

SSC Reference Manual

Mieszko Lis, Maxim N. Artyomov, Srinivas Devadas, and Arup Chakraborty

SSC version 0.4

Contents

1	Introduction	1
1.1	Related work	1
2	Writing SSC models	1
2.1	Compounds	1
2.2	Patterns	2
2.3	Initial species	4
2.4	Reactions	4
2.4.1	Changing properties	4
2.4.2	Forming and breaking bonds	5
2.4.3	Creating and destroying compounds	5
2.4.4	Multiple actions	5
2.5	Records	6
2.6	Expansion limits	6
2.7	Geometry and diffusion	7
2.7.1	Constructive Solid Geometry	7
2.7.2	Regions	8
2.7.3	Diffusion	8
2.8	Named constants	9
2.9	Comments	9
3	Simulating with SSC	9
3.1	Compiling and debugging	9
3.2	Simulating	10
4	Caveats	11
4.1	Patterns and reaction	11
5	References	11

1 Introduction

The Stochastic Simulation Compiler (SSC) is exact stochastic simulations of biochemical reaction networks. The models are written in a succinct, intuitive format, where reactions are specified with patterns, which mention only the part of the compound relevant to a given reaction, and correspond to an intuitive view of biochemical reactions.

SSC compiles the model into native executable code (much like, say, a C compiler compiles a C program), which implements a stochastic simulation algorithm equivalent to Gillespie's Stochastic Simulation Algorithm [G77] but better suited to large reaction networks. During compilation, the full set of compounds in the system is derived from the initial species and the specified reactions, and each reaction is expanded for all the actual compounds it can apply to.

1.1 Related work

The high-level modeling language and combinatorial expansion form a Term Rewriting System on graphs representing compounds; the principles and some of the syntax are therefore similar to those of κ [DL04] and BioNetGen [F09].

The simulation algorithm, while equivalent to the SSA [G77], scales significantly better (i.e., logarithmically) as the size of the model grows by storing the reaction propensities in a balanced heap, similarly to [W06] or [LP06].

2 Writing SSC models

SSC models consist of a set of initial species, a list of reactions, and indications of which species to record in the simulation output; all of these are, in turn, built out of *compounds* and *patterns*. The reactions are simulated in a container, possibly divided into several regions, whose geometry is specified using Constructive Solid Geometry (CSG).

Naturally, SSC can be used to model any biochemical network. For clarity, however, we will use an example model of the T-cell receptor (TCR) and its ligand, a MHC-peptide complex, throughout this manual. TCR can bind and unbind from MHC, and, upon productive binding, the internal domain of TCR can be phosphorylated at multiple sites through a series of events.

2.1 Compounds

The simplest compounds contain single species. Each species must, at the very least, have a *type* (which must be a single word starting with a letter); thus, to represent a T-cell receptor, we might write any of

```
TCR
TCellReceptor
T_cell_Receptor
```

More sophisticated species may also have *properties*, each of which has a name and a value; in biochemical networks, properties often represent activation sites that may be in one of several states (phosphorylation, methylation, etc.). In our example, we could model a MHC-peptide complex with an MHC species and a property indicating what kind of a peptide is bound, writing

```
MHC(peptide="none")
MHC(peptide="cognate")
MHC(peptide="noncognate")
```

for the different kinds of MHCs (like species types, property names must be single words, while values may be numbers or strings delimited by quotation marks). Similarly, we could model TCR phosphorylation state with a property, writing

```
TCR(phosphorylated_sites=0)
TCR(phosphorylated_sites=1)
TCR(phosphorylated_sites=2)
```

Alternately, we might choose to model each TCR phosphorylation site separately, writing, e.g.,

```
TCR(phosphorylation_site_1="on", phosphorylation_site_2="off")
```

and so on; the choice of which to select will depend on what kind of reactions are most convenient to write for a given phenomenon.

