BioNetGen Tutorial

From BioNetWiki

BioNetGen2 is a model description language for constructing and simulating rule-based models of biochemical systems. Building a model in BioNetGen2 (BNG2) can be done in either of two ways: 1) Writing a text file (a BioNetGen Language (BNGL) file) that specifies the elements of a model and simulation of the model. 2) Using RuleBuilder, a graphical interface that constructs BNGL files using user-drawn pictograms. This tutorial describes how to construct and simulate models using the BNGL language.

Contents

Structure of the Input File

BNGL files contain the data required to specify models and may also contain commands that generate and perform other operations on the model. First, I describe the components of the model specification.

The model specification consists of four blocks, each beginning with a line containing a *begin* <blockname> command and ending with a line containing an *end* <blockname> command. The five basic block names are parameters, molecule types, seed species, reaction rules and observables. They may appear in any order, although, because of the dependencies the above order is the most logical.

`parameters`

Parameters are used to provide numerical values for variables that may appear in the species and reaction rules blocks. Parameters are generally used to specify initial concentrations of species and rate constants for reaction rules. A simple parameter block looks like

```
begin parameters
 1 R0    1
 2 kp1  0.5
 3 km1  0.1
 4 kp2  1e-3
 5 km2  0.1
 6 p1  10
 7 d1   5
 8 kpA  1-e4
 9 kmA  0.02
end parameters
```

The number in the first column is an index, which is provided solely for the convenience of the user. Parameters are referred to both in input and output by name and the index is ignored by the program. Specification of an index is optional, e.g. the first parameter line could read

```
  R0 1
```

Parameter names may contain only alphanumeric characters and underscores (_). After the parameter name is a string that gives the numerical value of the parameter. Integer, decimal, and exponential notation are allowed.

molecule types

Molecules are structured objects comprised of components that can have states and can bind to each other, both within a molecule and between molecules. The molecule types block defines the molecules that are contained in the model, the components they contain, and the allowed states of the components. The purpose of declaring molecule types is to allow error checking of the remaining blocks. Each molecule type declaration begins with the name of the molecule followed by a list of components in parentheses. The tilde character ('~') precedes the allowed state values. Here is a typical example that illustrates the syntax:

```
begin molecule types
 1  L(r)
 2  R(l,d,Y~U~P)
 3  A(SH2)
end molecule types
```

As with the parameters block, the index preceding each molecule declaration is optional. These declarations indicate that the molecules L and A each consist of a single component, r and SH2, respectively, that do not have an associated state attribute. The R molecule is comprised of three components, two of which (l and d) are not allowed to have state labels. The component Y has two allowed state values, U and P. In the examples that follow we will refer to these molecules both by their names and what they represent. Here, L represents a ligand molecule with a single binding site, r, for a receptor molecule, R. R has a ligand binding site, l, a dimerization domain, d, and a phosphorylation site, Y, which has two allowed values, U (unphosphorylated) and P (phosphorylated).

For any molecule that we want to be able to bind another molecule we must define at least one binding component. A component that appears in a molecule type declaration without a state label may be used only for binding and may not take on a state label in subsequent occurrences of the same molecule. Note that the namespaces for components of different molecules are separated, so it is permissible in this example to have a molecule B(SH2), or even B(SH2~U). It is also allowed to have molecules with no components, which can be used as dummy species or counters. For example, a species could be declared as

```
E1() 1
```

Note that the component block enclosed in parenthesis must be present even if no components are present.

seed species

Seed species are the molecules and complexes that are present at the start of network generation. Each declaration of a seed species is followed by a number or parameter that gives its initial concentration. When a molecule is referenced in the species block, all of its components must be referenced and those components that can be in different states must have a defined state. Here is an example of a seed species declaration referring to the molecule R defined in the molecule types block above

```
R(l,d,Y~U)  R0
```

One important caveat is that spaces may not appear anywhere inside the molecule string. Thus, the declaration

```
R(l, d,Y~U) R0
```

produces an error that would cause the program to stop. In general, spaces may also not appear in any species or pattern (see next subsection for a definition of patterns). R0 in the second column of the species declaration indicates that initial concentration of the declared species is determined by the parameter R0. Referring to an undefined parameter here will cause an error. The initial concentration can also be set by using a numerical value in the second column, although it is better to use a parameter because this facilitates subsequent analysis of the model.

