifiers are ubiquitous in SpatKin programs. Any sequence of characters starting with a letter and consisting of numbers and letters is a valid identifier. Identifiers are treated in the case-sensitive manner. They have different yet intuitive scopes to denote parameters, molecules' names, molecular sites' names and aliases of observables.

B.2 Comments

Both single-line and multi-line comments are handled. Comments may occur anywhere besides quotations. Several customary code commenting styles are handled.

```
# This is a single line comment (BioNetGen-/shell-script-like).
% This is another single line comment (a'la Matlab/TeX).
// This is also a single line comment (as in C++).

/*
 * This is a multiline C-style comment.
 */

(* This ML- or Mathematica-like comment
   also spans more than one line. *)
```

Listing 19: Single- and multiline comment styles.

B.3 Grammars

```
program := parameters_section
           world_section
           regions_section
           molecule_types_section
           seed_species_section
           observables_section
           simulation_settings_section
```

Listing 20: General program grammar. All sections are required to appear, even if empty.

```
parameters_section :=
    begin parameters
        { id eval_expression }
    end [parameters]

eval_expression := eval_term
                   { ( '+' eval_term
                     | '−' eval_term ) }

eval_term := eval_factor
             { ( '*' eval_factor
               | '/' eval_factor ) }

eval_factor := ( real
               | id
               | '('eval_expression ')'
               | '−' eval_factor
               | '+' eval_factor )
```

Listing 21: Parameters section grammar.

```
world_section :=
    begin world
        topology plane size width:integer height:integer
        [random seed integer]
    end [world]
```

Listing 22: World section grammar.

```
regions_section :=
    begin regions
      [region_definition]
    end [regions]

region_definition := ( rectangle xcenter:integer ycenter:integer width:integer height:integer
                     | circle xcenter:integer ycenter:integer radius:real
                     | cells x:integer ',' y:integer [';' x:integer ',' y:integer]+
                     | grid cols:integer rows:integer [width:integer=1]
                     | region_expression
                     )
                     [diffusivity real]

region_expression := grammar follows that of eval_expression in Listing 21 with mandatory
                     brackets for two-argument operators; available operators are listed in Table 1
```

Listing 23: Regions section grammar.

```
molecule_types_section :=
    begin molecule types
        { (molecule_type | binder_type) }
    end [molecule types]

molecule_type := id '('[ id {',' id } ]')'  ['[' valency:integer ']'] [weight real]
binder_type := id '[' valency:integer ']'
```

Listing 24: Molecule types section grammar.

```
seed_species_section :=
    begin seed species
        { seed_species }
    end [seed species]

seed_species := (concrete_molecule | concrete_binder)
                ((integer | parameter) | occupancy (real | parameter)) [in region id]

concrete_molecule := mol:id '('[ id [concrete_site_state] {',' id [concrete_site_state] } ]')'

concrete_site_state := ('~U' | '~P' | '~PP') [ '!'bondnumber:integer ]

concrete_binder := bndr:id '[' '@' [ {',' '@' } ]']'
```

Listing 25: Seed species section grammar. Molecular seed species should have all their constituent sites listed.

```
event_rules_section :=
    begin event rules
        { ( movement_rule | reaction_rule | binder_rule
          | emergence_rule | extinction_rule ) }
    end [event rules]

movement_rule := ['"' name:string '"' ':']
                 '>>' molecules_pattern rate

reaction_rule := ( unidirectional_reaction_rule | bidirectional_reaction_rule )

unidirectional_reaction_rule := ['"' name:string '"' ':']
        molecules_pattern '->' molecules_pattern rate

bidirectional_reaction_rule := ['"' name:string '"','"' name:string '"' ':']
        molecules_pattern '<->' molecules_pattern rate,rate

binder_rule := ( binder_binding | binder_unbinding | binder_binding_unbinding )

binder_binding := ['"' name:string '"' ':']
        '+!' binder '+' molecule_pattern rate

binder_unbinding := ['"' name:string '"' ':']
        '-!' binder_pattern '.' molecule_pattern rate

binder_binding_unbinding := ['"' name:string '"','"' name:string '"' ':']
        '+-!' binder_pattern '&' molecule_pattern rate,rate

emergence_rule := ( molecule_emergence | binder_emergence )

extinction_rule := molecule_extinction

molecule_emergence := ['"' name:string '"' ':']
        '++' concrete_molecules rate [in region id] [since time] [until time]

binder_emergence := ['"' name:string '"' ':']
        '++' concrete_binder rate [in region id] [since time] [until time]

molecule_extinction := ['"' name:string '"' ':']
        '--' concrete_molecules rate [in region id] [since time] [until time]

time := ( real | parameter )
rate := ( real | parameter )
```

