pysb Documentation

Release 0.1.0-264-geb53

Jeremy Muhlich

August 15, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION TO MODELING

The premise to all modeling is that, based on experimental observations, we believe that a set of rules gobvern the behavior of a system. Modeling the behavior of such a system involves the *elucidation* of the rules that govern the system to understand our observations and the *use* of such rules to further predict the behavior of a system under a range of conditions. Thus, models can be used to both *explain* or *predict* the behavior of a system given a set of conditions. The best models can perform both tasks satisfactorily. A simple example is the colloquial story of Newton's apple. The observation was that the apple fell on Newton's head. He derived the simple yet incredibly powerful $F = ma$ whereby he observed that the Force, F, applied to an object of mass, m, resulted in an acceleration, a. We now know that this model holds for most conditions in every-day activities but we know that it fails for e.g. relativistic effects. Therefore a model has a domain of application and a limited usefulness. However, a successful model can be employed accurately for both the explanation and the prediction of a system. In the case of cell-molecular biology, we aim to develop models that describe the behavior of cellular systems. The model can guide us to understand what are our gaps in the observations that prevent us from generalizing a theory and, when they capture the key significant aspects of the behavior of a system, predict the outcome of the behavrio of a system under a given set of conditions.

1.1 PySB as modeling tool

PySB is a set of software tools that enables users to develop, implement, and execute biological models in the Python programming environment. One of the main advantages of PySB is that it leverages the power of a very powerful programming language to express biological concepts as parts of a program. The properties of the programming environment are therefore the same properties found in PySB. Python is an object-oriented programming language that provides a useful environment for programming techniques such as data abstraction, encapsulation, modularity, message-passing, polymorphism, and inheritance to name a few. In addition to these technical advantages, we chose Python due to its readable and clear syntax. In our view, one of the most difficult issues with current biological modeling is shareability and transparency, both of which are addresssed, at least in part, by a clear syntax and a programmatic flow of ideas. PySB can handle simple models, modular models, and multiple instances of models, as shown in the tutorial. We invite users to contribute and share their innovations and ideas to make PySB a better open-source tool for the programming community.

1.2 A quick example

Using and running PySB can be as simple as typing the following commands in your Python shell. Go to the directory containing the file simplemodel.py (usually pysb/examples) and try this at your shell!:

```
[host] > python earm_figures.py
```

You will see some feedback from the machine, depending on your operating system (and assuming PySB is correctly installed). After a few seconds of calculations you should get two figures. The first figure shows the experimental

death time determined form experiments (as dots with error bars) followed by the model-predicted average (solid line) and the standard deviation ranges (dashed lines). The second graph will show you the model signatures of three species, namely initiator caspase (IC) substrate, effector caspase (EC) substrate, and mitochondrial outer membrane permeabilization (MOMP) as indicated by release of Smac to the cytosol. You have now run a model! Feel free to open the files `earm_1_0.py` to see a simple model instantiation and `earm_figures.py` to see how the model is run and the figures are generated. If you want to learn how to build biological models in a systematic (and we think fun) way, visit our *Tutorial*.

CHAPTER

TWO

GETTING STARTED

2.1 Python Knowledge

For those unfamiliar with *Python* or programming there are several resources available online. We have found the ones below useful to learn *Python* in a practical and straightfoward manner.

Quick Python Overview (10 minutes or so):

- OpenOpt Python for the impatient
- Instant Python

Python for beginners, experienced users, or if you want a refresher:

- Official Python tutorial
- Python for non-programmers
- Dive into Python
- Thinking in Python
-

For experienced users of other languages:

- **NumPy for Matlab**
  - Also the Mathesaurus
  - Matlab commands in Numerical Python cheatsheet
- Scientific Python

2.2 Requirements

The following are what we consider the *necessary* to use PySB as a biological simulation tool. The versions listed are the ones that are known to work well with the material in this documentation. Later versions *should* work and earlier versions *might* work. Advanced users may want to replace these requirements as they see fit.

- Python 2.7: You will need a version of the Python interpreter in your machine.
- NumPy 1.7: You may not need NumPy for simple model building but you will want to have it for any sort of numerical manipulation of your model. The work presented here has been carried out using NumPy 1.7 or later.
- SymPy 0.7: Like NumPy, you may not need SymPy to carry out simple model building and instantiation but if you want to run numerical simulation,s SymPy will be a required tool for symbolic math manipulation.

- BioNetGen 2.1.8: The Biological Network Generator is a very useful tool for rules-based modeling. It is a very powerful and useful package for modeling and simulation of biological systems and provides a set of useful tools that could be used with PySB. As of now, PySB uses BioNetGen as a tool to generate the reaction connectivity network using its robust engine. If you want to generate biochemical representations of a biological system, you will need BioNetGen. BioNetGen depends on Perl 2, so you will need that as well.
- SciPy 0.10: Scientific Python provides a suite of extremely useful tools for scientific computing in the Python environment. For example, SciPy provides the LSODA integrator interface that we use in PySB.
- MatPlotLib 1.2 (PyLab): This package provides a very useful interface for generation, manipulation, export, etc of plots in two and three dimensions. If you want to visualize any type of plots you will need MatPlotLib.

2.3 Recommended

- iPython 0.13: Even though iPython is not a *requirement* it is **strongly** recommended. iPython provides a very nice and simple shell interface for the Python interpreter with such niceties as tab completion, object exploration, running and editing from the shell, debugging, and history to name a few. You want this.
- KaSim/Kappa : This wonderful rules-based package can be run natively from PySB to take advantage of its stochastic simulation capabilities and great visualization tools. It is a great complement to the modeling tools in BioNetGen.
- cookbooks

2.4 The easiest way to use PySB

Currently the easiest way to use PySB is through a virtual machine. We provide an `Ubuntu Linux ISO` image file based on `Ubuntu 12.04` to run *PySB*. With this you can get acquainted, build some simple models, and carry out simulations. If you decide you want to use PySB on your native computer platform we will be posting instructions for this in the near future.

To run *PySB* from the image you can download the image from The PySB website and burn it onto a bootable DVD or USB stick. Alternatively you can download a virtual machine software such as Virtual Box (FREE) or Parallels ($). Follow the instructions to setup the virtual machine booting from the `ISO` file to begin modeling with Python programs.

2.5 Instructions for Linux installations

coming soon

2.6 Instructions for OS X installations

coming soon

2.7 Instructions for Windows installations

coming soon

CHAPTER

THREE

TUTORIAL

This tutorial will walk you through the creation of your first PySB model. It will cover the basics, provide a guide through the different programming constructs and finally deal with more complex rule-building. Users will be able to write simple programs after finishing this section. In what follows we will assume you are issuing commands from a *Python* prompt (whether it be actual *Python* or a shell such as *iPython*. See *Getting Started* for details).

