Spatial Kappa Simulator
User Guide
v2.1.0

Anatoly Sorokin, Oksana Sorokina
The University of Edinburgh
Donal Stewart
DemonSoft.org

July 25, 2013

Acknowledgements

The development of Spatial Kappa was funded in part by SynthSys. SynthSys is a Centre for Integrative Systems Biology (CISB) funded by BBSRC and EPSRC, reference BB/D019621/1.

Contents

Rule-based modeling uses the rules to imply a model. It especially effective for the system with high combinatorial complexity, when relatively concise rule-set approximates the information about a complex biochemical network with enough precision. Proteins that participate in signaling cascades are normally composed of multiple subunits and binding domains. Those domains are typically promiscuous and could mediate more than one domain-domain interaction, which results in many possible protein complexes. Domain-domain interactions, in turn, are governed by post-translational modifications that provides additional level of variety. Therefore, when conventional modelling, such as ODE, takes the molecular structure into account - model size will grow exponentially with the number of interactions. Instead, the graphs could conveniently represent the structured objects (such as molecules with there sites and states) and graph- rewriting rules could approximate the interactions. Graph-rewriting rule specifies edge addition or removal when proteins are binding or unbinding. It also could change a state (label) of component to represent the post-translational modification. Each rule accounts for the class of reactions that chemicals with common structure could share. Only information that important for particular interaction is taken into account, so that the states of the domains that are not involved is ignored. All the reactions that are implied by one rule will be parameterized by the same rate constant, which gives the significant reduction of the model complexity.

Chapter 1

Kappa Language Extensions

Note on terminology. In the following 'voxel' means a subunit of a defined compartment. Species is mostly used to refer to a particular agent or complex in the model.

1.1 Existing Kappa language

The current Kappa language consists of the following constructs, as described in the KaSim 3.1 documentation (Feret and Krivine, 2012). A more formal description of the language grammar (including the spatial extensions) is given in Appendix A.

Biological/chemical objects of the model should be declared as agents and tokens. Agents represent the complex objects with multiple binding sites that can bind/unbind each other. They have a discrete number of instances. Tokens correspond to atomic particles of the model, they cannot bind to other tokens and agents, but can appear and disappear. They have continuous concentrations. Agent signature should contain the information about agent that will be used in the model, such as name, number of interactions sites an their states. For instance, the line:

```
Agent: A (x,y~u~p,z~0~1~2)
```

will declare that agent A has three binding sites, from which y can exist in two states (phosphorylated and unphosphorylated) and z - in three states, 1,2 and 3. More generalised:

Agents:

```
AgentName()
AgentName(stateWithValue~stateValue, stateWithNoValue, unboundSite, boundSite!1)
```

Declared agents are used to define the rules that describes the dynamics of the agents over time. Typical rule generally contains the left hand site and right hand site kappa expressions together with a corresponding kinetic rate. Rules could either be uni- or bi-directional. Rates could be a real numbers as well as algebraic expressions.

```
'my rule' kappa_expression -> kappa_expression @ rate
```

The examples of the most common rules are shown below:

Transform rules:

```
'state change' A(state~old) -> A(state~new) @ 0.1
'binding'      A(bindsite),B(bindsite) -> A(bindsite!1),B(bindsite!1) @ 0.1
'unbinding'    A(bindsite!1),B(bindsite!1) -> A(bindsite),B(bindsite) @ 0.1
'creation'     -> A() @ 0.1
'degradation'  A() -> @ 0.1
A(state~old) -> A(state~new) @ 0.1 # Unnamed transform rule
```

Initial conditions should be defined in a form of algebraic expressions that should be evaluated before the initialisation of the simulation, or variables.

Initial species values:

```
%init 1000 A(state~old),C()           # 1000 of each of A(..) and C()
%init 2000 A(bindsite!1),B(bindsite!1) # 2000 of the bound complex A(..),B(..)
%init 'n' (A(),A(y~p)) # will add 100 instances of A in default state A (x,y~u,z~0) and 100 A(x, y~p, z~0)
%var 'n' 100
```

Observables and named variables: Variables can be referenced in perturbation calculations, but do not show in outputs as observations. Observables define what you are getting as an output of the simulation.

```
%obs 'Label' A()  # All agents A()
%obs A()          # Unnamed observation, will default to 'A()' in outputs
%obs A(state~old) # All agents matching A(state~old)
%obs 'binding'    # Activity of the transform rule named 'binding'

%var 'Named variable' B()
```

Variables could be used to trigger the perturbations during the simulation.

Perturbations:

```
# When simulation time exceeds 5 seconds, change the rate of transform rule
#   named 'unbinding' to 10.0
%mod: $T > 5 do unbinding:= 10.0

# When the species value mapped to observable 'mRNA' drops below 50
#   set the rate of transform named 'transcribe' to 0.5
%mod: 'mRNA' < 50 do 'transcribe' := 0.5
```

Currently unused: There are some Kappa syntax elements not directly relevant to the spatial extensions, and will not appear in the extended grammar. One is a representation of causal flows, which uses the keyword `%causal`.

```
%causal: C@s2
%causal: A..B,C@s1 => C@s2
```

1.2 Concepts to encapsulate

The spatial Kappa language requires some encoding of the following concepts to be useful.

- A description of **compartments** and their **dimensions**. A model may have multiple compartments each containing reacting species. The dimensions are necessary to define the shape of a compartment, be it a single cell, a 1 dimensional linear array of voxels, a 2 dimensional grid of some form, or a 3 dimensional lattice structure. Relative differences in size of different compartments can be specified, e.g. the reacting volume of a nucleus relative to the surrounding cytosol. Relative differences in shape can also be specified, for example the thin layer of cytosol next to the inner surface of the plasma membrane versus the rest of the cytosol.

- Predefined compartment shapes. To concisely create more accurate models, predefined compartment shapes in 2 and 3 dimensions can be chosen, these include open and closed circles, spheres and cylinders.

- A description of **channels**, both intra-compartment and inter-compartment. The intra-compartment channel specification should be rich enough to allow description of multiple structures, e.g. in 1D linear arrays or circles, in 2D square or hexagonal meshes, cylinders or tori, in 3D cubes, filled cylinders, spheres, etc.

- Predefined channels. Commonly used inter and intra-compartment channels can be easily specified by name.

- A means of **locating species** within compartments, e.g. all DNA would reside within the nucleus, or cell receptors would be limited to the plasma membrane. Note that a multi-agent species need not be confined to a single voxel, but may span neighbouring, connected voxels.

- A means of **locating transition rules** within compartments, e.g. DNA transcription is isolated to the nucleus. The language should also allow the same transform rule to be specified with different rates depending on the location of the reacting species.

- A description of the **transport** (active or diffusive) of species within a compartment or between compartments along previously described channel structures. The rates of transport should be general to all species, or species specific. Note that transport of multi-voxel species should also be possible.

- **Variable rates of diffusion** depending on the size of species diffusing. There should be a means of describing how diffusion rates change as overall species size increases or species composition changes.

Additional concepts

- **Granularity** within compartments. It would be useful to be able to specify locations at the level of compartments or single voxels within a compartment. This would allow the model to represent, for example, a signal cascade being initiated as one point in the cytosol, and the resulting signal molecules being diffused through the cytosol.