Compounds consisting of multiple species are created by associating a named *site* in each of the bonded species with a unique *bond*. For example, our TCR–MHC complex might look like

```
TCR(groove#1) MHC(cleft#1)
```

where bond 1 connects *TCR*'s site *groove* with *MHC*'s *cleft* site (like property values, bond identifiers may be numbers or strings). Bonds may, of course, be combined with properties, e.g.,

```
MHC(peptide="cognate", cleft#1) TCR(groove#1)
```

for arbitrarily complex compounds.

In practice, only the initial species in a given model need to be written down; the rest are automatically derived by SSC from the initial species and the reactions specified in the model.

2.2 Patterns

While compounds form the basis of the actual simulation, *patterns* are what makes SSC powerful. A pattern omits information that is irrelevant to, say, a reaction, and thus allows describing a whole class of compounds in one fell swoop. From a modeling standpoint, patterns allow us to take advantage of the observation that biological molecules (e.g., proteins) often have many modification or bonding sites which are in different parts of the molecule and are activated or bonded more or less independently.

The simplest patterns look deceptively like species, e.g.,

```
MHC
```

As one might expect, this pattern will match species *MHC*. But, since it does not mention any properties, it will also match *MHC* with any properties set to any values: that is, it will match an empty *MHC* as well as an *MHC* bearing any peptide, e.g.,

```
MHC(peptide="noncognate")
```

as well as

```
MHC(peptide="none")
```

More interestingly, since the pattern doesn't mention any bonds, it will also match *MHC* that is part of any compound, e.g., *MHC*–*TCR* compounds, as in

```
MHC(cleft#1) TCR(groove#1)
```

or

```
MHC(peptide="cognate", cleft#1) TCR(groove#1, phosph_sites=1)
```

Like compounds, patterns can specify properties and bonds, in which case they only match compounds where the specified properties have the required values, and the various species are connected as the bonds describe (properties and bonds that are not mentioned are ignored). Thus, we can be more specific, and limit the match to *MHCs* not carrying any peptides:

```
MHC(peptide="none")
```

will match any *MHC* where *peptide* is "none", including

```
MHC(peptide="none")
```

and

```
MHC(peptide="none", cleft#1) TCR(groove#1, phosph_sites=2)
```

but not

```
MHC(peptide="cognate")
```

because the value of *peptide* does not match the pattern.

Similarly, bonds in patterns enforce the species types, bond site names, and how the species are connected (but not the actual bond names), ignoring any bonds that are not in the pattern. For example, we could look for any *MHC-TCR* complexes using

```
MHC(cleft#1) TCR(groove#1)
```

which will match

```
MHC(peptide="on", cleft#1) TCR(groove#1, phosph_sites=2)
```

and

```
MHC(peptide="none", corec#1, cleft#2) CD4(mhc#1) TCR(groove#2, phosph_sites=2)
```

but not

```
MHC(corec#1) TCR(groove#1)
```

(because *MHC*'s bond site name, *corec* does not correspond to the *cleft* required by the pattern) or

```
MHC(cleft#1) T_cell_Receptor(groove#1)
```

(because the species type is wrong: *T_cell_Receptor* instead of *TCR*). A pattern can also require that a bond site be vacant, in which case an empty # is written; e.g., we could match an *MHC* not bonded to a *TCR* with

```
MHC(cleft#)
```

which will match

```
MHC(peptide="none")
```

and

```
MHC(peptide="cognate", corec#1) CD4(mhc#1)
```

but not

```
MHC(peptide="cognate", cleft#1) TCR(groove#1, phosph_sites=0)
```

because *cleft* is not empty as required.

Finally, it's sometimes convenient to demand that a bond site be connected to *something*, regardless of what's on the other side of the bond; we write this by following the usual # with the don't-care symbol *_* (underscore). So we could look for an *MHC* molecule bound to *something* at its *cleft* site with

```
MHC(cleft#_)
```

which will match

```
MHC(peptide="cognate", cleft#1) TCR(groove#1, phosph_sites=0)
```

but not

```
MHC(peptide="none")
```

or

```
MHC(peptide="cognate", corec#1) CD4(mhc#1)
```

both of which have an unbonded *cleft* site.

