Spatial Kappa Translator
User Guide
v2.0.1

Donal Stewart
DemonSoft Ltd

February 28, 2013

Contents

Chapter 1

Spatial Kappa Translator User Guide

1.1 Obtaining the application

The application 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/`.

1.2 Using the translator

The translator converts Kappa model files from the current version of Spatial Kappa (v2.0.1) to the version of Kappa supported by the current release of KaSim (v2.01). It takes a single input kappa file and an optional output file name.

The generated output file can then be simulated using KaSim.

1.2.1 Running the executable jar

The simulator can be started by running the executable jar file:
`java -jar SpatialKappaTranslator-v2.0.1.jar  <input file path> [<output file path>]`

Double clicking the jar file usually works too.

1.2.2 Running from the Eclipse project

The main class of the simulator is
`org.demonsoft.spatialkappa.tools.SpatialTranslator`

Running as a Java Application with the command line arguments
`<input file path> [<output file path>]` will execute the translator.

Chapter 2

Kappa Language Extensions

Note on terminology. In the following 'voxel' means a subunit of a compartment. Current Kappa rules are referred to as 'transform' rules as they cause some form of transformation of their substrates, be it change of agent state, agent creation or deletion, agent binding or unbinding. Species is mostly used to refer to a particular agent or complex in the model.

2.1 Existing Kappa language

The current Kappa language consists of the following constructs, as described in the KaSim v2.01 documentation (Krivine, 2012). A more formal description of the language grammar (including the spatial extensions) is given in appendix A.

Comments:

```
# This is a comment
```

Agents:

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

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 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(..)
```

Observables and named variables: Variables can be referenced in perturbation calculations, but do not show in outputs as observations.

```
%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()
```

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
```

2.2 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 voxel, 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.
- 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.
- 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.
- A means of **locating transform 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 linkage structures. The rates of transport should be general to all species, or species specific.

Additional concepts

- **Granularity** within compartments. It would be useful to be able to specify locations at the level of compartments or single cells 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.

2.3 New language constructs

A full BNF description of the extended Kappa grammar is given in appendix A.

2.3.1 Compartments and voxels

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

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

For example

```
%compartment: SingleCell
%compartment: 1dArray [10]       # 10 voxels in size
%compartment: 2dArray [10][5]    # 10x5 voxels in size
%compartment: 3dArray [10][5][4] # 10x5x4 voxels in size
```

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

```
id ( '[' mathExpr ']' )*
```

where

```
mathExpr :
  mathAtom OPERATOR mathAtom
  | mathAtom

mathAtom :
  '(' mathExpr ')'
  | INT
  | VARIABLE_NAME

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

For example

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

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

In all situations where compartment references 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 voxel indices to match the dimensions of the compartment. Also, variable names are only permitted in compartment references within channel definitions, described below.

2.3.2 Channels

The structure of a compartment is further defined by how voxels within the compartment are linked to each other and to connected compartments. This linkage is defined as follows

```
'%channel:' linkName=id sourceCompartment=locationExpr '->' targetCompartment=locationExpr
| '%channel:' linkName=id
    '(' sourceCompartment=locationExpr '->' targetCompartment=locationExpr ')'
    ( '+' '(' sourceCompartment=locationExpr '->' targetCompartment=locationExpr ')' )*
```

Where `locationExpr` is as described above. For example

```
%compartment: 2dArray [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 \
    (2dArray[x][y] -> 2dArray[x+1][y]) + (2dArray[x][y] -> 2dArray[x -1][y]) +\
    (2dArray[x][y] -> 2dArray[x][y+1]) + (2dArray[x][y] -> 2dArray[x][y -1]) +\
    (2dArray[0][y] -> 2dArray[9][y])   + (2dArray[9][y] -> 2dArray[0][y]) +\
    (2dArray[x][0] -> 2dArray[x][199]) + (2dArray[x][199] -> 2dArray[x][0])
```

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

Compartment references 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. Compartment references 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.

Multiple channel definitions can use the same label (like `meshlinks` in the example above), in which case references to that label elsewhere in the model

acts on the union of all the channel definitions. This allows complex channel definitions, for example compartments constructed in the form of a circle or sphere, to be referenced concisely in transport rules.

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

2.3.3 Transport rules

These rules define the rate at which species move from one compartment (or cell) to another.