- **Backwards compatibility** with basic Kappa. Given the quantity of existing models, the extended language should allow the existing models to work as before without modification. The user should have the choice of not using the spatial aspects of the extended language with no rework penalty.

1.3 New language constructs

A full BNF description of the extended Kappa grammar is given in Appendix A. The new constructs are identifiable as new rule types (`%channel` and `%compartment`), and location or channel identifiers in existing rule types prefixed with ':'.

1.3.1 Compartments and voxels

Compartments are defined as single voxels or regular multidimensional arrays of voxels as follows

```
'%compartment:' name=id ('[' INT ']')*
```

For example

```
%compartment: SingleCell
%compartment: line  [10]       # 10 voxels in size
%compartment: plane [10][5]    # 10x5 voxels in size
%compartment: box   [10][5][4] # 10x5x4 voxels in size
```

Compartments or individual voxels within a compartment can be referenced using the following location syntax

```
':' id ( '[' cellIndexExpr ']' )*
```

where

```
cellIndexExpr :
  cellIndexAtom operator_cell_index cellIndexAtom
  | cellIndexAtom

cellIndexAtom :
  '(' cellIndexExpr ')'
  | INT
  | id

operator_cell_index :
  '+' | '-' | '*' | '/' | '%'
```

For example

```
:myCompartment                     # the compartment as a whole
:myCompartment [0][0][0]           # the first voxel in a 3d array compartment
:myCompartment [4]                 # the fifth voxel in a 1d linear arrays

:myCompartment [x][y][z]           # variable name usage described in
:myCompartment [x*2][y -1][z +(x*3)] #   channel section below
```

In all situations where locations are used, it is only legal to refer to the compartment as a whole by omitting the cell indices, or refer to a single voxel, by fully defining the correct number of cell indices to match the dimensions of the compartment. Also, variable names are only permitted in locations within channel definitions, described below.

Predefined compartment types

Commonly used non-rectangular compartments can also be concisely defined. These include both solid and hollow (open) shapes in 2 and 3 dimensions. The available predefined compartment types are

Name	Parameters			
2D				
OpenRectangle	height	width	thickness	
SolidCircle	diameter			
OpenCircle	diameter	thickness		
3D				
OpenCuboid	height	width	depth	thickness
SolidSphere	diameter			
OpenSphere	diameter	thickness		
SolidCylinder	diameter	length		
OpenCylinder	diameter	length	thickness	

This allows the creation of a voxellated approximation of the shape specified, which can then be used for simulation. For the open compartment types, thickness specifies how thick in voxels the compartment reaction volume is.

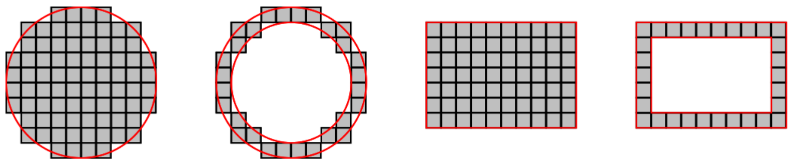


Figure 1.1: 2D compartment types (open and solid)

The syntax to specify these compartments is

```
'%compartment:' name=id type=id ('[' INT ']')*
```

For example

```
### 2D Shapes

%compartment: solidRectangle              [10][5]       # [height][width]
%compartment: openRectangle OpenRectangle [10][5]  [2]  # [height][width] [thickness]
%compartment: solidCircle   SolidCircle   [10]          # [diameter]
%compartment: openCircle    OpenCircle    [10]  [2]     # [diameter] [thickness]

### 3D Shapes

%compartment: solidCuboid                 [10][5][8]    # [height][width][depth]
%compartment: openCuboid    OpenCuboid    [10][5][8] [2] # [height][width][depth] [thickness]
%compartment: solidSphere   SolidSphere   [10]          # [diameter]
%compartment: openSphere    OpenSphere    [10]  [2]     # [diameter] [thickness]
%compartment: solidCylinder SolidCylinder [10][8]       # [diameter][length]
%compartment: openCylinder  OpenCylinder  [10][8]  [2]  # [diameter][length] [thickness]
```

1.3.2 Channels

The structure of a compartment is further defined by how voxels within the compartment are linked to each other and to connected compartments. The channels are then used in defining both static links between agents, and movement of agents through the geometry of the model. These channels are defined as follows

```
'%channel:' id channel
| '%channel:' id '(' channel ')' ('+' '(' channel ')')*
```

where

```
channel :
  source=locations '->' target=locations

locations :
  location (',' location)*
```

Where `location` is as described above. For example

```
%compartment: torus [10][200]    # 10x200 voxels in size

# Link all voxels to their horizontally adjacent neighbours
# Link all voxels to their vertically adjacent neighbours
# Wrap around the voxels on the left and right edges to create a cylinder
# Wrap around the voxels on the top and bottom edges to create a torus
%channel: meshlinks \
    (:torus[x][y] -> :torus[x +1][y]) + (:torus[x][y] -> :torus[x -1][y]) +\
    (:torus[x][y] -> :torus[x][y +1]) + (:torus[x][y] -> :torus[x][y -1]) +\
    (:torus[0][y] -> :torus[9][y])    + (:torus[9][y] -> :torus[0][y]) +\
    (:torus[x][0] -> :torus[x][199]) + (:torus[x][199] -> :torus[x][0])
```

The above code defines a thin torus composed of a 2d mesh.

Locations on the left hand side of the channel definitions above may contain either constant values or single variable names, not complex expressions. The variable names are used to define the dimensions which will be iterated through to produce links. Locations on the right hand side allow constant values or complex expressions involving the variables defined on the left hand side of the expression. It is invalid for the right hand expression to use variables not defined on the left. If setting the values of variables references valid voxels on both the left and right, then those voxels are deemed to be linked. References which refer to voxels outside the dimensions of the compartment are ignored, and no link is created. The references in a channel expression can refer to the same compartment, or two different compartments. The modulus operator `%` is useful in defining regular, repeating linkage patterns within the compartment, for example the 2D hexagonal mesh described in Appendix B.1.

Channels can make use of multiple source voxels simultaneously. For example if a model was to represent the movement of transmembrane proteins laterally along the surface of a membrane, then the channel used to describe the lateral motion would need to include simultaneous movement in two compartments (cytosol and membrane). This is represented as follows:

```
%compartment: membrane [5][5]
%compartment: cytosol [5][5][5]

%channel: diffusion \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x +1][y], :cytosol [u +1][v][0]) + \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x -1][y], :cytosol [u -1][v][0]) + \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x][y +1], :cytosol [u][v +1][0]) + \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x][y -1], :cytosol [u][v -1][0])
```

Here, variables x,y represent locations in the membrane, and u,v represent locations in the cytosol. The definition updates the locations in these two compartments in unison.

Further examples of compartment and channel specifications for common structures are given in Appendix B.1.

Predefined channel types

Commonly used 2D and 3D channel types can be concisely defined.