2.3 Initial species

The initial species indicate how many molecules of each species is available at the start of the simulation. The syntax is

```
new compound at count in regions
```

where *count* is a number or a variable specified at simulation time, and *regions* indicates where the species should initially appear. At the beginning of the simulation, the molecules are distributed uniformly at random within *regions* (*regions* may be omitted indicating distribution everywhere). For example, writing

```
new MHC(peptide="none") at 1000
new MHC(peptide="cognate", cleft#1)
    TCR(groove#1, phosph_sites=2) at 10
```

starts the simulation with a total of 1010 molecules, one thousand of which are *MHCs* without any peptide, and further ten are *MHC-TCR* complexes where the *MHC* carries a cognate peptide and the *TCR* has two phosphorylated sites. In spatially heterogeneous simulations, the initial compounds are distributed uniformly at random among the available subvolumes.

There are two other things of note here: one is that the second *new* statement is continued on a new line; this works pretty much everywhere provided that the continuation is indented with respect to the first line of the statement. Secondly, when creating new species, we specified all the *properties* but only the necessary *bonds*; this is because bonds can be empty, while properties must always carry values (indeed, if we skipped a property in a *new* statement, SSC would flag it as an error).

2.4 Reactions

Rather than specifying the full set of compounds on each side of the reaction, reactions in SSC describe local changes to properties and bonds; during compilation, SSC applies each reaction to all possible compound combinations. This corresponds to the intuitive view of many biochemical reactions (e.g., protein phosphorylation at a given site is often possible independently of other phosphorylation sites and regardless of whether the protein is in a complex or free).

Reactions look like this:

```
rxn pattern at propensity -> actions
```

Patterns (which may be empty for zero-order reactions) are described in Section 2.2, while the propensity may be a numeric constant or a variable name (a single word) that can be changed at simulation time. Actions specify the effects of the reactions (bond formation and breaking, property changes, and so on).

2.4.1 Changing properties

The simplest actions modify the properties of a matched species; these are used to represent such modifications as phosphorylation, methylation, and other kinds of activation. To change a property, we have to know how to refer to a specific species matched by a pattern and its properties; this is done by assigning the species to a pattern variable and using `.` (the period) to select a property of that species. First, specific species in a pattern may be given a name by prepending the species type with the variable name and a colon, e.g.,

```
MHC(peptide="cognate", cleft#1) t:TCR(phosph_sites=0, groove#1)
```

will match a *MHC-TCR* dimer (possibly as part of a larger complex) where the *MHC* has a cognate peptide and the *TCR* has two phosphorylated sites; inside this reaction, the *TCR* will be referred to as *t*. (We cannot just use the species name, *TCR*, because this would be ambiguous if there were more than one in the pattern). We can phosphorylate the *TCR* by setting its *phosph_sites* property to 1 with

```
rxn MHC(peptide="cognate", cleft#1) t:TCR(phosph_sites=0, groove#1)
    at 0.1 -> t.phosph_sites = 1
```

This reaction will occur with propensity 0.1, that is, with an average frequency of 0.1 s^{-1} times the number of molecules containing *MHC* in the compartment, and times the number of molecules containing *TCR*.

2.4.2 Forming and breaking bonds

Creating new bonds between two matched species is almost as simple: we name both species and select bond sites just like properties above and employ the # operator to create the bond: for example,

```
rxn m:MHC(cleft#) t:TCR(groove#) at 0.1 -> t.groove # m.cleft
```

will create an *MHC-TCR* dimer by connecting the *cleft* site of the *MHC* molecule with the *groove* site in the *TCR*. Note that the pattern requires both bond sites to be empty (otherwise SSC would flag an error). Also, the pattern specifies neither the peptide bound to *MHC* nor the phosphorylation state of *TCR*, and so the reaction will apply to any *MHC* and any *TCR*.