Note: Familiarity with rules-based biomodel encoding tools such as BioNetGen or Kappa would be useful to users unfamiliar with *Rules-based* approaches to modeling. A short *About Rules* is included for new users.

Note: Although a new user can go through the tutorial to understand how PySB works, a basic understanding of the Python programming language is essential. See the *Getting Started* section for some *Python* suggestions.

3.1 Modeling with PySB

A biological model in PySB will need the following components to generate a mathematical representation of a system:

- Model definitions: This instantiates the model object.
- Monomer definition: This instantiates the monomers that are allowed in the model.
- Parameters: These are the numerical parameters needed to create a mass-action or stochastic simulation.
- Rules: These are the set of statements that describe how *monomer species*, interact as prescribed by the parameters involved in a given rule. The collection of these rules is called the *model topology*.

The following examples will be taken from work in the Sorger lab in extrinsic apoptosis signaling. The initiator caspases, activated by an upstream signal, play an essential role activating the effector Bcl-2 proteins downstream. In this model, Bid is catalitically truncated and activated by Caspase-8, an initiator caspase. We will build a model that represents this activation as a two-step process as follows:

$$C8 + Bid \underset{kr}{\overset{kf}{}} C8 : Bid \quad \longleftarrow \text{Complex formation step}$$

$$C8 : Bid \overset{kc}{\rightarrow} C8 + tBid \quad \longleftarrow \text{Complex dissociation step}$$

Where tBid is the truncated Bid. The parameters `kf`, `kr`, and `kc` represent the forward, reverse, and catalytic rates that dictate the consumption of Bid via catalysis by C8 and the formation of tBid. We will eventually end up with a

mathematical representation that will look something like this:

$$\begin{aligned}
\frac{d[C8]}{dt} &= -kf[C8] * [Bid] + kr * [C8 : Bid] + kc * [C8 : Bid] \\
\frac{d[Bid]}{dt} &= -kf * [C8] * [Bid] + kr * [C8 : Bid] \\
\frac{d[C8 : Bid]}{dt} &= kf * [C8] * [Bid] - kr * [C8 : Bid] - kc * [C8 : Bid] \\
\frac{dt[tBid]}{dt} &= kc * [C8 : Bid]
\end{aligned} \tag{3.1}$$

The species names in square braces represent concentrations, usually give in molar (M) and time in seconds. These ordinary differential equations (ODEs) will then be integrated numerically to obtain the evolution of the system over time. We will explore how a model could be instantiated, modified, and expanded *without* having to resort to the tedious, repetitive, and error-prone writing and rewriting of equations as those listed above.

3.2 The Empty Model

We begin by creating a model, which we will call `mymodel`. Open your favorite Python code editor and create a file called `mymodel.py`. The first lines of a PySB program must contain these lines so you can type them or paste them in your editor as shown below. Comments in the *Python* language are denoted by a hash (`#`) in the first column.

```
# import the pysb module and all its methods and functions
from pysb import *

# instantiate a model
Model()
```

Now we have the simplest possible model – the empty model!

To verify that your model is valid and your PySB installation is working, run `mymodel.py` through the Python interpreter by typing the following command at your command prompt:

```
python mymodel.py
```

If all went well, you should not see any output. This is to be expected, because this PySB script *defines* a model but does not execute any contents. We will revisit these concepts once we have added some components to our model.

3.3 Monomers

Chemical *species* in PySB, whether they are small molecules, proteins, or representations of many molecules are all derived from *Monomers*. *Monomers* are the superunit that defines how a *species* can be defined and used. A *Monomer* is defined using the keyword `Monomer` followed by the desired *monomer* name and the *sites* relevant to that monomer. In PySB, like in BioNetGen or Kappa, there are two types of *sites*, namely bond-making/breaking sites and state sites. The former allow for the description of bonds between *species* while the latter allow for the assignment of *states* to species. Following the first lines of code entered into your model in the previous section we will add a *monomer* named 'Bid' with a bond site 'b' and a state site 'S'. The state site will contain two states, the untruncated (`u`) state and the truncated (`t`) state as shown:

```
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})
```

Note that this looks like a Python function call. This is because it *is* in fact a Python function call! [1] The first argument to the function is a string (ecnlosed in quotation marks) specifying the monomer's name and the second argument is a list of strings specifying the names of its sites. Note that a monomer does not need to have state sites. There is also a third, optional argument for specifying whether any of the sites are "state sites" and the list of valid states for those sites. We will introduce state sites later.

Let's define two monomers in our model, corresponding to Caspase-8, an initiator caspase involved in apoptosis (`C8`) and BH3-interacting domain death agonist (`Bid`) (ref?):

```
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'])
```

Note that although the C8 monomer only has one site 'b', you must still use the square brackets to indicate a *list* of binding sites. Anticipating what comes below, the `'S'` site will become a state site and hence, we choose to represent it in upper case but this is not mandatory.

Now our model file should look like this:

```
# import the pysb module and all its methods and functions
from pysb import *

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})
```

We can run `python mymodel.py` again and verify there are no errors, but you should still have not output given that we have not *done* anything with the monomers. Now we can do something with them.

Run the *ipython* (or *python*) interpreter with no arguments to enter interactive mode (be sure to do this from the same directory where you've saved `mymodel.py`) and run the following code:

```
>>> import mymodel as m
>>> m.model.monomers
```

You should see the following output:

```
Monomer(name='C8', sites=['b'], site_states={})
Monomer(name='Bid', sites=['b', 'S'], site_states={})
```

In the first line, we treat `mymodel.py` as a *module* [2] and import its symbol `model`. In the second and third lines, we loop over the `monomers` attribute of `model`, printing each element of that list. The output for each monomer is a more verbose, explicit representation of the same call we used to define it. [3]

Here we can start to see how PySB is different from other modeling tools. With other tools, text files are typically created with a certain syntax, then passed through an execution tool to perform a task and produce an output, whether on the screen or to an output file. In PySB on the other hand we write Python code defining our model in a regular Python module, and the elements we define in that module can be inspected and manipulated as Python objects interactively in one of the Python REPLs such as *iPython* or *Python*. We will explore this concept in more detail in the next section, but for now we will cover the other types components needed to create a working model.

[1] Technically speaking it's a constructor, not just any old function.

[2] *Python* allows users to write PySB code to a file. This file can be later used as an executable script or from an interactive instance. Such files are called *modules* and can be imported into a Python instance. See Python modules for details.

[3] The astute Python programmer will recognize this as the `repr` of the monomer object, using keyword arguments in the constructor call.