Let's expand the species declaration to include L and A:

```
begin species
 1  L(r)       L0
 2  R(l,d,Y~U) R0
 3  A(SH2)     A0
end species
```

As with all other blocks in the BNGL file, the index here is optional and ignored when the file is processed. However, BNG2 does assign an index to each species (based on the order of appearance or generation), and this index is written to the species block in the output NET file created using the generate_network command (see below). These indices are used to refer to species in the reactions and groups blocks of the NET file.

Only molecules declared in the molecule types block may appear in the seed species block, and these must conform to the molecule types definitions. For example the species

```
R(l,d,Y~P)
```

follows the definition of the molecule R given above, but

```
R(l,d1,d2,Y~P)
```

does not because it refers to two components, d1 and d2, that are not in the molecule type declaration.

Species may also be comprised of complexes of two or more molecules, connected by bonds between molecular components. Bonds may join components within the same molecule or between different molecules. Bonds are declared as links between components indicated by an '!' character followed by a bond name, usually an integer. Each bond name must occur exactly two times within a species to specify the pair of components that are connected by the bond. The '.' character is used to concatenate molecules within a complex. For example, a ligand-receptor complex would be written as

```
L(r!1).R(l!1,d,Y~U)
```

with the link labeled 1 indicating a bond between component r of molecule L and component l of molecule R. A larger complex containing a dimer of ligand-bound receptors would be declared as

```
L(r!1).R(l!1,d!3,Y~U).L(r!2).R(l!2,d!3,Y~U)
```

Bond 1 joins the first L molecule to the first R molecule, Bond 2 joins the second L to the second R, and Bond 3 joins the two R's. Note that the bond labels are used only to indicate the connectivity and are interchangeable. Thus, the ligand-receptor complex

```
L(r!2).R(l!2,d,Y~U)
```

is identical to and interchangeable with the first complex:

```
L(r!1).R(l!1,d,Y~U).
```

Edge (bond) labels can be concatenated with state labels to allow bonds involving components that have state labels. They may also be concatenated with other edge labels to allow multiple bonds to the same components, although this usage is deprecated because it can often lead to confusion in constructing the reaction rules. An example of a component with both state and edge labels is the receptor-adaptor complex involving a bond between the receptor phoshpotyrosine and the SH2 domain of the adaptor:

```
R(l,d,Y~P!1).A(SH2!1)
```

Generally speaking, it is not necessary to define complexes that appear transiently prior to network generation, because they will be generated automatically by application of the rules. On the other hand, it may be desirable to use complexes to represent multi-subunit proteins that are constitutively associated. For example, a protein consisting of an alpha and a beta subunit could be represented as

```
alpha(Y1076~U,b!1).beta(Y1055~U,a!1).
```

If no reaction rule is specified to dissociate this complex, this complex will be indivisible.

observables

This block is used to define sums over the concentrations of species sharing similar attributes, which correspond to the quantities that are measured in typical biological experiments. Two standard examples are the total observed phosphorylation of a protein and the amount of a protein that is co-complexed with another protein. For the molecules defined above, examples of these types are

```
begin observables
 Molecules R_phos R(Y~P!?)
 Molecules A_R    A(SH2!1).R(Y~P!1)
end observables
```

The first column of an observable declaration indicates the type of observable. Currently, two types are allowed, Molecules and Species. Molecules is the more common type and indicates a weighted sum over the species selected by the pattern(s) defining a group, with the weight given by the number of matches found for each species. For example, a dimer containing two phosphorylated receptors R would have a weight of 2 in the observable R_phos. On the other hand, Species would count the dimer once. As you may have guessed, the second column specifies the name of the observable, while the remaining entries (separated by spaces) are patterns (see below) that select species contributing to the observable.