```
'%transport:' (transportName=LABEL)? linkName=id (agentGroup)? kineticExpr
```

where

```
agentGroup :
  agent (',' agent)*

kineticExpr :
  '@' a=rateValueExpr

rateValueExpr :
  number | '$INF'   # $INF means infinite rate
```

See appendix A for the definition of the remaining constructs. For examples of usage

```
# All species diffusing around the torus defined above
%transport: 'general diffusion' meshlinks @ 0.1

# Immediate exit of the named species through a uni-directional channel.
# Note the label for the transport is optional.
%transport: CaChannel Calcium() @ $INF
```

In the second example above, care would need to be taken in defining the channel `CaChannel` in the correct direction.

2.4 Additions to existing language constructs

New clauses were added to existing Kappa language constructs to allow the use of compartments and links. In all cases, existing Kappa syntax is still valid. Where necessary, the optional label in existing Kappa statements is non-optional in statements using compartments to avoid ambiguity. When basic Kappa syntax is used in the context of a spatially aware model, the statement is generally taken to apply to all compartments in the model equally.

2.4.1 Transform rules

The following transform statement was added

```
transformName=LABEL locationExpr (a=agentGroup)? '->' (b=agentGroup)? kineticExpr
```

For example

```
# mRNA degradation happens outside the nuclear membrane
'degradation' cytosol mRNA() -> @ 6.21

# Heating of the agent occurs in cell 0 of the cytosol compartment
'heating' cytosol[0] A(state~blue) -> A(state~red) @ 1.0
```

As mentioned before valid locations are either entire compartments or single compartment cells.

2.4.2 Initial values

The following initial value statement was added

```
initExpr :
  '%init:' INT locationExpr agentGroup
```

For example

```
# Distribute 2000 blue agents within the cells of cytosol equally
%init: 2000 cytosol A(state~blue)

# Add 500 red agents to cell 5 of the cytosol compartment
%init: 500 cytosol[5] A(state~red)
```

2.4.3 Observations

The following observation statement was added

```
obsExpr :
  '%obs:' LABEL locationExpr agentGroup
```

For example

```
# Count all blue agents in all cells of the cytosol
%obs: 'cytosol blue' cytosol A(state~blue)

# Count all red agents in cell 0 of the cytosol compartment
%obs: 'red[0]' cytosol[0] A(state~red)
```

2.4.4 Located agents

Within transform, observation and initial value statements, the user can specify location of individual agents using the following syntax

```
agent :
  id (':' locationExpr)? ('(' (iface (',' iface)*)? ')')?
```

For example

```
%init: 1000 A:cytosol(s!1:out)
```

Where a statement contains both rule and agent locations, the agent locations override the rule locations. For example

```
membrane Ligand(r),Receptor-membrane(dl!1:in,ligand),\
    Receptor-cytosol:cytosol(dl!1,state~inactive) -> \
    Ligand:membrane(r!2),Receptor-membrane(dl!1:in,ligand!2),\
    Receptor-cytosol:cytosol(dl!1,state~active) @ 'link rate'
```

2.4.5 Located agent links

When writing agent links, channels may be named. Doing so means that the link between two agents spans a channel between two locations. This is useful in defining trans-membrane complexes for example. Not specifying a channel in an agent link means that the agents comprising the link must be in the same compartment. Agent link channel specifications and agent location specifications must be consistent with each other to be considered valid.

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

For example