3.4 Parameters

A `Parameter` is a named constant floating point number used as a reaction rate constant, compartment volume or initial (boundary) condition for a species (*parameter* in BNG). A parameter is defined using the keyword `Parameter` followed by its name and value. Here is how you would define a parameter named 'kf1' with the value 4×10^{-7}:

```
Parameter('kf1', 4.0e-7)
```

The second argument may be any numeric expression, but best practice is to use a floating-point literal in scientific notation as shown in the example above. For our model we will need three parameters, one each for the forward, reverse, and catalytic reactions in our system. Go to your `mymodel.py` file and add the lines corresponding to the parameters so that your file looks like this:

```
# import the pysb module and all its methods and functions
from pysb import *

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})

# input the parameter values
Parameter('kf', 1.0e-07)
Parameter('kr', 1.0e-03)
Parameter('kc', 1.0)
```

Once this is done start the *ipython* (or *python*) intepreter and enter the following commands:

```
>>> import mymodel as m
>>> m.model.parameters
```

and you should get an output such as:

```
{'kf': Parameter(name='kf', value=1.0e-07),
 'kr': Parameter(name='kr', value=1.0e-03),
 'kc': Parameter(name='kc', value=1.0    )}
```

Your model now has monomers and parameters specified. In the next section we will specify rules, which specify the interaction between species and parameters.

Warning: PySB or the integrators that we suggest for use for numerical manipulation do not keep track of units for the user. As such, the user is responsible for keeping track of the model in units that make sense to the user! For example, the forward rates are typically in $M^{-1}s^{-1}$, the reverse rates in s^{-1}, and the catalytic rates in s^{-1}. For the present examples we have chosen to work in a volume size of $1.0pL$ corresponding to the volume of a cell and to specify the Parameters and Initial conditions in numbers of molecules per cell. If you wish to change the units you must change *all* the parameter values accordingly.

3.5 Rules

Rules, as described in this section, comprise the basic elements of procedural instructions that encode biochemical interactions. In its simplest form a rule is a chemical reaction that can be made general to a range of monomer states or very specific to only one kind of monomer in one kind of state. We follow the style for writing rules as described in BioNetGen but the style proposed by Kappa is quite similar with only some differences related to the implementation

details (e.g. mass-action vs. stochastic simulations, compartments or no compartments, etc). We will write two rules to represent the interaction between the reactants and the products in a two-step manner as described in the *About Rules* section.

The general pattern for a rule consists of the statement *Rule* and in parenthesis a series of statements separated by commas, namely the rule name (string), the rule interactions, and the rule parameters. The rule interactions make use of the following operators:

```
*+* operator to represent complexation
*<>* operator to represent backward/forward reaction
*>>* operator to represent forward-only reaction
*%* operator to represent a binding interaction between two species
```

To illustrate the use of the operators and the rule syntax we write the complex formation reaction with labels illustrating the parts of the rule:

```
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S='u') <> C8(b=1) % Bid(b=1, S='u'), *[kf, kr])
          |               |     |           |         |     |    |     |             |
          |               |     |           |         |     |    |     |            parameter list
          |               |     |           |         |     |    |     |
          |               |     |           |         |     |    |    bound species
          |               |     |           |         |     |    |
          |               |     |           |         |     |   binding operator
          |               |     |           |         |     |
          |               |     |           |         |    bound species
          |               |     |           |         |
          |               |     |           |        forward/backward operator
          |               |     |           |
          |               |     |          unbound species
          |               |     |
          |               |    complexation / addition operator
          |               |
          |              unbound species
        rule name
```

The *rule name* can be any string and should be enclosed in single (‘) or double (”) quotation marks. The species are *instances* of the mononmers in a specific state. In this case we are requiring that *C8* and *Bid* are both unbound, as we would not want any binding to occur with species that are previously bound. The *complexation* or *addition* operator tells the program that the two species are being added, that is, undergoing a transition, to form a new species as specified on the right side of the rule. The forward/backward operator states that the reaction is reversible. Finally the *binding* operator indicates that there is a bond formed between two or more species. This is indicated by the matching integer (in this case *1*) in the bonding site of both species along with the *binding* operator. If a non-reversible rule is desired, then the *forward-only* operator can be relplaced for the *forward/backward* operator.

In order to actually change the state of the Bid protein we must now edit the monomer so that have an acutal state site as follows:

```
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})
```

Having added the state site we can now further specify the state of the Bid protein whe it undergoes rule-based interactions and explicitly indicate the changes of the protein state.

With this state site added, we can now go ahead and write the rules that will account for the binding step and the unbinding step as follows:

```
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S='u') <>C8(b=1) % Bid(b=1, S='u'), kf, kr)
Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) % Bid(b=None, S='t'), kc)
```

As shown, the initial reactants, *C8* and *Bid* initially in the unbound state and, for Bid, in the ‘u’ state, undergo a complexation reaction and further a dissociation reaction to return the original *C8* protein and the *Bid* protein but now

in the 't' state, indicating its truncation. Make these additions to your `mymodel.py` file. After you are done, your file should look like this:

```
# import the pysb module and all its methods and functions
from pysb import *

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})

# input the parameter values
Parameter('kf', 1.0e-07)
Parameter('kr', 1.0e-03)
Parameter('kc', 1.0)

# now input the rules
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S=None) <> C8(b=1) % Bid(b=1, S=None), kf, kr)
Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) + Bid(b=None, S='t'), kc)
```

Once you are done editing your file, start your *ipython* (or *python*) interpreter and type the commands at the prompts below. Once you load your model you should be able to probe and check that you have the correct monomers, parameters, and rules. Your output should be very similar to the one presented (output shown below the '>>>' python prompts).:

```
>>> import mymodel as m
>>> m.model.monomers
   {'C8': Monomer(name='C8', sites=['b'], site_states={}),
   'Bid': Monomer(name='Bid', sites=['b', 'S'], site_states={'S': ['u', 't']})}
>>> model.parameters
   {'kf': Parameter(name='kf', value=1.0e-07),
    'kr': Parameter(name='kr', value=1.0e-03),
    'kc': Parameter(name='kc', value=1.0    )}
>>> m.model.rules
   {'C8_Bid_bind': Rule(name='C8_Bid_bind', reactants=C8(b=None) +
   Bid(b=None, S='u'), products=C8(b=1) % Bid(b=1, S='u'),
   rate_forward=Parameter(name='kf', value=1.0e-07),
   rate_reverse=Parameter(name='kr', value=1.0e-03)),
   'tBid_from_C8Bid': Rule(name='tBid_from_C8Bid', reactants=C8(b=1) %
   Bid(b=1, S='u'u), products=C8(b=None) + Bid(b=None, S=t),
   rate_forward=Parameter(name='kc', value=1.0))}
```

With this we are almost ready to run a simulation, all we need now is to specify the initial conditions of the system.

3.6 Initial conditions

Having specified the *monomers*, the *parameters* and the *rules* we have the basics of what is needed to generate a set of ODEs and run a model. From a mathematical perspective a system of ODEs can only be solved if a bound is placed on the ODEs for integration. In our case, these bounds are the initial conditions of the system that indicate how much non-zero initial species are present at time *t=0s* in the system. In our system, we only have two initial species, namely *C8* and *Bid* so we need to specify their initial concentrations. To do this we enter the following lines of code into the `mymodel.py` file:

```
Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)
Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)
```

A parameter object must be declared to specify the initial condition rather than just giving a value as shown above. Once the parameter object is declared (i.e. *C8_0* and *Bid_0*) it can be fed to the *Initial* definition. Now that we have specified the initial conditions we are basically ready to run simulations. We will add an *observables* call in the next section prior to running the simulation.