Patterns are similar to Species in that they are comprised of one or more molecules and may contain components, component state labels, and edges. Unlike Species, however, they do not have to be fully specified. For example, Components of defined molecules may be missing, state labels of multi-state components may be absent indicating that they can take on any value, edges names may be given with wildcards to select various connectivity, and the molecules in a complex need not be connected. It is this incompleteness of specification that allows Patterns to select a range of species and thus makes them so useful both in defining observables and reaction rules.

Consider the first observable above, R_phos. Because the components l and d of the molecule R are omitted from the pattern, the binding state of these components does not affect the match of the pattern onto a given species. The only component of R that affects the match is Y, which must be in the P state. The edge label '?', a wildcard, has the special meaning that a bond may or may not be present, analogous to the meaning of the '?' character in regular expressions. Examples of species that are matched by this pattern are

```
 R(l,d,Y~P)
 L(r!1).R(l!1,d,Y~P)
 R(l,d,Y~P!1).A(SH2!1)
 R(l,d!1,Y~P).R(l,d!1,Y~P)
```

The last species on the list would have two matches, because the pattern appears in both the first and second instance of R. Thus, the concentration of this species would be weighted by a factor of 2, as appropriate for the observable corresponding to the total phosphorylation of R.

The species (and weights) selected by an observable are output in the groups block of the NET file created by generate_network (see below). The groups block has the format

```
begin groups
 1 R_phos 2,3,5,2*6
end groups
```

where the integers in the third column refer to the indices of the selected species. Weights are indicated by multipliers in front of an index, e.g., the species with index 6 has a weight of 2.

The values of observables over a timecourse generated by the simulate_{ode,ssa} commands are written to a GDAT file (see below). The header at the top of the file lists the variable written to each column.

reaction rules

Reaction rules are the generators of species and reactions in a BioNetGen model. Each rule has four basic elements: reactant patterns, an arrow, product patterns, and a rate law. Reactant patterns are used to select a set of reactant species to which the rule will be applied. The arrow indicates whether the rule is applied in the forward direction only or in the forward and reverse directions. The product patterns define how the selected species are transformed by the rule and also act as the species selectors when the rule is applied in reverse (if applicable). Rules may add or delete molecules and edges and may change component state labels. Components may not be added or deleted. Thus, if a component in a molecule appears on the LHS of a rule, it must also appear on the RHS, unless the molecule is destroyed in the reaction. The default rate law is the elementary rate law, which gives the rate of a reaction as the product of a constant factor (the rate constant) and the reactant concentrations. Limited support for more complex rate laws, such as Michaelis-Menton, is also provided.

Let us consider a simple example of a reaction rule for the reversible binding of ligand to receptor. Using the molecules we defined above, we can write

```
begin reaction rules
 1  L(r) + R(l,d) <-> L(r!1).R(l!1,d) kp1, km1
end reaction rules
```

The reactant pattern L(r) selects ligand molecules that have an unbound r component. Since L molecules have only one component, the only species that is selected by this pattern is L(r). When multiple reactant or product patterns appear in a rule, they may be separated by the plus character ('+'), which is used to specify molecularity. (The molecularity of a reaction is one plus the number of plus signs on the reactant side.) The R(l,d) pattern selects R molecules with unbound l and d components regardless of the phosphorylation state of the Y component. It would, for example select the species R(l,d,Y~U) and R(l,d,Y~P). By specifying the component d and not indicating any binding state, we are requiring that the d component be unbound. We could have written the rule to be independent of the state of d by simply omitting it from R, which would allow the ligand to associate with R molecules that have formed dimers. The general principle is that reaction rules should only include molecules, components, states, and edges that are either required for the reaction to occur or are modified by the reaction. The bidirectional arrow indicates that the rule is to be applied in both the forward and reverse directions. A unidirectional (forward only) reaction is specified by the '->' arrow. The rate laws appear after the product patterns. For reversible reactions two rate laws are required, separated by commas. Here, each rate law is given by a single parameter, indicating that elementary rate laws are to be used. Additional arguments may appear after the rate constants to modify the behavior of the rule, but these will be discussed in a future edition of this tutorial.