```
%channel: out cytosol [x] -> membrane [x]
%obs: 'dimer AB' A:cytosol(s!1:out),B:membrane(s!1)
```

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

Appendix A

Spatial Kappa Grammar

The following is a cut down version of the Antlr grammar used in the Kappa translator. 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 :
  ruleExpr NEWLINE!
  | compartmentExpr NEWLINE!
  | channelDecl NEWLINE!
  | transportExpr NEWLINE!
  | initExpr NEWLINE!
  | plotExpr NEWLINE!
  | obsExpr NEWLINE!
  | varExpr NEWLINE!
  | modExpr NEWLINE!
  | agentExpr NEWLINE!
  | COMMENT!
  | NEWLINE!

ruleExpr :
  label? transformExpr kineticExpr
  | label? locationExpr transformExpr kineticExpr

transformExpr :
  (a=agentGroup)? '->' (b=agentGroup)?

agentGroup :
  agent (',' agent)*

agent :
  id (':' locationExpr)? ('(' (iface (',' iface)*)? ')')?

iface :
  id stateExpr? linkExpr?

stateExpr :
```

```
  '~' id

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

kineticExpr :
  '@' varAlgebraExpr

initExpr :
  '%init:' INT locationExpr? agentGroup
  | '%init:' label locationExpr? agentGroup

agentExpr :
  '%agent:' agentName=id ('(' (agentIfaceExpr (',' agentIfaceExpr)*)? ')')?

agentIfaceExpr :
  id stateExpr*


compartmentExpr :
  '%compartment:' id ('[' INT ']')*

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

channelExpr :
  sourceCompartment=locationExpr '->' targetCompartment=locationExpr

transportExpr :
  '%transport:' (transportName=label)? linkName=id (agentGroup)? kineticExpr

locationExpr :
  sourceCompartment=id compartmentIndexExpr*

compartmentIndexExpr :
  '[' cellIndexExpr ']'

plotExpr :
  '%plot:' label

obsExpr :
  '%obs:' label? agentGroup
  | '%obs:' label? locationExpr agentGroup

varExpr :
  '%var:' label varAlgebraExpr
  | '%var:' label agentGroup

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

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

varAlgebraExpExpr :
```

```
  a=varAlgebraAtom operator_exp b=varAlgebraExpExpr
  | a=varAlgebraAtom

varAlgebraAtom :
  '(' varAlgebraExpr ')'
  | number
  | label
  | '[' 'inf' ']'
  | '[' 'pi' ']'
  | '[' 'T' ']'
  | '[' 'E' ']'
  | operator_unary varAlgebraAtom
  | operator_binary_prefix a=varAlgebraAtom b=varAlgebraAtom

modExpr :
  '%mod:' booleanExpression 'do' effect untilExpression?


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

booleanOperator :
  '&&' | '||'

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


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

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

untilExpression :
  'until' booleanExpression

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

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

id :
  ALPHANUMERIC ( ALPHANUMERIC | '_' | '-' )*

label :
  LABEL
```

```
number :
  ( INT | FLOAT )

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

operator_exp :
  | '^'

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

operator_binary_prefix :
  '[' 'mod' ']'

operator_mult :
  '*' | '/'

operator_add :
  '+' | '-'


INT :
  NUMERIC

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

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

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

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

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

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

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

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

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.

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])
```

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
```

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
```

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: 6way \
    (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))])

%transport: 'diffusion-A' 6way A() @ 0.4
%transport: 'diffusion-B' 6way B() @ 0.4
%transport: 'diffusion-C' 6way C() @ 0.4

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

%obs: 'Red' cytosol A()
%obs: 'Green' cytosol B()
%obs: 'Blue' cytosol 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: 6way \
    (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))])

%transport: 'diffusion-A' 6way A(bindings~0) @ 0.1
%transport: 'diffusion-B' 6way 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 cytosol A(a,b,bindings~0)
%init: 400 cytosol B(a,b,c)

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

Bibliography

Krivine, J. (2012). *KaSim manual (v 2.0)*, v2.0 edition.

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.