3.7 Observables

In our model we have two initial species (*C8* and *Bid*) and one output species (*tBid*). As shown in the (4.1) derived from the reactions above, there are four mathematical species needed to describe the evolution of the system (i.e. *C8*, *Bid*, *tBid*, and *C8:Bid*). Although this system is rather small, there are situations when we will have many more species than we care to monitor or characterize throughout the time evolution of the (4.1). In addition, it will often happen that the desirable species are combinations or sums of many other species. For this reason the rules-based engines we currently employ implemented the *Observables* call which automatically collects the necessary information and returns the desired species. In our case, we will monitor the amount of free *C8*, unbound *Bid*, and active *tBid*. To specify the observables enter the following lines in your `mymodel.py` file as follows:

```
Observable('obsC8', C8(b=None))
Observable('obsBid', Bid(b=None, S='u'))
Observable('obstBid', Bid(b=None, S='t'))
```

As shown,the observable can be a species. As we will show later the observable can also contain wild-cards and given the "don't care don't write" approach to rule-writing it can be a very powerful approach to observe activated complexes.

3.8 Simulation and analysis

By now your `mymodel.py` file should look something like this:

```
# import the pysb module and all its methods and functions
from pysb import *

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})

# input the parameter values
Parameter('kf', 1.0e-07)
Parameter('kr', 1.0e-03)
Parameter('kc', 1.0)

# now input the rules
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S=None) <> C8(b=1) % Bid(b=1, S=None), *[kf, kr])
Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) + Bid(b=None, S='t'), kc)

# initial conditions
```

```
Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)
Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)

# Observables
Observable('obsC8', C8(b=None))
Observable('obsBid', Bid(b=None, S='u'))
Observable('obstBid', Bid(b=None, S='t'))
```

You can use a few commands to check that your model is defined properly. Start your *ipython* (or *python*) interpreter and enter the commands as shown below. Notice the output should be similar to the one shown (output shown below the '>>>' ` prompts):

```
>>> import mymodel as m
>>> m.model.monomers
   {'C8': Monomer(name='C8', sites=['b'], site_states={}),
    'Bid': Monomer(name='Bid', sites=['b', 'S'], site_states={'S': ['u', 't']})}
>>> m.model.parameters
   {'kf': Parameter(name='kf', value=1.0e-07),
    'kr': Parameter(name='kr', value=1.0e-03),
    'kc': Parameter(name='kc', value=1.0    ),
    'C8_0': Parameter(name='C8_0', value=1000),
    'Bid_0': Parameter(name='Bid_0', value=10000)}
>>> m.model.observables
   {'obsC8': <pysb.core.Observable object at 0x104b2c4d0>,
    'obsBid': <pysb.core.Observable object at 0x104b2c5d0>,
    'obstBid': <pysb.core.Observable object at 0x104b2c6d0>}
>>> m.model.initial_conditions
   [(C8(b=None), Parameter(name='C8_0', value=1000)), (Bid(b=None, S=u), Parameter(name='Bid_0', valu
>>> m.model.rules
   {'C8_Bid_bind': Rule(name='C8_Bid_bind', reactants=C8(b=None) +
   Bid(b=None, S=None), products=C8(b=1) % Bid(b=1, S=None),
   rate_forward=Parameter(name='kf', value=1.0e-07),    rate_reverse=Parameter(name='kr', value=1.0e-
    'tBid_from_C8Bid': Rule(name='tBid_from_C8Bid', reactants=C8(b=1)
    % Bid(b=1, S=u), products=C8(b=None) + Bid(b=None, S=t),    rate_forward=Parameter(name='kc', val
```

With this we are now ready to run a simulation! The parameter values for the simulation were taken directly from typical values in the paper about extrinsic apoptosis signaling. To run the simulation we must use a numerical integrator. Common examples include LSODA, VODE, CVODE, Matlab's ode15s, etc. We will use two *python* modules that are very useful for numerical manipulation. We have adapted the integrators in the *SciPy*[#sp]_ module to function seamlessly with PySB for integration of ODE systems. We will also be using the *PyLab* [4] package for graphing and plotting from the command line.

We will begin our simulation by loading the model from the *ipython* (or *python*) interpreter as shown below:

```
>>> import mymodel as m
```

You can check that your model imported correctly by typing a few commands related to your model as shown:

```
>>> m.mymodel.monomers
>>> m.mymodel.rules
```

Both commands should return information about your model. (Hint: If you are using iPython, you can press tab twice after "m.mymodel" to tab complete and see all the possible options).

Now, we will import the *PyLab* and PySB integrator module. Enter the commands as shown below:

[4] PyLab: http://www.scipy.org/PyLab

```
>>> from pysb.integrate import odesolve
>>> import pylab as pl
```

We have now loaded the integration engine and the graph engine into the interpreter environment. You may get some feedback from the program as some functions can be compiled at runtime for speed, depending on your operating system.Next we need to tell the integrator the time domain over which we wish to integrate the equations. For our case we will use 20000s of simulation time. To do this we generate an array using the *linspace* function from *PyLab*. Enter the command below:

```
>>> t = pl.linspace(0, 20000)
```

This command assigns an array in the range $[0..20000]$ to the variable t. You can type the name of the variable at any time to see the content of the variable. Typing the variable t results in the following:

```
>>> t
array([     0.        ,    408.16326531,    816.32653061,   1224.48979592,
        1632.65306122,   2040.81632653,   2448.97959184,   2857.14285714,
        3265.30612245,   3673.46938776,   4081.63265306,   4489.79591837,
        4897.95918367,   5306.12244898,   5714.28571429,   6122.44897959,
        6530.6122449 ,   6938.7755102 ,   7346.93877551,   7755.10204082,
        8163.26530612,   8571.42857143,   8979.59183673,   9387.75510204,
        9795.91836735,  10204.08163265,  10612.24489796,  11020.40816327,
       11428.57142857,  11836.73469388,  12244.89795918,  12653.06122449,
       13061.2244898 ,  13469.3877551 ,  13877.55102041,  14285.71428571,
       14693.87755102,  15102.04081633,  15510.20408163,  15918.36734694,
       16326.53061224,  16734.69387755,  17142.85714286,  17551.02040816,
       17959.18367347,  18367.34693878,  18775.51020408,  19183.67346939,
       19591.83673469,  20000.        ])
```

These are the points at which we will get data for each ODE from the integrator. With this, we can now run our simulation. Enter the following commands to run the simulation:

```
>>> yout = odesolve(m.model, t)
```