Name	Each voxel connected to
2D	
EdgeNeighbour	4 neighbours which share an edge in grid
Hexagonal	6 neighbours which share an edge in hexagonal grid
Neighbour	8 neighbours which share an edge or corner in grid
3D	
FaceNeighbour	6 neighbours which share a face in grid
Neighbour	26 neighbours which share a face, edge or corner in grid

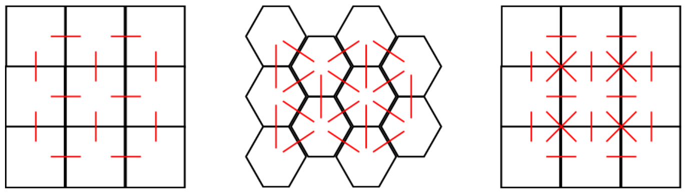


Figure 1.2: 2D channel types: EdgeNeighbour, Hexagonal and Neighbour

There are also predefined directional channel types usable in both 2D and 3D compartments

Name	Each voxel connected to
Radial	neighbours both directly towards and away from compartment centre
RadialIn	neighbours both directly towards compartment centre
RadialOut	neighbours both directly away from compartment centre
Lateral	neighbours at same distance from compartment centre

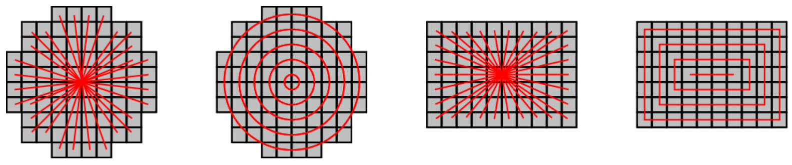


Figure 1.3: Directed channel types: Radial and Lateral

The syntax to use a predefined channel type is

```
'%channel:' id channel
| '%channel:' id '(' channel ')' ('+' '(' channel ')')*
```

where

```
channel :
  type=id source=locations '->' target=locations
```

Where `locations` is as described above. For example

```
%compartment: solidRectangle[5][5] # [height][width]

%channel: radial Radial :solidRectangle -> :solidRectangle
%channel: radialIn RadialIn :solidRectangle -> :solidRectangle
%channel: radialOut RadialOut :solidRectangle -> :solidRectangle
%channel: lateral Lateral :solidRectangle -> :solidRectangle
```

Using predefined channel types between compartments

It is possible to use predefined channel types (usually 'Neighbour') to describe movement between compartments. For this to be valid, the compartments must have compatible geometry. Compatible compartments are treated as being nested one inside the other and the channel then specifies the diffusion (or agent linkage) across the boundary between the two compartments. The compartments must fit together exactly, with no overlapping voxels or gaps. For example, a circle may only be nested inside a larger open circle whose dimensions (diameter and thickness) allow the smaller circle to fit exactly inside.

```
%compartment: cytosol   SolidCircle   [3]         # [diameter]
%compartment: membrane  OpenCircle    [7] [2]     # [diameter] [thickness]

%channel: domainLink Neighbour (:cytosol -> :membrane) + (:membrane -> :cytosol)
```

1.3.3 Locating agents

The definitions above can now be used to locate species within the model. Any rule that accepts a definition of agents (i.e. transition, var, obs, init rules), now allows these agents to be located. For each group of agents, a prefixed location constrains the agents to that location. For example

```
%compartment: membrane [5][5]
%compartment: cytosol [5][5][5]

%init: 1000 A                   # A distributed evenly among all voxels in model
%init: 1000 :cytosol B          # B distributed evenly among all voxels in cytosol
%init: 1000 :membrane[2][2] C   # C in one voxel of the membrane only
```

In addition, individual agents can have a specified location. For example

```
%init: 1000 B:cytosol             # B distributed evenly among all voxels in cytosol
%init: 1000 C:membrane[2][2](s~u) # C in one voxel of the membrane only
```

When locations are specified both for agent groups and individual agents, the individual agent location takes precedence. This allows for concise definition of agent groups where all but one of the agents in the group share a location.

Voxel wildcards may also be used when specifying agent location. For example

```
%compartment: cytosol [5][5][5]

%init: 1000 :cytosol[5][5][?] A # A distributed along a single edge of compartment
%init: 1000 :cytosol[?][?][2] B # B distributed in a plane across middle of compartment
```

1.3.4 Agent links

Agents in neighbouring voxels linked by a defined channel can be linked together. This is an extension of the basic Kappa link syntax to name the channel used to link the agents. For example

```
%compartment: membrane [5][5]
%compartment: cytosol [5][5][5]

%channel: domainLink \
    (:membrane [x][y] -> :cytosol [x][y][0]) + (:cytosol [x][y][0] -> :membrane [x][y])

%init: 1000 A:membrane(d!1:domainLink), B(d!1)
```

The above describes a model where the species A-B exists in two compartments, B in the cytosol and A embedded in the membrane. When specifying agent links using channels, only one end of the link needs to specify the channel. If a link does not specify the channel, it is assumed that both agents party to the link exist in the same voxel.

Links including channels can be created or broken in the same way as basic Kappa links in transition rules.

1.3.5 Species movement

Species can move along defined channels. Species movement is described using the `->:` operator.

```
(source=location)? '->:' channelName=id (target=location)?
| (a=agentGroup)? '->:' channelName=id (b=agentGroup)?
```

Movement transition rules can either constrain the movement by species chosen, or by source location. For example

```
%compartment: membrane [5][5]

%channel: diffusion \
    (:membrane [x][y] -> :membrane [x+1][y]) + (:membrane [x][y] -> :membrane [x - 1][y]) + \
    (:membrane [x][y] -> :membrane [x][y+1]) + (:membrane [x][y] -> :membrane [x][y - 1])

'diffusion A' A(s,t) ->:diffusion A(s,t) @ 1.0
'diffusion B' B(s,t) ->:diffusion B(s,t) @ 1.0
'diffusion AB' A(s!1,t),B(s!1,t) ->:diffusion A(s!1,t),B(s!1,t) @ 0.5

'diffusion all' ->:diffusion @ 1.0 # All species located in a single voxel will match this rule
```

To describe movement of species which span more than one voxel, use the multi agent channel definition above. For example

```
%compartment: membrane [5][5]
%compartment: cytosol [5][5][5]

%channel: diffusion \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x+1][y], :cytosol [u+1][v][0]) + \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x -1][y], :cytosol [u -1][v][0]) + \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x][y+1], :cytosol [u][v+1][0]) + \
    (:membrane [x][y], :cytosol [u][v][0] -> :membrane [x][y -1], :cytosol [u][v -1][0])

%channel: domainLink \
    (:membrane [x][y] -> :cytosol [x][y][0]) + (:cytosol [x][y][0] -> :membrane [x][y])

'diffusion A' A_m:membrane(s,t,d!1:domainLink),A_c(d!1) ->:diffusion \
              A_m:membrane(s,t,d!1:domainLink),A_c(d!1) @ 1.0
'diffusion B' B_m:membrane(s,t,d!1:domainLink),B_c(d!1) ->:diffusion \
              B_m:membrane(s,t,d!1:domainLink),B_c(d!1) @ 1.0

'diffusion AB' A_m:membrane(s!2,t,d!1:domainLink),A_c(d!1), \
               B_m:membrane(s!2,t,d!3:domainLink),B_c(d!3) ->:diffusion \
               A_m:membrane(s!2,t,d!1:domainLink),A_c(d!1), \
               B_m:membrane(s!2,t,d!3:domainLink),B_c(d!3) @ 0.5
```

Fixed location constraints

It is necessary in some models to distinguish between species which can diffuse freely within all voxels of a compartment, and species which are fixed to a single voxel. The predefined location `:fixed`, used on the right hand side of a transition rule, allows this. For example, in the model below, agent `A()` can diffuse freely, while agent `B()` must remain static.

```
%compartment: cytosol [5][5][5]

%channel: diffusion Neighbour :cytosol -> :cytosol

'diffusion A' :cytosol A(),B() ->:diffusion :cytosol A(),B:fixed() @ 1.0
```

1.3.6 Instance specific reaction rates

The rate of a transition rule can depend on the composition and size of the particular species it is being applied to. This is possible for all types of transition, not just diffusion transitions.