Other rate laws

Other rate laws can be invoked using one of the keywords for the allowed rate law types followed by a list of parameters in the parenthesis. The two rate law types covered here are Sat and MM.

The reaction

```
S + E -> P + E Sat(kcat,Km)
```

will have the rate law, rate= kcat*[S]*[E]/(Km + [S]). Note that the term in the denominator must appear first in the reaction rule.

Also, note that the second species (in this case 'E') is optional on the left hand side of the reaction, such that the reaction

```
E -> E + P Sat(kcat,Km)
```

will have the rate law, rate= kcat*[E]/(Km + [E]).

There is also a MM rate law type that gives the rate of the reaction corrected for the amount of S bound in the ES complex (solves quadratic formula for [S]free). This will give a closer approximation to the kinetics of the two elementary processes. The reaction

```
S + E -> P + E MM(kcat,Km)
```

will have the rate law, rate= kcat*[S]_free*[E]/(Km + [S]_free), where [S]_free is determined by

```
[S]_free= 0.5*(([S]-Km-[E]) + ( ([S]-Km-[E])^2 + 4*Km*[S])^(1/2) )
```

The transformation in the L+R rule introduced in the section above results in two separate species, one matching the first pattern and one matching the second, being combined into a single species joined by a new bond between the matched molecule L component r and the matched molecule R component l. The component d is not modified, but it must be in an unbound state for the reaction to occur. The reverse transformation involves the breaking of a bond between L and R to create two separate species. Here are some examples of specific reactions generated by this rule:

```
L(r) + R(l,d,Y~U) -> L(r!1).R(l!1,d,Y~U) kp1
L(r!1).R(l!1,d,Y~U) -> L(r) + R(l,d,Y~U) km1
L(r) + R(l,d,Y~P) -> L(r!1).R(l!1,d,Y~P) kp1
L(r) + R(l,d,Y~P!1).A(SH2!1) -> L(r!2).R(l!2,d,Y~P!1).A(SH2!1) kp1
```

Note that the '+' operator in the rule above has a restrictive effect: if the breaking of the bond between L and R does not produce two separate species (not joined by any bond), then the rule is not applied and no reaction is generated. This is a general property of rules: the number of product species must equal the number of product patterns of the rule. Thus, intramolecular and intermolecular dissociation must be handled as separate rules. For example, suppose that A and B are joined by a direct bond in a complex, but may also be joined indirectly by unspecified molecules. In the latter case, breaking of the direct bond between A and B would not break up the complex containing both. If the only rule given for the dissociation of the direct bond were

```
 A(b!1).B(a!1) -> A(b!1) + B(a!1) kmAB
```

the bond between A and B would not break inside a complex containing an indirect link. To allow this bond to break, we would need the additional rule

```
A(b!1).B(a!1) -> A(b).B(a) kmAB
```

for intramolecular bond dissociation. In the future, we might provide syntax allowing both types of bond breakage to be specified in a single rule.

Let us now consider a rule for receptor dimerization that introduces some new syntactic features:

```
R(l!+,d) + R(l!+,d) <-> R(l!+,d!2).R(l!+,d!2) kp2, km2
```

The rule specifies that two species each containing a receptor with a bound l component and an unbound d component may be joined by a bond connecting the unbound d components. The fact the l must be bound is specified by the edge label !+, where the + character must match one or more bonds, analogous to its meaning within a regular expression. Because this rule is symmetric on both the reactant and product sides (with respect to interchange of the R molecule patterns), each set of possible reactants will have two identical matches causing each reaction to be generated twice. BioNetGen automatically detects this symmetry and generates the resulting reactions with the correct multiplicity. This is explained in more detail below

NB: the rate constants associated with rules are assumed to be single-site rate constants (statistical factors are added automatically as needed to account for multiple sites); furthermore, the rate constant associated with a bimolecular reaction is assumed to be of the A+B type (a factor of 0.5 is added automatically if the two reactant species matched by the rule are identical).