To verify that the simulation run you can see the content of the *yout* object. For example, check for the content of the *Bid* observable defined previously:

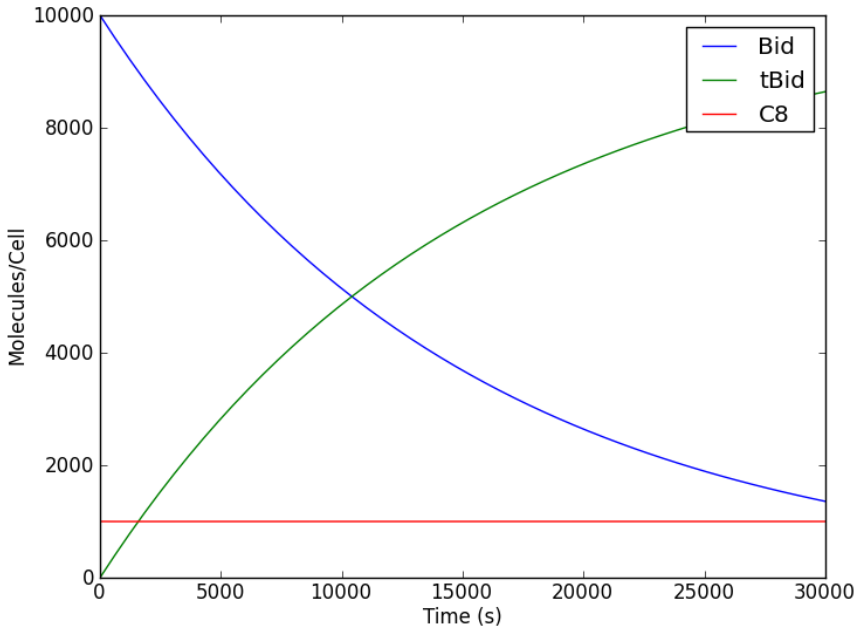
```
>>> yout['obsBid']
array([ 10000.        ,   9601.77865674,   9224.08135988,   8868.37855506,
         8534.45591732,   8221.19944491,   7927.08884234,   7650.48970981,
         7389.81105408,   7143.5816199 ,   6910.47836131,   6689.32927828,
         6479.10347845,   6278.89607041,   6087.91189021,   5905.45001654,
         5730.89003662,   5563.68044913,   5403.32856328,   5249.39176146,
         5101.47069899,   4959.20384615,   4822.26262101,   4690.34720441,
         4563.18294803,   4440.51745347,   4322.11815173,   4207.77021789,
         4097.27471952,   3990.44698008,   3887.11517373,   3787.11923497,
         3690.30945136,   3596.54594391,   3505.69733323,   3417.64025401,
         3332.25897699,   3249.44415872,   3169.09326717,   3091.10923365,
         3015.40034777,   2941.87977234,   2870.4652525 ,   2801.07879018,
         2733.64632469,   2668.09744369,   2604.36497901,   2542.38554596,
         2482.09776367,   2423.44473279])
```

As you may recall we named some observables in the Observables section above. The variable *yout* contains an array of all the ODE outputs from the integrators along with the named observables (i.e. *obsBid*, *obstBid*, and *obsC8*) which can be called by their names. We can therefore plot this data to visualize our output. Using the commands imported from the *PyLab* module we can create a graph interactively. Enter the commands as shown below:

```
>>>pl.ion()
>>>pl.figure()
```

```
>>>pl.plot(t, yout['obsBid'], label="Bid")
>>>pl.plot(t, yout['obstBid'], label="tBid")
>>>pl.plot(t, yout['obsC8'], label="C8")
>>>pl.legend()
>>>pl.xlabel("Time (s)")
>>>pl.ylabel("Molecules/cell")
>>>pl.show()
```

You should now have a figure in your screen showing the number of *Bid* molecules decreaing from the initial amount decreasing over time, the number of *tBid* molecules increasing over time, and the number of free *C8* molecules decrease to about half. For help with the above commands and to see more commands related to *PyLab* check the documentation [5]. Your figure should look something like the one below:



Congratulations! You have created your first model and run a simulation!

3.9 Visualization

It is useful to visualize the species and reactions that make a model. We have provided two methods to visualize species and reactions. We recommend using the tools in Kappa and BioNetGen for other visualization tools such as contact maps and stories.

The simplest way to visualize a model is to generate the graph file using the programs available from the command line. The files are located in the `.../pysb/tools` directory. The files to visualize reactions and species are `render_reactions.py` and `render_species.py`. These python scripts will generate `.dot` graph files that can be visualized using several tool such as OmniGraffle in `OS X` or GraphViz in all major platforms. For this tutorial we will use the GraphViz renderer. For this example will visualize the `mymodel.py` file that was created earlier. Issue the following command, replacing the comments inside square brackets‘‘[]‘‘ with the correct paths. We will first generate the `.dot` from the command line as follows:

```
[path-to-pysb]/pysb/tools/render_reactions.py [path-to-pysb-model-file]/mymodel.py > mymodel.dot
```

If your model can be properly visualized you should have gotten no errors and should now have a file called `mymodel.dot`. You can now use this file as an input for any visualization tool as described above. You can follow the same procedures with the `render_species.py` script to visualize the species generated by your models.

CHAPTER

FOUR

ADVANCED MODELING

In this section we continue with the above tutorial and touch on some advanced techniques for modeling using compartments, the definition of higher order rules using functions, and model calibration using the PySB utilities. Although we provide the functions and utilities we have found useful for the community, we encourage users to customize the modeling tools to their needs and add/contribute to the PySB modeling community.

4.1 Higher-order rules

For this section we will show the power working in a programming environment by creating a simple function called "catalyze". Catalysis happens quite often in models and it is one of the basic functions we have found useful in our model development. Rather than typing many lines such as:

```
Rule("association",  Enz(b=None) + Sub(b=None, S="i") <> Enz(b=1)%Sub(b=1,S="i"), kf, kr)
Rule("dissociation", Enz(b=1)%Sub(b=1,S="i") >> Enz(b=None) + Sub(b=None, S="a"), kc)
```

multiple times, we find it more powerful, transparent and easy to instantiate/edit a simple, one-line function call such as:

```
catalyze(Enz, Sub, "S", "i", "a", kf, kr, kc)
```