An agent description enclosed in '||' may be used in a rate equation.

```
'|' agentGroup '|'
```

where `agentGroup` is agent definition syntax as used on the left hand side of transition rules.

When applied to particular instances of complexes, the number of matching agent structures in the complex instances chosen are counted and substituted into the rate.

For example, in the model below, diffusion rate is inversely proportional to the number of `A()` in the chosen complex

```
%compartment: array [10]
%channel: diffusion :array[x] -> :array[x+1]

%agent: A(x,y)

%init: 100 :array[0] A(x,y!1), A(x!1,y!2), A(x!2,y!3), A(x!3,y)
%init: 100 :array[0] A(x,y!1), A(x!1,y)
%init: 100 :array[0] A(x,y)

'diffusion A' A(x) ->:diffusion A(x) @ 1 / |A()|
```

In the second example, the diffusion rate depends on the agent types within individual complex instances

```
%compartment: array [10]
%channel: diffusion :array[x] -> :array[x+1]

%agent: A(x,y)
%agent: B(x,y)

%init: 100 :array[0] A(x,y!1), A(x!1,y!2), A(x!2,y!3), A(x!3,y)
%init: 100 :array[0] A(x,y!1), A(x!1,y!2), B(x!2,y!3), B(x!3,y)
%init: 100 :array[0] B(x,y!1), B(x!1,y!2), B(x!2,y!3), B(x!3,y)
%init: 100 :array[0] B(x,y!1), B(x!1,y!2), A(x!2,y!3), A(x!3,y)

'diffusion AX' A(x) ->:diffusion A(x) @ 1 / (|A()| + 10 * |B()|)
'diffusion BX' B(x) ->:diffusion B(x) @ 1 / (|A()| + 10 * |B()|)
```

In the final example, the diffusion rate depends on the states of the agents within individual complex instances

```
%compartment: array [10]
%channel: diffusion :array[x] -> :array[x+1]

%agent: A(x,y,s~y~n)

%init: 100 :array[0] A(s~y,x,y!1), A(s~y,x!1,y!2), A(s~y,x!2,y!3), A(s~y,x!3,y)
%init: 100 :array[0] A(s~y,x,y!1), A(s~y,x!1,y!2), A(s~n,x!2,y!3), A(s~n,x!3,y)
%init: 100 :array[0] A(s~n,x,y!1), A(s~n,x!1,y!2), A(s~n,x!2,y!3), A(s~n,x!3,y)
%init: 100 :array[0] A(s~n,x,y!1), A(s~n,x!1,y!2), A(s~y,x!2,y!3), A(s~y,x!3,y)

'diffusion A' A(x) ->:diffusion A(x) @ 1 / (|A(s~y)| + 10 * |A(s~n)|)
```

The '||' clause allows any agent declaration that could appear on the left hand side of a transition rule, including agents, complexes, agent state and agent location.

1.3.7 Voxel breakdown of observables

Observables using the constructs `%obs:` and `%var:` can be used to get the total amount of species per compartment. For example

```
%obs: 'A in cytosol' :cytosol A()
%var: 'A in cytosol' :cytosol A()
```

However, for some simulations, recording amount of observable in each voxel of a compartment is useful. In these cases, using the keyword `voxel` will allow recording of observables per voxel. This works only on observable or variable declarations which specify a compartment as the location of the declaration. For example

```
%obs: voxel 'A in cytosol' :cytosol A()
%var: voxel 'A in cytosol' :cytosol A()
```

The above each declare variables which record for each voxel within the compartment `cytosol`.

Warning - as all voxels within the dimensions of the compartment are recorded for each observable using the `voxel` keyword at each timepoint, the output data file will rapidly become very large.

For example models demonstrating the use of the language extensions, refer to Appendix B.

Chapter 2

Spatial Kappa simulator User Guide

2.1 Obtaining the simulator

The simulator is available from GitHub as the source Eclipse project, or as a single executable jar file. Both are available at https://github.com/lptolik/SpatialKappa/.

2.2 Starting the simulator

2.2.1 Running the executable jar

The simulator can be started by running the executable jar file:
`java -jar SpatialKappa-v2.1.0.jar`

Double clicking the jar file usually works too.

2.2.2 Running from the Eclipse project

The main class of the simulator is
`org.demonsoft.spatialkappa.ui.SpatialKappaSimulator`

Running as a Java Application will bring up the simulator.

2.3 Using the simulator

The initial screen appears as figure 2.1.

The toolbar options are:

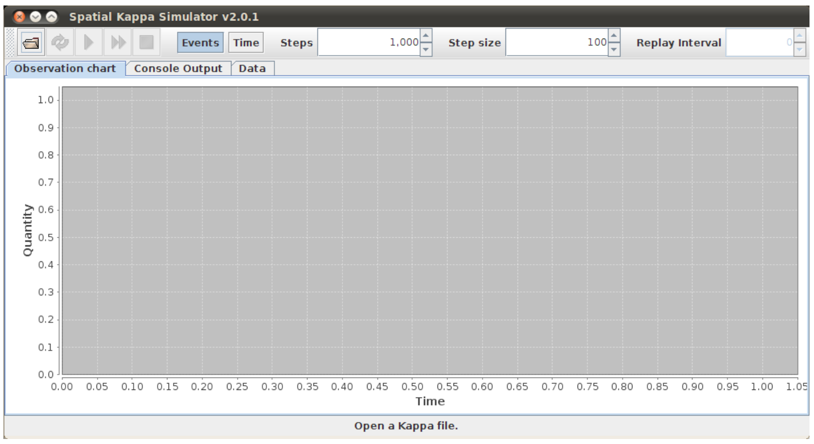
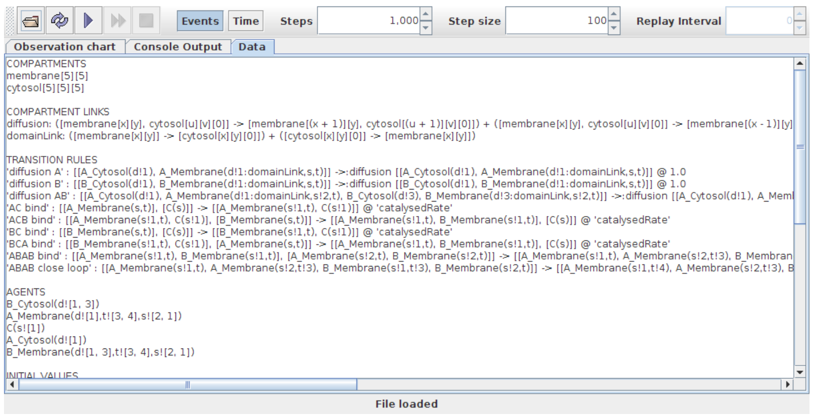