As a final example we consider two rules that change the phosphorylation state of the receptor:

```
R(d!+,Y~U) -> R(d!+,Y~P) p1
R(Y~P) -> R(Y~U) d1
```

The first rule states that the phosphorylation reaction occurs only when the receptor dimerization domain (d) is bound, i.e. phosphorylation occurs within a dimer. The second rule allows dephosphorylation to take place regardless of the state of the dimerization domain (or any other component of R) provided that the Y component is unbound. Similarly, the first rule also requires that Y is unbound when it is phosphorylated, presumably because the catalytic domain of the kinase or phosphatase must be able to access the tyrosine residue. Let's consider application of the first rule to the complex L(r!1).R(l!1,d!2,Y~U).L(r!1).R(l!1,d!2,Y~U), a dimer of two ligand-bound unphosphorylated receptors. The reactant pattern R(d!+,Y~U) has two instances in this complex because it matches either the first or second receptor. Thus, the rule will be applied twice to this species, each generating an instance of the reaction

```
L(r!1).R(l!1,d!2,Y~U).L(r!1).R(l!1,d!2,Y~U) -> \
L(r!1).R(l!1,d!2,Y~U).L(r!1).R(l!1,d!2,Y~P) p1
```

Note that the '\character at the end of the line can be used to continue a single line, which is convenient because all block entries must be contained on a single line in both BNGL and NET files. The length of a line is unimportant however. If the species index of the reactant and product are 20 and 21 respectively, this reaction (index 43) would appear in the NET file specifying the reaction network as

```
43 20 21 2*p1
```

indicating that reaction 43 is the unimolecular reaction transforming species 20 into species 21 with rate 2*p1*[S20]. The square brackets indicate concentration of species 20. The multiplicative factor of 2 arises because there are two ways this transformation can occur, and unlike the symmetric case above, these paths correspond to distinct physical mechanisms.

Generating and simulating the network

The blocks we have discussed above specify the elements of a BNG2 model. Statements in the BNGL file not enclosed within blocks are interpreted as commands by the BNG command interpreter. These should occur after the model specification blocks. At present, the command set is fairly limited but provides access to a number of different options for applying the rules to generate a network and for simulating a network. The major commands,

```
generate_network
simulate_ode
simulate_ssa
writeSBML
```

are discussed in this section.

When the BNGL file is parsed, the model specification is loaded into a data structure called BNGModel. The commands operate on this structure, sending output both to the BNGModel and to various files, as shown in Fig. 1.

Figure 1. Input and output files used by BioNetGen2.

```
-------------------------------------------------------------------------------
-----------
|BNGL file|  -> BNG2.pl -> generate_network   -> NET file (contains generated
-----------                                               species, reactions,
                                                          and observables)
                        -> simulate_{ode,ssa} -> CDAT file (concentrations of
                                                             all species)
                                              -> GDAT file (concentrations of
                                                             observables)
                        -> writeSBML          -> XML file (contains parameters,
                                                  species, reactions, and
                                                  observables in SBML level 2
                                                  format)
-------------------------------------------------------------------------------
```

The basic syntax for all commands is the same

```
command_name({param1=>value1,param2=>value2,hash1=>{hash_param1_param1=>hash_value1,...},...});
```

Command parameters are passed using Perl hashes so that parameter values can be specified in any order.

`generate_network`

The generate_network command generates a complete or partial network of species, reactions, and observables through iterative application of the rules to the initially defined species. For each iteration, the entire set of rules is applied to all of the current species, potentially generating new reactions and species. New species generated at the current iteration are not added to the species list until all of the rules have been applied. The order in which the rules are specified in the input file therefore does not affect the species and reactions generated at each iteration, although it will affect the order in which they appear in the species and reaction lists. The parameters that affect the behavior of the generate_network command are given in the Table below.