We find that the functional form captures what we mean to write: a chemical species (the substrate) undergoes catalytic activation (by the enzyme) with a given set of parameters. We will now describe how a function can be written in PySB to automate the scripting of simple concepts into a programmatic format. Examine the function below:

```
def catalyze(enz, sub, site, state1, state2, kf, kr, kc):    # (0) function call
    """2-step catalytic process"""                           # (1) reaction name
    r1_name = '%s_assoc_%s' % (enz.name, sub.name)           # (2) name of association reaction for ru
    r2_name = '%s_diss_%s' % (enz.name, sub.name)            # (3) name of dissociation reaction for
    E = enz(b=None)                                          # (4) define enzyme state in function
    S = sub({'b': None, site: state1})                       # (5) define substrate state in function
    ES = enz(b=1) % sub({'b': 1, site: state1})              # (6) define state of enzyme:substrate co
    P = sub({'b': None, site: state2})                       # (7) define state of product
    Rule(r1_name, E + S <> ES, kf, kr)                       # (8) rule for enzyme + substrate associa
    Rule(r2_name, ES >> E + P, kc)                           # (9) rule for enzyme:substrate dissociat
```

As shown it takes about ten lines to write the catalyze function (shorter variants are certainly possible with more advanced *Python* statements). The skeleton of every function in *Python*

As shown, *Monomers*, *Parameters*, *Species*, and pretty much anything related to rules-based modeling are instantiated as objects in *Python*. One could write functions to interact with these objects and they could be instantiated and inherit methods from a class. The limits to programming biology with PySB are those enforced by the *Python* language itself. We can now go ahead and embed this into a model. Go back to your `mymodel.py` file and modify it to look something like this:

```
# import the pysb module and all its methods and functions
from pysb import *


def catalyze(enz, sub, site, state1, state2, kf, kr, kc):    # function call
    """2-step catalytic process"""                           # reaction name
    r1_name = '%s_assoc_%s' % (enz.name, sub.name)            # name of association reaction for rule
    r2_name = '%s_diss_%s' % (enz.name, sub.name)            # name of dissociation reaction for rule
    E = enz(b=None)                                          # define enzyme state in function
    S = sub({'b': None, site: state1})                       # define substrate state in function
    ES = enz(b=1) % sub({'b': 1, site: state1})              # define state of enzyme:substrate comple
    P = sub({'b': None, site: state2})                       # define state of product
    Rule(r1_name, E + S <> ES, kf, kr)                       # rule for enzyme + substrate associatio
    Rule(r2_name, ES >> E + P, kc)                           # rule for enzyme:substrate dissociation

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})

# input the parameter values
Parameter('kf', 1.0e-07)
Parameter('kr', 1.0e-03)
Parameter('kc', 1.0)

# OLD RULES
# Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S=None) <> C8(b=1) % Bid(b=1, S=None), *[kf, kr])
# Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) + Bid(b=None, S='t'), kc)
#
# NEW RULES
# Catalysis
catalyze(C8, Bid, 'S', 'u', 't', kf, kr, kc)


# initial conditions
Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)
Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)

# Observables
Observable('obsC8', C8(b=None))
Observable('obsBid', Bid(b=None, S='u'))
Observable('obstBid', Bid(b=None, S='t'))
```

With this you should be able to execute your code and generate figures as described in the previous sections.

4.2 Using provided macros

For further reference we invite the users to explore the `macros.py` file in the `.../pysb/` directory. Based on our experience with modeling signal transduction pathways we have identified a set of commonly-used constructs that can serve as building blocks for more complex models. In addition to some meta-macros useful for instantiating user macros, we provide a set of macros such as `equilibrate`. `bind`, `catalyze`, `catalyze_one_step`, `catalyze_one_step_reversible`, `synthesize`, `degrade`, `assemble_pore_sequential`, and

pore_transport. In addition to these basic macros we also provide the higher-level macros bind_table and catalyze_table which we have found useful in instantiating the interactions between families of models.

In what follows we expand our previous model example of Caspase-8 by adding a few more species. The initiator caspase, as was described earlier, catalytically cleaves Bid to create truncated Bid (tBid) in this model. This tBid then catalytically activates Bax and Bak which eventually go on to form pores at the mitochondria leading to mitochondrial outer-membrane permeabilization (MOMP) and eventual cell death. To introduce the concept of higher-level macros we will show how the bind_table macro can be used to show how a family of inhibitors, namely Bcl-2, Bcl-xL, and Mcl-1 inhibits a family of proteins, namely Bid, Bax, and Bak.

In your favorite editor, go ahead and create a file (I will refer to it as ::file::*mymodel_fxns*). Many rules that dictate the interactions among species depend on a single binding site. We will begin by creating our model and declaring a generic binding site. We will also declare some functions, using the PySB macros and tailor them to our needs by specifying the binding site to be passed to the function. The first thing we do is import PySB and then import PySB macros. Then we declare our generic site and redefine the pysb.macros for our model as follows:

```
# import the pysb module and all its methods and functions
from pysb import *
from pysb.macros import *

# some functions to make life easy
site_name = 'b'
def catalyze_b(enz, sub, product, klist):
    """Alias for pysb.macros.catalyze with default binding site 'b'.
    """
    return catalyze(enz, site_name, sub, site_name, product, klist)
def bind_table_b(table):
    """Alias for pysb.macros.bind_table with default binding sites 'bf'.
    """
    return bind_table(table, site_name, site_name)
```

The first two lines just import the necessary modules from PySB. The catalyze_b' function, tailored for the model, takes four inputs but feeds six inputs to the pysb.macros.catalyze function, hence making the model more clean. Similarly the bind_table_b function takes only one entry, a list of lists, and feeds the entries needed to the pysb.macros.bind_table macro. Note that these entries could be contained in a header file to be hidden from the user at model time.

With this technical work out of the way we can now actually start our mdoel building. We will declare two sets of rates, the bid_rates that we will use for all the Bid interactions and the bcl2_rates which we will use for all the Bcl-2 interactions. Thesevalues could be specified individually as desired as desired but it is common practice in models to use generic values for the reaction rate parameters of a model and determine these in detail through some sort of model calibration. We will use these values for now for illustrative purposes.

The next entries for the rates, the model declaration, and the Monomers follow:

```
# Bid activation rates
bid_rates = [        1e-7, 1e-3, 1] #

# Bcl2 Inhibition Rates
bcl2_rates = [1.428571e-05, 1e-3] # 1.0e-6/v_mito

# instantiate a model
Model()

# declare monomers
Monomer('C8',     ['b'])
Monomer('Bid',    ['b', 'S'], {'S':['u', 't', 'm']})
Monomer('Bax',    ['b', 'S'], {'S':['i', 'a', 'm']})
Monomer('Bak',    ['b', 'S'], {'S':['i', 'a']})
```

```
Monomer('BclxL', ['b', 'S'], {'S':['c', 'm']})
Monomer('Bcl2', ['b'])
Monomer('Mcl1', ['b'])
```

As shown, the generic rates are declared followed by the declaration of the monomers. We have the `C8` and `Bid` monomers as we did in the initial part of the tutorial, the MOMP effectors `Bid`, `Bax`, `Bak`, and the MOMP inhibitors `Bcl-xL`, `Bcl-2`, and `Mcl-1`. The `Bid`, `Bax`, and `BclxL` monomers, in addition to the active and inactive terms also have a `'m'` term indicating that they can be in a membrane, which in this case we indicate as a state. We will have a translocation to the membrane as part of the reactions.