Bonds are broken just as easily by using the bond identifier used in the pattern; for example, to break the *MHC-TCR* complex formed above, we might write

```
rxn MHC(cleft#1) TCR(groove#1) at 0.1 -> break 1
```

using 1 to identify the bond.

2.4.3 Creating and destroying compounds

To add new molecules to the simulation, we use the *new* action. For example, *TCR* molecules are constantly created (and internalized) on the T-cell surface, so we might write

```
rxn at 0.1 -> new TCR(phosph_sites=0)
```

to indicate that new *TCR* molecules should appear with an average frequency of 0.1 s^{-1} . Because we left the pattern empty, this is a zero-order reaction; we could, of course, just as easily have made the creation of new species dependent on some reactants. Finally, a compound matched by the pattern can be copied, as in this somewhat contrived example:

```
rxn x:MHC(peptide="none") at 0.1 -> new x
```

which simply duplicates any molecules containing an *MHC* without a peptide.

Destroying molecules is just as simple: we name the species in the pattern and employ the *destroy* action to remove it from the simulation; to model the internalization of non-bonded *TCR* into the membrane to match the *new* example above, we could then write

```
rxn t:TCR(groove#) at 0.1 -> destroy t
```

Note that we specified that the bond site *groove* must be empty. If we had not, writing instead

```
rxn t:TCR at 0.1 -> destroy t
```

the pattern would have matched any compound containing a *TCR*, and the reaction would have broken all bonds incident on the *TCR* and destroyed it, leaving the rest of the compound alone. It's also possible to destroy the entire matched compound containing *TCR* by using *destroy complex* instead of *destroy*, e.g.,

```
rxn t:TCR at 0.1 -> destroy complex t
```

which would match and destroy any compound containing *TCR*.

2.4.4 Multiple actions

A reaction may have multiple actions, in which case they appear on separate lines or are delimited by semicolons. For example, if we had chosen to model *TCR* phosphorylation sites separately, we could write a reaction where both are phosphorylated by a fully activated kinase as

```
rxn t:TCR(corec#1, phos_site_1="off", phos_site_2="off")
    CD4(tcr#1, lck="fullyactive") at 0.1 ->
    t.phos_site_1="on"
    t.phos_site_2="on"
```

or

```
rxn t:TCR(corec#1, phos_site_1="off", phos_site_2="off")
    CD4(tcr#1, lck="fullyactive") at 0.1 ->
        t.phos_site_1="on"; t.phos_site_2="on"
```

When written on multiple lines like this, the actions must be indented to the right of the `rxn` keyword.

2.5 Records

Records specify what is to be recorded during simulation. The simplest form,

```
record all
```

records every possible state of each compound in the model separately. Using a pattern-based record, or specifying regions,

```
record pattern in regions
```

allows for more interesting observations. For example, we might be interested in the total amount of doubly phosphorylated *TCR*, whether free or part of a bigger complex, only in region *Membrane*; we would then write

```
record TCR(phosph_sites=2) in Membrane
```

If, on the other hand, we were only interested in the amount of free doubly-phosphorylated *TCR*, we would have to make sure that its bond sites are empty:

```
record TCR(phosph_sites=2, groove#) in Membrane
```

Finally, each record can be given a description, which is then used in simulation output, by following the keyword `record` with some text in quotation marks:

```
record "free fully phosph. TCR everywhere" TCR(phosph_sites=2, groove#)
```

Records not named like this use the pattern it matches as their descriptions.