Listing 26: Event rules section grammar. Grammar production **molecules_pattern** denotes a list of molecules separated by '+' or '.' or '&'. A **molecule_pattern** is analogous to the **concrete_molecule**, defined in the seed species grammar, with the exception that not all sites have to be listed and additionally the site states of possible binding (!?) and any binding (!+) are allowed. Of note, intermolecular bonds are worked out based on bond numbers rather than on those separators.

```
observables_section :=
    begin observables
        { name:id molecule_pattern
            [color color_spec]
            [group groupname:id {',' groupname:id }] }
    end [observables]

color_spec := (colorname:id | rgb red:real ',' green:real ',' blue:real)
```

Listing 27: Observables section grammar.

```
simulation_settings_section :=
    begin simulation
        [description '"' string '"']
        stop_condition
        logging_frequency
        [snapshots off]
        [random seed integer]
    end [simulation]

stop_condition := ( time [end] real
                  | steps integer )

logging_frequency := ( observer intervals integer
                     | observer interval time real
                     | observer every steps integer )
```

Listing 28: Simulation settings section grammar.

C Computational efficiency

Time τ simulated in a single time step is inversely proportional to the effective rate of the slowest chemical process in the system, $\tau \sim 1/k_{\text{eff}} \simeq (k+m)/(k \times m)$, where m is the rate of hopping on the lattice [Nałęcz-Jawecki *et al.*, 2015, *J. Chem. Phys.* **143**, 215102]. Real time required to reach equilibrium, t_{eq}, is proportional to τ, m, lattice area S, and its occupancy ρ: $t_{\text{eq}} \sim \tau \times m \times S \times \rho = (k+m)/k \times S \times \rho$. For diffusion-limited processes $k \ll m$, and then $t_{\text{eq}} \sim m/k \times S \times \rho$.

This dependence is valid for monostable systems. In bistable systems, simulations are usually expected to last long enough to observe multiple transitions between steady states as this enables characterization of their relative stability. In such systems expected time to transition is much harder to etimate and depends on stability of steady states and lattice size [Kochańczyk *et al.*, 2013, *J. R. Soc. Interface* **10**, 20130151]; bistable systems are thus not appropriate for benchmarking. Raw performance of the simulator is characterized below in the benchmark of a simple system in which for simplicity only diffusive events are considered.

Benchmark of diffusive events

This benchmark (see Listing 29 and Figure 2) demonstrates that for medium and large lattices the number of events simulated in a unit of time depends very weakly on the lattice size. Better performance observed for small lattices can be likely attributed to hierarchical computer memory organization.

```
begin parameters     m  1.0   rho 0.1                      end
begin world          topology plane  size _SIDE_ _SIDE_   end
begin regions                                             end
begin molecule types A()                                  end
begin seed species   A() density rho                      end
begin event rules    >> A()  m                            end
begin observables    A A()                                end
begin settings       time end 1000  observer intervals 1  end
```

Listing 29: A *template* for input files used for benchmarking diffusive events. Strings `_SIDE_` are intended to be replaced in an automated way.