Figure 2.1: Initial view

📂	Open a Kappa or Kappa replay file
🔄	Reopen the current file
▶	Run the current file
⏩	Replay the current replay file or last run
■	Stop the currently running simulation
Events Time	Choose event or time based simulation

2.3.1 Opening a Kappa or Kappa replay file

Select the 'Open' button on the toolbar and select the file to open. The current implementation expects Kappa source files to have the suffix `.ka` and Kappa replay files (discussed later) to have the suffix `.kareplay`. If the file is parsed successfully, a summary of the Kappa model is displayed in the 'Data' pane (see figure 2.2).



```
COMPARTMENTS
membrane[5][5]
cytosol[5][5][5]

COMPARTMENT LINKS
diffusion: ([membrane[x][y], cytosol[u][v][0]] -> [membrane[(x + 1)][y], cytosol[(u + 1)][v][0]]) + ([membrane[x][y], cytosol[u][v][0]] -> [membrane[(x - 1)][y]
domainLink: ([membrane[x][y]] -> [cytosol[x][y][0]]) + ([cytosol[x][y][0]] -> [membrane[x][y]])

TRANSITION RULES
'diffusion A' : [[A_Cytosol(d!1), A_Membrane(d!1:domainLink,s,t)]] ->:diffusion [[A_Cytosol(d!1), A_Membrane(d!1:domainLink,s,t)]] @ 1.0
'diffusion B' : [[B_Cytosol(d!1), B_Membrane(d!1:domainLink,s,t)]] ->:diffusion [[B_Cytosol(d!1), B_Membrane(d!1:domainLink,s,t)]] @ 1.0
'diffusion AB' : [[A_Cytosol(d!1), A_Membrane(d!1:domainLink,s!2,t), B_Cytosol(d!3), B_Membrane(d!3:domainLink,s!2,t)]] ->:diffusion [[A_Cytosol(d!1), A_Meml
'AC bind' : [[A_Membrane(s,t)], [C(s)]] -> [[A_Membrane(s!1,t), C(s!1)]] @ 'catalysedRate'
'ACB bind' : [[A_Membrane(s!1,t), C(s!1)], [B_Membrane(s,t)]] -> [[A_Membrane(s!1,t), B_Membrane(s!1,t)], [C(s)]] @ 'catalysedRate'
'BC bind' : [[B_Membrane(s,t)], [C(s)]] -> [[B_Membrane(s!1,t), C(s!1)]] @ 'catalysedRate'
'BCA bind' : [[B_Membrane(s!1,t), C(s!1)], [A_Membrane(s,t)]] -> [[A_Membrane(s!1,t), B_Membrane(s!1,t)], [C(s)]] @ 'catalysedRate'
'ABAB bind' : [[A_Membrane(s!1,t), B_Membrane(s!1,t)], [A_Membrane(s!2,t), B_Membrane(s!2,t)]] -> [[A_Membrane(s!1,t), A_Membrane(s!2,t!3), B_Membran
'ABAB close loop' : [[A_Membrane(s!1,t), A_Membrane(s!2,t!3), B_Membrane(s!1,t!3), B_Membrane(s!2,t)]] -> [[A_Membrane(s!1,t!4), A_Membrane(s!2,t!3), B

AGENTS
B_Cytosol(d![1, 3])
A_Membrane(d![1],t![3, 4],s![2, 1])
C(s![1])
A_Cytosol(d![1])
B_Membrane(d![1, 3],t![3, 4],s![2, 1])

INITIAL VALUES
```

Figure 2.2: Data pane showing loaded Kappa model

Any errors in reading the Kappa file are shown in the ’Console Output’ pane.

The currently open Kappa file can be refreshed from disk by selecting the ’Reopen’ button. Useful when editing the Kappa model.

2.3.2 Running a simulation

With a successfully opened Kappa model, one can run a simulation by selecting the ’Run’ button. Simulation parameters can be set on the toolbar before running. There is the option to do an event or a time based simulation. For an event based simulation, the number of steps for the simulation (i.e. data points on the time series chart), and the number of finite rate events per step can be set. Equivalent options for time based simulation can also be set.

The simulation can be halted at any point by selecting the ’Stop’ button. Note that complex simulations may take some time to start up while data structures are being generated.

A replay file of the simulation is created in the same directory as the Kappa model. The file format is similar to that produced by KaSim.

2.3.3 Running a simulation replay

As the simulation runs, the state of the simulation observables are logged to disk in a replay file after every step. Once the simulation is complete, this replay file can be rerun by selecting the ’Replay’ button. The ’Replay Interval’ field allows a delay (in ms) to be added between each step.

The file format is similar to that produced by KaSim. Renaming KaSim data output to have the suffix `.kareplay` will allow KaSim output to also be visualised using this tool.

2.4 Time series chart

While the raw data produced from simulations is useful, visualisation of the data is important. There is a simple visualisation panel in the simulator. This is dynamically updated as the simulation runs to give the user an idea of how the simulation is progressing. It is however basic in comparison to some of the commercial simulation data visualisation tools available.

The time series chart is similar to the standard Gnuplot output from KaSim. It is a line graph showing observable quantity against time for all observable definitions in the model. The excellent JFreeChart (Gilbert et al., 2010) library was used for generating the charting component. The chart has formatting and save capability, and is zoomable.

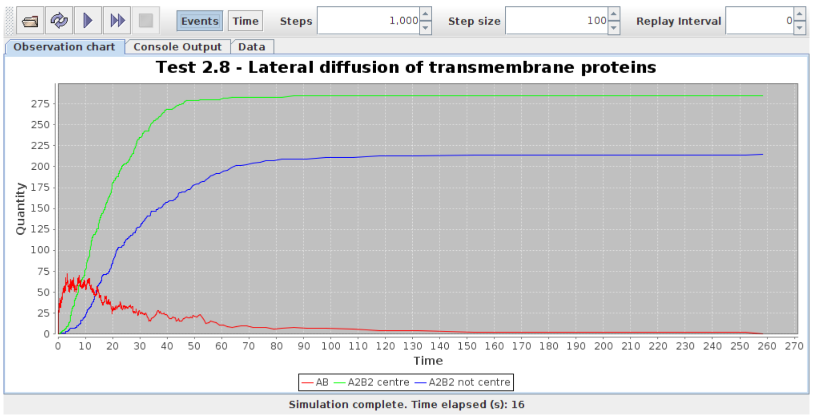


Figure 2.3: Sample time series chart output

Chapter 3

Spatial Kappa reference implementation

The algorithm implemented by Spatial Kappa simulator is similar to one described in Elf and Ehrenberg (2004). In this chapter we briefly outlined main steps of the Spatial Kappa simulator algorithm.