We can now begin to write some checmical procedures. The first procedure is the catalytic activation of `Bid` by `C8`. This is followed by the catalytic activation of Bax and Bak.

```
# Activate Bid
catalyze_b(C8, Bid(S='u'), Bid(S='t'), [KF, KR, KC])

# Activate Bax/Bak
catalyze_b(Bid(S='m'), Bax(S='i'), Bax(S='m'), bid_rates)
catalyze_b(Bid(S='m'), Bak(S='i'), Bak(S='a'), bid_rates)
```

As shown, we simply state the soecies that acts as an *enzyme* as the first function argument, the species that acts as the *reactant* with the enzyme as the second argument (along with any state specifications) and finally the *product* species. The `bid_rates` argument is the list of rates that we declared earlier.

You may have noticed a problem with the previous statements. The `Bid` species undergoes a transformation from state `S='u'` to `S='t'` but the activation of `Bax` and `Bak` happens only when `Bid` is in state `S='m'` to imply that these events only happen at the membrane. In order to transport `Bid` from the `'t'` state to the `'m'` state we need a transporf function. We achieve this by using the *equilibrate* macro in PySB between these states. In addition we use this same macro for the transport of the `Bax` species and the `BclxL` species as shown below.

```
# Bid, Bax, BclxL "transport" to the membrane
equilibrate(Bid(b=None, S='t'),   Bid(b=None, S='m'), [1e-1, 1e-3])
equilibrate(Bax(b=None, S='m'),   Bax(b=None, S='a'), [1e-1, 1e-3])
equilibrate(BclxL(b=None, S='c'), BclxL(b=None, S='m'), [1e-1, 1e-3])
```

According to published experimental data, the Bcl-2 family of inhibitors can inhibit the initiator `Bid` and the effector `Bax` and `Bak`. These family has complex interactions with all these proteins. Given that we have three inhibitors, and three molecules to be inhibited, this indicates nine interactions that need to be specified. This would involve writing nine reversible reactions in a rules language or at least eighteen reactions for each direction if we were writing the ODEs. Given that we are simply stating that these species *bind* to inhibit interactions, we can take advantage of two things. In the first case we have already seen that there is a *bind* macro specified in PySB. We can further functionalize this into a higher level macro, naemly the *bind_table* macro, which takes a table of interactions as an argument and generates the rules based on these simple interactions. We specify the bind table for the inhibitors (top row) and the inhibited molecules (left column) as follows.

```
bind_table_b([[                  Bcl2,  BclxL(S='m'),       Mcl1],
              [Bid(S='m'), bcl2_rates,  bcl2_rates,   bcl2_rates],
              [Bax(S='a'), bcl2_rates,  bcl2_rates,         None],
              [Bak(S='a'),       None,  bcl2_rates,   bcl2_rates]])
```

As shown the inhibitors interact by giving the rates of interactions or the *"None"* Python keyword to indicate no interaction. The only thing left to run this simple model is to declare some initial conditions and some observables. We declare the following:

```
# initial conditions
Parameter('C8_0',    1e4)
Parameter('Bid_0',   1e4)
Parameter('Bax_0',  .8e5)
```

```
Parameter('Bak_0',  .2e5)
Parameter('BclxL_0', 1e3)
Parameter('Bcl2_0',  1e3)
Parameter('Mcl1_0',  1e3)

Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)
Initial(Bax(b=None, S='i'), Bax_0)
Initial(Bak(b=None, S='i'), Bak_0)
Initial(BclxL(b=None, S='c'), BclxL_0)
Initial(Bcl2(b=None), Bcl2_0)
Initial(Mcl1(b=None), Mcl1_0)

# Observables
Observable('obstBid', Bid(b=None, S='m'))
Observable('obsBax', Bax(b=None, S='a'))
Observable('obsBak', Bax(b=None, S='a'))
Observable('obsBaxBclxL', Bax(b=1, S='a')%BclxL(b=1, S='m'))
```

By now you should have a file with all the components that looks something like this:

```
# import the pysb module and all its methods and functions
from pysb import *
from pysb.macros import *

# some functions to make life easy
site_name = 'b'
def catalyze_b(enz, sub, product, klist):
    """Alias for pysb.macros.catalyze with default binding site 'b'.
    """
    return catalyze(enz, site_name, sub, site_name, product, klist)

def bind_table_b(table):
    """Alias for pysb.macros.bind_table with default binding sites 'bf'.
    """
    return bind_table(table, site_name, site_name)

# Default forward, reverse, and catalytic rates
KF = 1e-6
KR = 1e-3
KC = 1

# Bid activation rates
bid_rates = [        1e-7, 1e-3, 1] #

# Bcl2 Inhibition Rates
bcl2_rates = [1.428571e-05, 1e-3] # 1.0e-6/v_mito

# instantiate a model
Model()

# declare monomers
Monomer('C8',    ['b'])
Monomer('Bid',   ['b', 'S'], {'S':['u', 't', 'm']})
Monomer('Bax',   ['b', 'S'], {'S':['i', 'a', 'm']})
Monomer('Bak',   ['b', 'S'], {'S':['i', 'a']})
Monomer('BclxL', ['b', 'S'], {'S':['c', 'm']})
Monomer('Bcl2', ['b'])
Monomer('Mcl1', ['b'])
```

```
# Activate Bid
catalyze_b(C8, Bid(S='u'), Bid(S='t'), [KF, KR, KC])

# Activate Bax/Bak
catalyze_b(Bid(S='m'), Bax(S='i'), Bax(S='m'), bid_rates)
catalyze_b(Bid(S='m'), Bak(S='i'), Bak(S='a'), bid_rates)

# Bid, Bax, BclxL "transport" to the membrane
equilibrate(Bid(b=None, S='t'),   Bid(b=None, S='m'), [1e-1, 1e-3])
equilibrate(Bax(b=None, S='m'),   Bax(b=None, S='a'), [1e-1, 1e-3])
equilibrate(BclxL(b=None, S='c'), BclxL(b=None, S='m'), [1e-1, 1e-3])


bind_table_b([[                  Bcl2,  BclxL(S='m'),       Mcl1],
              [Bid(S='m'), bcl2_rates,  bcl2_rates,   bcl2_rates],
              [Bax(S='a'), bcl2_rates,  bcl2_rates,         None],
              [Bak(S='a'),       None,  bcl2_rates,   bcl2_rates]])

# initial conditions
Parameter('C8_0',    1e4)
Parameter('Bid_0',   1e4)
Parameter('Bax_0',  .8e5)
Parameter('Bak_0',  .2e5)
Parameter('BclxL_0', 1e3)
Parameter('Bcl2_0',  1e3)
Parameter('Mcl1_0',  1e3)

Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)
Initial(Bax(b=None, S='i'), Bax_0)
Initial(Bak(b=None, S='i'), Bak_0)
Initial(BclxL(b=None, S='c'), BclxL_0)
Initial(Bcl2(b=None), Bcl2_0)
Initial(Mcl1(b=None), Mcl1_0)

# Observables
Observable('obstBid', Bid(b=None, S='m'))
Observable('obsBax', Bax(b=None, S='a'))
Observable('obsBak', Bax(b=None, S='a'))
Observable('obsBaxBclxL', Bax(b=1, S='a')%BclxL(b=1, S='m'))
```

With this you should be able to run the simulations and generate figures as described in the basic tutorial sections.