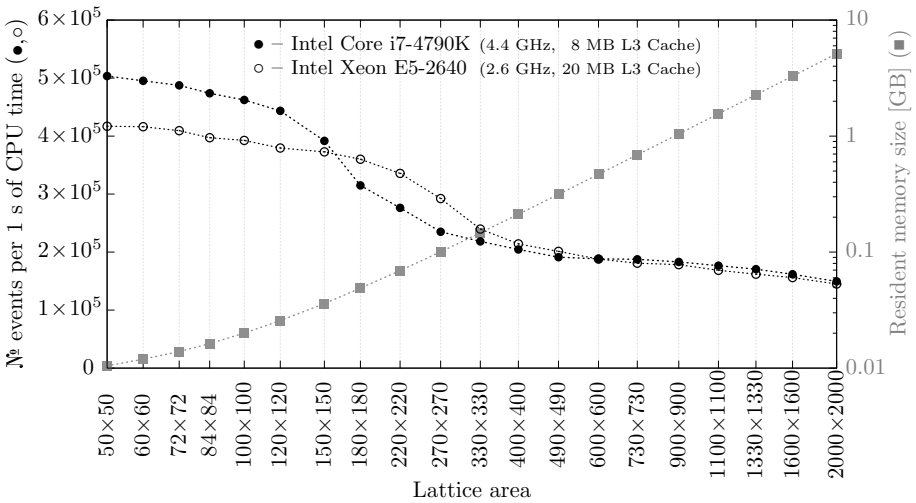

Figure 2: Benchmark of diffusive events in simulations of lattices of different sizes with occupancy of 10%. Single-core results for two different CPUs indicate that when passing from small- to medium-sized lattices, performance aggravates slightly in the CPU cache size-dependent manner. (Files that can be used to run analogous benchmarks can be found in directory benchmark in SpatKin source code distribution.)

D Predefined colors

These colors can be assigned to observables by name.

crimson	slategray	orange
deeppink	darkturquoise	goldenrod
violetred	cyan	darkkhaki
palevioletred	aqua	darkorange
mediumvioletred	darkcyan	khaki
hotpink	teal	lightgoldenrod
fuchsia	skyblue	palegoldenrod
magenta	lightseagreen	chocolate
darkmagenta	lightblue	saddlebrown
purple	mediumturquoise	peru
orchid	turquoise	orangered
darkviolet	cadetblue	sandybrown
violet	powderblue	burlywood
plum	mediumspringgreen	tan
mediumorchid	darkslategrey	sienna
indigo	darkslategray	coral
darkorchid	paleturquoise	red
thistle	springgreen	lightsalmon
blueviolet	aquamarine	tomato
mediumpurple	mediumaquamarine	darkred
blue	mediumseagreen	maroon
mediumblue	seagreen	darksalmon
darkblue	lime	firebrick
navy	green	salmon
navyblue	darkgreen	brown
lightslateblue	limegreen	indianred
mediumslateblue	forestgreen	lightcoral
slateblue	palegreen	rosybrown
darkslateblue	lightgreen	gainsboro
midnightblue	darkseagreen	lightgrey
royalblue	lawngreen	lightgray
dodgerblue	chartreuse	silver
cornflowerblue	greenyellow	darkgray
deepskyblue	yellowgreen	darkgrey
steelblue	olivedrab	gray
lightsteelblue	darkolivegreen	grey
lightskyblue	yellow	dimgrey
lightslategrey	olive	dimgray
lightslategray	gold	black
slategrey	darkgoldenrod	

Color grid 1: Darker colors.

lightpink	lightyellow	floralwhite
pink	beige	bisque
lavenderblush	ivory	antiquewhite
lavender	lemonchiffon	peachpuff
ghostwhite	cornsilk	linen
aliceblue	wheat	seashell
lightcyan	moccasin	mistyrose
azure	navajowhite	snow
mintcream	papayawhip	white
honeydew	blanchedalmond	whitesmoke
lightgoldenrodyellow	oldlace	

Color grid 2: Brighter colors.