Apart from the differences accounted for the rule-based nature of the Spatial Kappa simulator, the major difference between our algorithm and the classical Next Subvolume algorithm is that there there is no difference between application of the reaction event and diffusion event. The reason for different treatment of diffusion events in the original algorithm was the optimisation of the performance: type of event defined one or two voxel states has to be updated. With introduction of multi compartmental complexes the benefit of different treatment has disappeared, so in Spatial Kappa all events are equal in terms of Gillespie algorithm.

3.1 Initialization

Kappa language grammar and the Spatial Kappa extension are defined with ANTLR 3.0 parser generation framework (see also Appendix A). The grammar definition allows to perform number of initialization steps during the parsing phase, but most initialisation performed by simulator itself.

Rule-based nature of the simulator allows to avoid some steps of initialisation, for example, there is no need for connectivity matrix. The channel definitions would play a role of rules defining diffusion reactions in the same way as reaction rules replace reaction definitions.

The main task of initialisation step is the distribution of agents and complexes between voxels. There are two options to do this: model can define exact number of agents and complexes in particular voxel, or total amount of agents in the system. In the later case simulator will distribute agents homogeneously among all voxels of the system.

The next initialisation step is to define possible applications of the reaction and diffusion rules based upon initial state of the system. In this step we need to calculate all possible mappings between rule left-hand-sides and system state and calculate activities for each of that mappings. The activity of the rule is the sum of activities of each of its mappings. Activities for all rules that has no mappings to the state of the system are set to zero.

3.2 Main simulation loop

The main simulation loop of the Spatial Kappa simulator is the same as in Next Subvolume algorithm with modifications due to rule-based nature of the model definition. There are two types of model declarations that need to be processed before ordinary rules: perturbations and infinite rate rules.

3.2.1 Perturbations

Perturbation is a standard Kappa language construct that allow model to change its state when special firing conditions are met. For example, add drug to the system at particular time point, or open the ion channel when transmembrane potential reach some value. In the Spatial Kappa simulator perturbation are checked first in the main simulation loop and if firing conditions are met then modifications to the system state are applied.

3.2.2 Infinite rate rules

It is possible to set the rate constant of rule to infinity in Spatial Kappa model. For example to compare Spatial Kappa model with ordinary Kappa model we could set infinity rate to all diffusion rules. Rules which has rate constants specified by equation could get infinity rate after application of the equation to the system state. It is obvious that infinite rate rules requires special treatment as their application do not progress the system time. In the Spatial Kappa simulator infinite rate rules are applied to the system after perturbation check until either no rules left or state of the system change too much. Application of the infinite rate rule is similar to the finite rate rule, it just do not change the system time, but due to the way they are applied infinite rate rules could significantly slow down the simulation.

3.2.3 Finite rate rules

After application of infinite rate rules system pick one of the rules and one of that rule mappings and apply it to the system. The process of selection of the rule and one of its mapping is the same as in Next Subvolume method: simulator calculates total activity $r_{tot} = \sum_i r_i$ or $m^i_{tot} = \sum_j m^i_j$ and generate random number $rand$ uniformly distributed between 0 and 1. Choose rule (mapping) if $rand < r_i/r_{tot}$ ($rand < m^i_j/m^i_{tot}$) to pick the rule (mapping) for application.

After rule application simulator change the state of the system according to the rule right-hand-side, progress the system time by the same $\Delta\tau$ as in Next Subvolume method and recalculate the rule mappings and their activities.

Appendix A

Spatial Kappa Grammar

The following is a cut down version of the Antlr grammar used in the Kappa simulator. The syntax has been trimmed for readability, as the original Antlr grammar has artificial constructs for dealing with left recursion, etc. It is read basically as BNF notation with assignments (`variable=bnfConstruct`). The existing basic Kappa grammar is shown in `black`, with the spatial constructs shown as `blue`.

```
prog :
  (line)*

line :
  agentDecl NEWLINE!
  | compartmentDecl NEWLINE!
  | channelDecl NEWLINE!
  | ruleDecl NEWLINE!
  | initDecl NEWLINE!
  | plotDecl NEWLINE!
  | obsDecl NEWLINE!
  | varDecl NEWLINE!
  | modDecl NEWLINE!
  | COMMENT!
  | NEWLINE!

ruleDecl :
  label? transition rate

transition :
   (source=location)? CHANNEL_TRANSITION channelName=id (target=location)?
  | (a=agentGroup)? CHANNEL_TRANSITION channelName=id (b=agentGroup)?
  | (a=agentGroup)? FORWARD_TRANSITION (b=agentGroup)?

agentGroup :
  '(' agentGroup ')'
  | location? agent (',' agent)*

agent :
  id (location)? '(' (agentInterface (',' agentInterface)*)? ')'

agentInterface :
  id state? link?
```

```
state :
  '~' stateId

link :
  '!' INT ( ':' channelName=id)?
  | '!' '_' ( ':' channelName=id)?
  | '?'

rate :
  '@' varAlgebraExpr

initDecl :
  '%init:' (INT | label) agentGroup

agentDecl :
  '%agent:' agentName=id '(' (agentDeclInterface (',' agentDeclInterface)*)? ')'

agentDeclInterface :
  id state*

compartmentDecl :
  '%compartment:' name=id (type=id)? ('[' INT ']')*

channelDecl :
  '%channel:' linkName=id channel
  | '%channel:' linkName=id '(' channel ')' ('+' '(' channel ')')*

channel :
  (type=id)? source=locations FORWARD_TRANSITION target=locations

locations :
  location (',' location)*

location :
  ':' 'fixed'
  | ':' sourceCompartment=id compartmentIndexExpr*

compartmentIndexExpr :
  '[' '?' ']'
  | '[' cellIndexExpr ']'

plotDecl :
  '%plot:' label

obsDecl :
  '%obs:' label varAlgebraExpr
  | '%obs:' voxel? label? agentGroup

varDecl :
  '%var:' label varAlgebraExpr
  | '%var:' voxel? label agentGroup

varAlgebraExpr :
  a=varAlgebraMultExpr (op=operator_add b=varAlgebraMultExpr )*

varAlgebraMultExpr :
  a=varAlgebraExpExpr (op=operator_mult b=varAlgebraExpExpr )*
```

```
varAlgebraExpExpr :
  a=varAlgebraAtom ('^' b=varAlgebraExpExpr )*

varAlgebraAtom :
  '(' varAlgebraExpr ')'
  | number
  | label
  | '[' 'inf' ']'
  | '[' 'pi' ']'
  | '[' 'Tsim' ']'
  | '[' 'Tmax' ']'
  | '[' 'Emax' ']'
  | '[' 'T' ']'
  | '[' 'E' ']'
  | operator_unary varAlgebraAtom
  | '|' agentGroup '|'

modDecl :
  '%mod:' 'repeat' perturbationExpression 'until' booleanExpression
  | '%mod:' perturbationExpression

perturbationExpression
  '(' perturbationExpression ')'
  | booleanExpression 'do' effects

booleanExpression :
  a=booleanAtom (op=booleanOperator b=booleanAtom )*

booleanOperator :
  '&&' | '||'

relationalOperator :
  '<' | '>' | '=' | '<>'

booleanAtom :
  '(' booleanExpression ')'
  | '[' 'true' ']'
  | '[' 'false' ']'
  | '[' 'not' ']' booleanAtom
  | a=varAlgebraExpr relationalOperator b=varAlgebraExpr

effects
  '(' effects ')'
  | effect (';' effect)*

effect :
  '$SNAPSHOT'
  | '$STOP'
  | '$ADD' varAlgebraExpr agentGroup
  | '$DEL' varAlgebraExpr agentGroup
  | '$UPDATE' label varAlgebraExpr

cellIndexExpr :
  a=cellIndexAtom operator_cell_index b=cellIndexAtom
  | a=cellIndexAtom

cellIndexAtom :
  '(' cellIndexExpr ')'
  | INT
  | id
```

```
id :
  ( 'a'..'z' | 'A'..'Z' ) ( ALPHANUMERIC | '_' | '-' | '+' )*

stateId :
  ALPHANUMERIC

label :
  LABEL

number :
  ( INT | FLOAT )

operator_cell_index :
  ( '+' | '*' | '-' | '/' | '%' | '^' )

operator_unary :
  '[' 'log' ']'
  | '[' 'sin' ']'
  | '[' 'cos' ']'
  | '[' 'tan' ']'
  | '[' 'sqrt' ']'
  | '[' 'exp' ']'
  | '[' 'int' ']'

operator_mult :
  '*' | '/' | '[' 'mod' ']'

operator_add :
  '+' | '-'

CHANNEL_TRANSITION :
  '->:'

FORWARD_TRANSITION :
  '->'

INT :
  NUMERIC

FLOAT :
  NUMERIC '.' NUMERIC EXPONENT?
  | '.' NUMERIC EXPONENT?
  | NUMERIC EXPONENT

ALPHANUMERIC :
  ( NUMERIC | 'a'..'z' | 'A'..'Z' )+

NUMERIC :
  ('0'..'9')+

EXPONENT :
  ('e'|'E') ('+'|'-')? NUMERIC

LABEL :
  '\'' .* '\''

COMMENT :
  '#' ~( '\n' | '\r' )*

NEWLINE :
  '\r'? '\n' | '\r'
```

```
WS :
  ( ' ' | '\t' | '\\' NEWLINE )+
```