2.6 Expansion limits

Because SSC automatically derives the set of possible species in the model by starting with the initial species and applying reactions, it's not difficult to make the expansion run forever by building bigger and bigger compounds; e.g.,

```
rxn x:A(right#) y:A(left#) at 0.1 -> x.right # y.left
```

will create bigger and bigger chains of *As*. It's not difficult to run into this: if we chose to model a coreceptor that could bind to both *MHC* and *TCR*, we might write

```
rxn m:MHC(cleft#) t:TCR(groove#) at 0.1 -> m.cleft # t.groove
rxn m:MHC(corec#) c:CD4(mhc#) at 0.1 -> m.corec # c.mhc
rxn t:TCR(corec#) c:CD4(tcr#) at 0.1 -> t.corec # c.tcr
```

which would keep expanding into longer and longer chains of $\dots\text{-CD4-MHC-TCR-CD4}\dots$ and cause SSC to run forever (or, rather, until it ran out of memory). Since the longer the chains, the less likely they are to form, we might decide to cut them off at some number of molecules using the `limit` statement

```
limit at 20
```

which would disallow any complexes containing more than twenty molecules. In general, the `limit` statement has the form

```
limit pattern at constant in regions
```

and can be used to limit only a specific kind of complexes; for example, we could write

```
limit CD4 at 20
```

to limit any compound containing *CD4* to twenty molecules, regardless of region.

2.7 Geometry and diffusion

SSC supports spatially resolved simulations where the simulation compartment of an arbitrary shape is divided into many cubic subvolumes of equal size. The compartment is specified using constructive solid geometry (CSG), and, within each subvolume, reactions are simulated as usual, while species may only diffuse across subvolumes. The simulation compartment may be divided into any number of regions—for example, cell interior, membrane, and exterior—and initial species, reactions, and records may be limited to a specific region.

2.7.1 Constructive Solid Geometry

Constructive Solid Geometry (CSG) allows describing arbitrary shapes using elementary shapes (e.g., spheres, cylinders, cubes), set operations (union, intersection, and difference), and simple geometrical transformations (translation and scaling). For example, a bacillus-shaped cell might be described with the union of two translated spheres and a cylinder, and a cell membrane might be described by scaling the cell down and subtracting it from the original.

Compartment shapes are specified using CSG with a right-handed coordinate system. The simplest shapes can be written with a single CSG primitive:

```
ball radius 5
```

creates a sphere with a five-unit radius around the origin,

```
box width 100 height 100 depth 1
```

results in a box with dimensions $100 \times 100 \times 1$ units centered on the origin, and

```
cylinder radius 5 height 20
```

gives a cylinder around the *z* axis, 20 units in height and with base radius of five units, also centered around the origin. Shapes can be translated in 3D space by specifying the *x*, *y*, and *z* components of the translation; for example,

```
move 3 5 7
  ball radius 5
```

will translate the ball three units along the *x* axis, five along the *y* axis, and seven along the *z* axis. Finally, more complex shapes can be made by joining, intersecting, and subtracting the basic shapes; e.g.,

```
union
  cylinder radius 5 height 20
  move 0 0 -10
    ball radius 5
  move 0 0 10
    ball radius 5
```

results in a shape somewhat resembling a bacillus. Shape intersections and differences are expressed the same manner, replacing the `union` keyword with `intersect` or `diff`, e.g.,

```
intersect
  move -2.5 0 0
    ball radius 5
  move 2.5 0 0
    ball radius 5
```

or

```
diff
  move -2.5 0 0
    ball radius 5
  move 2.5 0 0
    ball radius 5
```

2.7.2 Regions

The simulation compartment comprises any number of named *regions*, each with a shape specified using CSG. For example,

```
region World
  box 1 1 1
```

defines a cubic compartment named *World*.

For simulation, the compartment is discretized into cubic subvolumes of user-specified size. For example,

```
subvolume edge 1
```

would make the subvolume dimensions $1 \times 1 \times 1$ units, resulting in one subvolume for the $1 \times 1 \times 1$ box above (i.e., equivalent to a spatially homogeneous simulation), while

```
subvolume edge 0.1
```

would divide the box into 1000 subvolumes.