Table 1. Parameters for the generate_network command

Name	Function	Default value
check_iso	Perform isomorphism check for species that generate identical strings (keep this on unless you know what you're doing!)	1 (On)
max_agg	Max. number of molecules in one species	1e99
max_iter	Max. number of rule applications	100
max_stoich	Sets limit for number of molecules of each specified type in one species (hash- see syntax above)	Unset
print_iter	Print NET file after each iteration	0 (Off)
prefix	Base name of NET file	Base name of BNGL file
overwrite	Overwrite exisiting NET file	0 (Off)
verbose	Additional output for debugging	0 (Off)

Calling generate_network with the default parameters (no parameters specified) will cause network generation to proceed until the set of species and reactions no longer increases with further application of the rules. Some rules sets generate an infinite number of species and reactions, so in practice the maximum number of iterations is set to a finite value.

Once network generation terminates, the resulting set of species and reactions is written to the file PREFIX.net, where PREFIX can be user-specified and defaults to the base name of the BNGL file. This NET file is used by the simulation back end, accessed by the simulate_ode and simulate_ssa commands, to simulate the time evolution of the species concentrations. The syntax of the NET file is described here.

The file example1.bngl, provided in the Tutorial directory of the BNG2 distribution, contains a simple model based on the discussion above and illustrates the use of several commands with alternate parameters settings.

simulate_ode

The simulate_ode command is used to compute a timecourse of the network using ordinary differential equations to represent the average concentration of each species. The initial concentrations for species defined in the Species block are set to the declared values, while the concentrations of all species generated by generate_network are set to zero. simulate_ode calls the program Network, which provides an interface to the general-purpose ODE-solver CVODE. Adaptive implicit multi-step methods are used in the propagation to ensure efficient and accurate solution of the ODE's, which are usually stiff for networks with more than a few species. For very large systems (more than a few hundred species), then the sparse=>1 option is recommended. (We find that most systems are sparse.) With this option, CVODE uses sparse matrix methods to avoid storage of the n_species x n_species Jacobian matrix, which becomes prohibitive for systems with more than about 10^4 species, and the iterative GMRES algorithm to solve the required systems of linear equations. This enables networks with 10^3-10^4+ species to be simulated in a few minutes (or less) on standard processors, even with stiffness.

Note that the parameter sample_times, which sets the times at which the concentrations are to be sampled, is an example of an array-valued parameter. Its argument is a comma-separated list of numbers enclosed by square brackets. The command

```
simulate_ode({sample_times=>[1,10,100]});
```

would cause the species concentrations and observable values to be printed at times of 0, 1, 10, and 100.

Table 2. Parameters for the simulate_ode command. * indicates required parameters.

Name	Function	Default value
atol	Absolute error tolerance for species concentrations	1e-8
n_steps	Number of intervals at which to report concentrations	1
netfile	Name of a network file to use for simulation	None
prefix	Base name of CDAT and GDAT files	Base name of BNGL file
rtol	Relative error tolerance for species concentrations	1e-8
sample_times	Times at which concentrations are reported (supercedes requirment for t_end)	None
sparse	Turns on use of sparse matrix formation of the Jacobian and iterative solution of linear equations using GMRES. Recommended for networks with more than a few hundred species	0 (Off)
steady_state	Setting this to non-zero turns on check for equilibration of species concentrations. Time course will stop when steady state is reached. If it is not reached in the allotted time the simulation will stop with an error.	0 (Off)
t_start	Starting time for integration.	0
t_end*	End time for integration	None
verbose	Additional output for debugging	0 (Off)

simulate_ssa

The simulate_ssa command is used to compute a timecourse of the network using the Gillespie direct algorithm for simulation of the stochastic master equations. When using this method to simulate a network, the species concentrations must be provided in number of molecules (per cell or per fraction of a cell) and the rate constants must be scaled accordingly. BNG2 does not currently provide any functions for performing unit conversions. The simulate_ssa interface is still incomplete and additional functionality will be added as needed. Currently, only one simulation run can be performed per command invocation.