Appendix B

Spatial Kappa Examples

B.1 Spatial Kappa patterns

The following are generic shapes, with their equivalent Spatial Kappa representations. These are intended to be copied during model development. Explicitly specified models are shown first to demonstrate the syntax, then more concise versions are shown where possible.

B.1.1 1 dimensional patterns

Linear array

```
%compartment: array [n] # Replace n with length of array
%channel: intra-array (:array [x] -> :array [x +1]) + (:array [x] -> :array [x -1])
```

Circle

```
%compartment: circle [n] # Replace n with number of cells in circle
%channel: intra-circle \
    (:circle [x] -> :circle [x +1]) + (:circle [x] -> :circle [x -1]) + \
    (:circle [n -1] -> :circle [0]) + (:circle [0] -> :circle [n -1]) # Replace n-1 as above
```

B.1.2 2 dimensional surfaces

Rectangular mesh

There are 2 variants here, 4-way linked and 8-way linked.

```
%compartment: mesh [n][m] # Replace n and m with dimensions of mesh

# 4-way diffusion
%channel: intra-mesh \
    (:mesh [x][y] -> :mesh [x +1][y]) + (:mesh [x][y] -> :mesh [x -1][y]) + \
    (:mesh [x][y] -> :mesh [x][y +1]) + (:mesh [x][y] -> :mesh [x][y -1])

# or 8-way diffusion
%channel: intra-mesh \
    (:mesh [x][y] -> :mesh [x +1][y]) + (:mesh [x][y] -> :mesh [x -1][y]) + \
    (:mesh [x][y] -> :mesh [x][y +1]) + (:mesh [x][y] -> :mesh [x][y -1]) + \
    (:mesh [x][y] -> :mesh [x +1][y +1]) + (:mesh [x][y] -> :mesh [x -1][y -1]) + \
    (:mesh [x][y] -> :mesh [x +1][y -1]) + (:mesh [x][y] -> :mesh [x -1][y +1])
```

These can also be specified using channel types as follows

```
%compartment: mesh [n][m] # Replace n and m with dimensions of mesh

# 4-way diffusion
%channel: intra-mesh EdgeNeighbour :mesh -> :mesh

# or 8-way diffusion
%channel: intra-mesh Neighbour :mesh -> :mesh
```

Hexagonal mesh

Again, 2 variants depending on what overall shape is required. The first form has a simpler representation of intra-compartment links, but the overall structure is rhomboid, whereas the second produces an overall rectangular shape at the expense of more complicated link statements.

The second variant demonstrates handling of alternate odd-even linkage depending on the column of the structure.

```
%compartment: mesh [n][m] # Replace n and m with dimensions of mesh

# Variant 1 - rhomboid mesh
%channel: intra-mesh \
    (:mesh [x][y] -> :mesh [x][y +1]) + (:mesh [x][y] -> :mesh [x][y -1]) + \
    (:mesh [x][y] -> :mesh [x +1][y]) + (:mesh [x][y] -> :mesh [x -1][y]) + \
    (:mesh [x][y] -> :mesh [x +1][y +1]) + (:mesh [x][y] -> :mesh [x -1][y -1])

# Variant 2 - rectangular mesh
%channel: intra-mesh \
    (:mesh [x][y] -> :mesh [x][y +1]) + (:mesh [x][y] -> :mesh [x][y -1]) + \
    (:mesh [x][y] -> :mesh [x +1][y]) + (:mesh [x][y] -> :mesh [x -1][y]) + \
    (:mesh [x][y] -> :mesh [x +1][(y +1)-(2*(x%2))]) + \
    (:mesh [x][y] -> :mesh [x -1][(y -1)+(2*((x -1)%2))])

# The above statement alternates [x +1][y +1] and [x +1][y -1] as x increases
```

Variant 2 can also be specified using channel types as follows

```
%compartment: mesh [n][m] # Replace n and m with dimensions of mesh

# Variant 2 - rectangular mesh
%channel: intra-mesh Hexagonal :mesh -> :mesh
```

Cylinder and torus

By connecting together the top and bottom edges of a mesh as described above, we get a cylinder. By also connecting together the left and right edges we get a torus.

```
%compartment: mesh [n][m] # Replace n and m with dimensions of mesh

# 4-way diffusion mesh
%channel: intra-mesh \
    (:mesh [x][y] -> :mesh [x +1][y]) + (:mesh [x][y] -> :mesh [x -1][y]) + \
    (:mesh [x][y] -> :mesh [x][y +1]) + (:mesh [x][y] -> :mesh [x][y -1])
%channel: intra-mesh mesh [x][y] <-> mesh [x+1][y]
%channel: intra-mesh mesh [x][y] <-> mesh [x][y+1]

# cylinder
```

```
%channel: intra-mesh \
    (:mesh [x][y] -> :mesh [x +1][y]) + (:mesh [x][y] -> :mesh [x -1][y]) + \
    (:mesh [x][y] -> :mesh [x][y +1]) + (:mesh [x][y] -> :mesh [x][y -1]) + \
    (:mesh [x][y] -> :mesh [x][m -1]) + (:mesh [x][y] -> :mesh [x +1][0])
# Replace m-1 as above

# torus
%channel: intra-mesh \
    (:mesh [x][y] -> :mesh [x +1][y]) + (:mesh [x][y] -> :mesh [x -1][y]) + \
    (:mesh [x][y] -> :mesh [x][y +1]) + (:mesh [x][y] -> :mesh [x][y -1]) + \
    (:mesh [x][m -1] -> :mesh [x][0]) + (:mesh [x][0] -> :mesh [x][m -1]) + \
    (:mesh [n -1][y] -> :mesh [0][y]) + (:mesh [0][y] -> :mesh [n -1][y])
# Replace n-1 as above
```