Initial species, reactions, and reports can be limited to specific regions (see Section 2.3, Section 2.4, Section 2.5, and Section 2.6), while each species created by a reaction is limited to the region where the relevant reaction took place and the regions the species can diffuse to.

2.7.3 Diffusion

For compounds to diffuse among the subvolumes, we must specify a diffusion rate:

```
diffusion pattern at constant in regions
```

As usual, the pattern can be empty, in which case the diffusion applies to all species; the regions component may also be empty, indicating that the diffusion applies within all regions and at every region boundary. For example,

```
diffusion at 0.1
```

specifies that all compounds should diffuse with propensity 0.1, while

```
diffusion TCR(groove#) at 0.01 in Cell, Cell<->Membrane
```

specifies that any *TCR* with an unbonded *groove* site should diffuse with propensity 0.01 within the region *Cell* and across the boundary between regions *Cell* and *Membrane* (but not *within* the membrane).

A sequence of *diffusion* statements may be used to specify different diffusion propensities for different compounds or in different regions, in which case the propensity comes from the lexically last diffusion statement matching a given species. For example,

```
diffusion at 0.01
diffusion MHC at 0.1
diffusion MHC(peptide="none") at 1.0 in Cell
```

sets the diffusion propensity for all species and across all regions to 0.01 by default, 0.1 for compounds containing *MHC*, and 1.0 for those carrying no peptide and in region *Cell*.

2.8 Named constants

To avoid recompiling the model for different molecule counts and reaction rates, numeric constants in the model may be replaced with variable names (which are words), which must then be supplied at simulation time; for example,

```
new TCR at num_TCR
rxn t:TCR(phosph_sites=0) at k_on -> t.phosph_sites = 1
diffusion TCR at k_diff_TCR
```

2.9 Comments

Comments start with `--` (two consecutive hyphens); everything following the `--` until the end of line is ignored. E.g.,

```
-- comments can span the entire line
new TCR at 10 -- or follow any statement
```

3 Simulating with SSC

Simulating models written in SSC consists of two steps: compiling the model, which expands the pattern-based description into all possible species and reactions, and running the resulting simulator executable, which actually carries out the simulation.

3.1 Compiling and debugging

Once the model has been written to a file (say, `model.rxn`), it's compiled by running

```
ssc model.rxn
```

which produces output resembling

```
reading: model.rxn...
expanding reactions...
expansion complete after 2240 steps: 135 compounds and 1120 reactions
simulator executable: model
```

and a simulator executable, in this case called `model`.

Increasing verbosity (with the `-v` flag, which can be repeated several times to gradually increase verbosity) is useful when the compilation seems to be taking forever (because, for example, the set of possible compounds is infinite, and should be bounded by `limit`), or for seeing what goes on during compilation. Perhaps the most useful debugging tool, however, is the `--save-expanded` flag, which saves the *expanded* reaction network to a given file; e.g.,

```
ssc --save-expanded=expanded.txt model.rxn
```

will save the expanded network to `expanded.txt`. Since the expanded network contains all possible species in the system and all possible reactions among them in a human-readable format, it allows the user to verify that the model as written represents the desired reaction network.

Finally, a description of available SSC command line flags may be obtained with the `--help` flag:

```
ssc --help
Usage:
  ssc [OPTIONS] INPUT-FILE
Options:
  -o FILE --output=FILE      write compiled simulator to FILE
  --no-codegen               don't generate simulation code
  --no-compile               don't compile generated simulation code
```

```

--fatal-warnings  make warnings fatal
-v               --verbose          chatter more
-q               --quiet           don't chatter at all
--version        show version and exit
-h               --help            show usage info

```

3.2 Simulating

The easiest (and fastest) way to run the simulator is to specify the `-e` flag followed by simulation end time (in seconds). When the simulation finishes, it will output the final time together with the counts recorded by the various record statements, separated by TAB characters:

```

./model -e 100
time    # TCR(p1="on", p2="on")
100.000044      7978

```

The simulation may finish before the specified end time if no more reactions are possible; this generally does not happen in spatially resolved simulations because, although all reactions may have run out of reactants, diffusion can still take place.