The main difference between the simulate_ssa command and standard Gillespie implementations, is that species and reactions can be generated by application of the reation rules on-the-fly. No special parameters are required to access this functionality. This is done by keeeping track of whether the reaction rules have been applied to each species. When a species to which the rules have not been applied becomes populated for the first time, the rules are applied to that species to generate new reactions and species and to update observables. This adaptive generation of the network is particularly useful for infinite networks. A simulation can be carried out adaptively by setting a small value for max_iter in the generate_network command prior to the simulate_ssa command, say 1. The stochastic simulation will then map out the network over the course of the simulation.

Table 3. Parameters for the simulate_ssa command. * indicates required parameters.

Name	Function	Default value
n_steps	Number of intervals at which to report concentrations	1
prefix	Base name of CDAT and GDAT files	Base name of BNGL file
sample_times	Times at which concentrations are reported (supercedes requirment for t_end)	None
t_start	Starting time for integration.	0
t_end*	End time for integration	None
verbose	Additional output for debugging	0 (Off)

writeSBML

This command is a simple utility to write a network generated by BNG2 to an XML-encoded file adhering to the the System Biology Markup Language (SBML) Level 2 specification. SBML files may be imported to a large number of different applications that provide a wide range of simulation and analysis tools (http://sbml.org/).

Table 4. Parameters for the writeSBML command.

Name	Function	Default value
prefix	Base name of CDAT and GDAT files	Base name of BNGL file
verbose	Additional output for debugging	0 (Off)

Other BNG Commands

setParameter

The setParameter command sets a parameter value. The syntax is as follows: setParameter("_parameter_name_", "_parameter_value_").

setConcentration

The setConcentration command sets the concentration of a particular species. The syntax is as follows: setConcentration("_species_name_", "_concentration_value_").

writeNET

If for some reason BNG is not correctly writing the network to a file (the .net file), this can be forced with the writeNET command. The syntax is as follows: writeNET({prefix=>_filename_}), where _filename_ does not include the .net extension. Quotation marks are not needed around the file name.

readFile

The readFile command can be used to load a previously generated network file. The syntax is as follows: readFile({file=>"filename.net"}). Click here for an example of usage.

Pitfalls, Features, and Bugs

Symmetric Reaction Rules

This is a confusing subject, so let's consider a simple example.

```
A(a) + A(a) -> A(a!1).A(a!1) kd
```

If A has a second domain b~U~P, then BNG will generate the following reactions

```
A(a,b~U) + A(a,b~U) -> ... 0.5*kd
A(a,b~U) + A(a,b~P) -> ... kd
A(a,b~P) + A(a,b~P) -> ... 0.5*kd
```

Only the second reaction agrees with what you think it should be. The extra factor of 0.5 is coming from the reaction kinetics, because the two reactants are the same, rather than multiplicities. Consider two reactions A + B -> ... and A + A -> ... . If N_A= N_B, the number of collsions per second for the first reaction is proportional to N_A*N_B and for the second reaction is proportional to N_A*(N_A-1)/2. The factor of two that BNG is adding is due to the factor of two in the denominator. We decided to put this in the net file explicitly, because we assumed that most programs would not add the extra factor of two when computing the rates of symmetric reactions. If the factor of two is not added, then the total rate of reactions generated by the rule is not correct. In the example above, the rates of the reactions (without correction) would be

```
A(a,b~U) + A(a,b~U) -> ... kd
A(a,b~U) + A(a,b~P) -> ... kd
A(a,b~P) + A(a,b~P) -> ... kd
```

which neglects the fact that the symmetric reaction should occur at half the rate of the asymmetric one.

References

M. L. Blinov et al. (2005) "Graph theory for rule-based modeling of biochemical networks." BioCONCUR 2005, San Francisco, CA.

J. R. Faeder et al. (2005) "Rule-based modeling of biochemical networks." Complexity, 10, 22-41.