The predefined compartment `OpenCylinder` allows creation of a cylinder with closed ends

```
%compartment: openCylinder  OpenCylinder  [10][8] [2]    # [diameter][length] [thickness]
```

B.2 Sample Spatial Kappa models

The following are a couple of simple spatial kappa models to demonstrate the use of the language features.

B.2.1 2d diffusion model

This model shows simple diffusion from a point for three distinct species.

```
%agent: A()
%agent: B()
%agent: C()

%compartment: cytosol [30][30]

# 6-way diffusion
%channel: hexagonal \
    (:cytosol [x][y] -> :cytosol [x][y +1]) + (:cytosol [x][y] -> :cytosol [x][y -1]) + \
    (:cytosol [x][y] -> :cytosol [x +1][y]) + (:cytosol [x][y] -> :cytosol [x -1][y]) + \
    (:cytosol [x][y] -> :cytosol [x +1][(y +1)-(2*(x%2))]) + \
    (:cytosol [x][y] -> :cytosol [x -1][(y -1)+(2*((x -1)%2))])

'diffusion' ->:hexagonal @ 0.4

%init: 10000 :cytosol[10][10] A()
%init: 10000 :cytosol[10][14] B()
%init: 10000 :cytosol[14][14] C()

%obs: 'Red' A()
%obs: 'Green' B()
%obs: 'Blue' C()
```

A more concise version would be

```
%agent: A()
%agent: B()
%agent: C()

%compartment: cytosol [30][30]

# 6-way diffusion
%channel: hexagonal Hexagonal :cytosol -> :cytosol

'diffusion' :cytosol ->:hexagonal :cytosol @ 0.4

%init: 10000 :cytosol[10][10] A()
%init: 10000 :cytosol[10][14] B()
%init: 10000 :cytosol[14][14] C()

%obs: 'Red' A()
%obs: 'Green' B()
%obs: 'Blue' C()
```

B.2.2 Bi-trivalent binding model

A variant of bi-trivalent binding test model (Yang et al., 2008). This spatial model allows unbound species to diffuse through the compartment, but bound species remain within a cell of the compartment.

```
%agent: A(a,b,bindings~0~1~2)
%agent: B(a,b,c)

%compartment: cytosol [20][20]

# 6-way diffusion
%channel: hexagonal \
    (:cytosol [x][y] -> :cytosol [x][y +1]) + (:cytosol [x][y] -> :cytosol [x][y -1]) + \
    (:cytosol [x][y] -> :cytosol [x +1][y]) + (:cytosol [x][y] -> :cytosol [x -1][y]) + \
    (:cytosol [x][y] -> :cytosol [x +1][(y +1)-(2*(x%2))]) + \
    (:cytosol [x][y] -> :cytosol [x -1][(y -1)+(2*((x -1)%2))])

'diffusion-A' A(bindings~0) ->:hexagonal A(bindings~0) @ 0.1
'diffusion-B' B(a,b,c) ->:hexagonal B(a,b,c) @ 1

A(a,b,bindings~0), B(a) -> A(a!1,b,bindings~1),B(a!1) @ 1
A(a,b,bindings~0), B(b) -> A(a!1,b,bindings~1),B(b!1) @ 1
A(a,b,bindings~0), B(c) -> A(a!1,b,bindings~1),B(c!1) @ 1
A(a,b,bindings~0), B(a) -> A(a,b!1,bindings~1),B(a!1) @ 1
A(a,b,bindings~0), B(b) -> A(a,b!1,bindings~1),B(b!1) @ 1
A(a,b,bindings~0), B(c) -> A(a,b!1,bindings~1),B(c!1) @ 1
A(a,b!_,bindings~1), B(a) -> A(a!1,b!_,bindings~2),B(a!1) @ 1
A(a,b!_,bindings~1), B(b) -> A(a!1,b!_,bindings~2),B(b!1) @ 1
A(a,b!_,bindings~1), B(c) -> A(a!1,b!_,bindings~2),B(c!1) @ 1
A(a!_,b,bindings~1), B(a) -> A(a!_,b!1,bindings~2),B(a!1) @ 1
A(a!_,b,bindings~1), B(b) -> A(a!_,b!1,bindings~2),B(b!1) @ 1
A(a!_,b,bindings~1), B(c) -> A(a!_,b!1,bindings~2),B(c!1) @ 1

A(a!1,b,bindings~1),B(a!1) -> A(a,b,bindings~0), B(a) @ 0.01
A(a!1,b,bindings~1),B(b!1) -> A(a,b,bindings~0), B(b) @ 0.01
A(a!1,b,bindings~1),B(c!1) -> A(a,b,bindings~0), B(c) @ 0.01
A(a,b!1,bindings~1),B(a!1) -> A(a,b,bindings~0), B(a) @ 0.01
A(a,b!1,bindings~1),B(b!1) -> A(a,b,bindings~0), B(b) @ 0.01
A(a,b!1,bindings~1),B(c!1) -> A(a,b,bindings~0), B(c) @ 0.01
A(a!1,b!_,bindings~2),B(a!1) -> A(a,b!_,bindings~1), B(a) @ 0.01
A(a!1,b!_,bindings~2),B(b!1) -> A(a,b!_,bindings~1), B(b) @ 0.01
A(a!1,b!_,bindings~2),B(c!1) -> A(a,b!_,bindings~1), B(c) @ 0.01
A(a!_,b!1,bindings~2),B(a!1) -> A(a!_,b,bindings~1), B(a) @ 0.01
A(a!_,b!1,bindings~2),B(b!1) -> A(a!_,b,bindings~1), B(b) @ 0.01
A(a!_,b!1,bindings~2),B(c!1) -> A(a!_,b,bindings~1), B(c) @ 0.01

%init: 600 A(a,b,bindings~0)
%init: 400 B(a,b,c)

%obs: 'Red' A(bindings~2)
%obs: 'Green' A(bindings~1)
%obs: 'Blue' A(bindings~0)
```

Bibliography

Elf, J. and Ehrenberg, M. (2004). Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Syst. Biol*, 1(2):230.

Feret, J. and Krivine, J. (2012). *KaSim Manual*.

Gilbert, D. et al. (2010). Jfreechart website http://www.jfree.org/jfreechart/.

Yang, J., Monine, M., Faeder, J., and Hlavacek, W. (2008). Kinetic Monte Carlo method for rule-based modeling of biochemical networks. *Physical Review E*, 78(3):31910.