We can also produce a trajectory sampled at regular intervals by adding the `-t` flag:

```

./model -t 10 -e 100
time    # TCR(p1="on", p2="on")
10.000113      1770
20.000569      3697
30.000014      5370
40.000026      6521
50.000630      7180
60.000108      7571
70.000109      7741
80.000205      7801
90.000810      7792
100.000677     7896

```

When the `-e` flag is omitted, the simulation will run forever (or until no more reactions can occur).

When some constants (reaction rates or counts) were specified as variables in the model file, the simulator must be provided with a configuration file containing the variable values with the `-c` flag, e.g.,

```
./model -e 100 -c model.cfg
```

The configuration file syntax is a list of pairs of the form

```
variable = number
```

and, optionally, comments starting with `#`; e.g.,

```

k_on = 1000
k_off = 0.008
# TCR bond formation and breaking
k_form_tcr = 0.00025
k_break_tcr_ag = 0.019 # TCR bound to cognate peptide

```

A full list of flags that may be passed to the simulator SSC command line options may be obtained with the `-h` flag:

```

./model -h
Options:
-c FILE    read variable values from FILE
-e T       end simulation after time T
-s T       report trajectory mean and std. dev from time T until the end
-d T       report amount probability distribution from time T until the end

```

```
-t T      sample trajectory at interval T (0 = at every step)
          (default: report at the end of simulation only)
-r N      initialize random number generator with seed n
-R        report random seed used
-v        show version information
-h        show this help message
```

4 Caveats

4.1 Patterns and reaction

Because reactions in SSC are specified in terms of patterns, they must be instantiated for all possible combinations of matching species during compilation. This can at times conflict with intuition; for example, some compounds may match patterns in multiple ways, resulting in multiple reactions for a set of species. Consider the hypothetical reaction

```
left:A(p="off", bond#1) right:A(bond#1, p="off") at k -> left.p = "on"
```

When this reaction is applied to the compound

```
A(p="off", site1#1) A(site1#1, p="off", site2#2) B(site#2)
```

it can yield two chemically distinguishable complexes, depending on which *A* molecule is matched as *left* and altered: either

```
A(p="on", site1#1) A(site1#1, p="off", site2#2) B(site#2)
```

or

```
A(p="off", site1#1) A(site1#1, p="on", site2#2) B(site#2)
```

When the same reaction is applied to a symmetric compound like

```
A(p="off", site1#1) A(site1#1, p="off")
```

it can also create two species (since *left* can still be assigned to either *A*), although the two are chemically indistinguishable. In either case, two reactions will be instantiated, effectively doubling the reaction constant *k* in the latter case.

5 References

- [DL04] DL04 Vincent Danos and Cosimo Laneve (2004). Formal molecular biology. *Theoretical Computer Science*, 325:69–110.
- [F09] F09 James Faeder et al. (2009). Rule-based Modeling of Biochemical Systems with BioNet-Gen. *Methods in Molecular Biology*, I. V. Maly, , ed., Humana Press.
- [G77] G77 Daniel Gillespie (1977). Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81:403–434.
- [LP06] LP06 Hong Li and Linda Petzold (2006). Logarithmic Direct Method for discrete stochastic simulation of chemically reacting systems. Technical report, University of California Santa Barbara, Computational Science and Engineering Group.
- [W06] W06 Dennis Wylie et al. (2006). A hybrid deterministic-stochastic algorithm for modeling cell signaling dynamics in a spatially inhomogeneous environments and under the influence of external fields. *Journal of Physical Chemistry B*, 110:12749–12765.