4.3 Compartments

We will continue building on your `mymodel_fxns.py` file and add one more species and a compartment. In extrinsic apoptosis, once *tBid* is activated it translocates to the outer mitochondrial membrane where it interacts with the protein *Bak* (residing in the membrane).

4.4 Model calibration

COMING SOON: ANNEAL

4.5 Modules

CHAPTER

FIVE

ABOUT RULES

5.1 Overview

In rules-based modeling, units that undergo transformations such as proteins, small molecules, protein complexes, etc are termed *species*. The interactions among these *species* are then represented using structured objects that describe the interactions between the *species* and constitute what we describe as *rules*. The specific details of how species and rules are specified can vary across different rules-based modeling approaches. In PySB we have chosen to ascribe to the approaches found in BioNetGen and Kappa, but other approaches are certainly possible for advanced users interested in modifying the source codbe. Each rule, describing the interaction between *species* or sets of *species* must be assigned a set of *parameters* associated with the nature of the *rule*. Given that BioNetGen and Kappa both describe interactions using a mass-action kinetics formalism, the *parameters* will necessarily consist of reaction rates. In what follows we describe how a model can be instantiated in PySB, how *species* and *rules* are specified, and how to run a simple simulation.

5.2 Reference to Rules-based languages

PySB uses the rules languages grammar of BioNetGen and Kappa almost verbatim with the differences being mostly synctactic. This has been done on purpose to keep compatibility with these languages and leverage their available simulating tools. We invited interested users to explore the BioNetGen Tutorial or the Introduction to Kappa Syntax pages for further information. Understanding of any of these languages will make working with PySB rules a very straightforward exercise for the user.

CHAPTER

SIX

PYSB MODULES REFERENCE

6.1 PySB core

class pysb.core.**Compartment**(*name*, *parent=None*, *dimension=3*, *size=None*, *_export=True*)
Model component representing a bounded reaction volume.

Methods

class pysb.core.**ComplexPattern**(*monomer_patterns*, *compartment*, *match_once=False*)
A bound set of MonomerPatterns, i.e. a pattern to match a complex.

In BNG terms, a list of patterns combined with the '.' operator.

Methods

copy()
Implement our own brand of shallow copy.

The new object will have references to the original compartment, and copies of the monomer_patterns.

is_concrete()
Return a bool indicating whether the pattern is 'concrete'.

'Concrete' means the pattern satisfies ANY of the following: 1. All monomer patterns are concrete 2. The compartment is specified AND all monomer patterns are site-concrete

is_equivalent_to(*other*)
Checks for equality with another ComplexPattern

class pysb.core.**Component**(*name*, *_export=True*)
The base class for all the things contained within a model.

Methods

exception pysb.core.**ComponentDuplicateNameError**
Issued by ComponentSet.add when a component is added with the same name as an existing one.

class pysb.core.**ComponentSet**(*iterable=[]*)
An add-and-read-only container for storing model Components. It behaves mostly like an ordered set, but components can also be retrieved by name *or* index by using the [] operator (like a dict or list). Components may not be removed or replaced.

Methods

rename(*c*, *new_name*)
Change a component's name in our data structures

exception pysb.core.**InvalidComponentNameError**(*name*)
Issued by Component.__init__ when the given name is not valid.

class pysb.core.**Model**(*name=None*, *_export=True*)
Container for monomers, compartments, parameters, and rules.

Methods

all_component_sets()
Return a list of all ComponentSet objects

enable_synth_deg()
Add components needed to support synthesis and degradation rules.

has_synth_deg()
Return true if model uses synthesis or degradation reactions.

parameters_compartments()
Returns a ComponentSet of the parameters used as compartment sizes

parameters_initial_conditions()
Returns a ComponentSet of the parameters used as initial conditions

parameters_rules()
Returns a ComponentSet of the parameters used as rate constants in rules

parameters_unused()
Returns a ComponentSet of the parameters not used in the model at all

reset_equations()
Clear out anything generated by bng.generate_equations or the like

exception pysb.core.**ModelExistsWarning**
Issued by Model constructor when a second model is defined.

class pysb.core.**Monomer**(*name*, *sites=[]*, *site_states={}*, *_export=True*)
Model component representing a protein or other molecule.

Methods

class pysb.core.**MonomerAny**
A wildcard monomer which matches any species.

This is only needed where you would use a '+' in BNG.

Methods

class pysb.core.**MonomerPattern**(*monomer*, *site_conditions*, *compartment*)
A pattern which matches instances of a given monomer, possibly with restrictions on the state of certain sites.

Methods

is_concrete()
Return a bool indicating whether the pattern is 'concrete'.

'Concrete' means the pattern satisfies ALL of the following: 1. All sites have specified conditions 2. If the model uses compartments, the compartment is specified.

is_site_concrete()
Return a bool indicating whether the pattern is 'site-concrete'.

'Site-concrete' means all sites have specified conditions.

class pysb.core.**MonomerWild**
A wildcard monomer which matches any species, or nothing (no bond).

This is only needed where you would use a '?' in BNG.

Methods

class pysb.core.**Observable**(*name*, *reaction_pattern*, *_export=True*)
Model component representing a linear combination of species.

May be used in rate law expressions.

Methods

class pysb.core.**Parameter**(*name*, *value=0.0*, *_export=True*)
Model component representing a named constant floating point number.

Parameters are used as reaction rate constants, compartment volumes and initial (boundary) conditions for species.

Methods

class pysb.core.**ReactionPattern**(*complex_patterns*)
A pattern for the entire product or reactant side of a rule.

Essentially a thin wrapper around a list of ComplexPatterns. In BNG terms, a list of complex patterns combined with the '+' operator.

class pysb.core.**RuleExpression**(*reactant_pattern*, *product_pattern*, *is_reversible*)
A container for the reactant and product patterns of a rule expression.

Contains one ReactionPattern for each of reactants and products, and a boolean indicating reversibility. This is a temporary object used to implement syntactic sugar through operator overloading. The Rule constructor takes an instance of this class as its first argument, but simply extracts its fields and discards the object itself.

exception pysb.core.**SymbolExistsWarning**
Issued by model component constructors when a name is reused.

pysb.core.**as_complex_pattern**(*v*)
Internal helper to 'upgrade' a MonomerPattern to a ComplexPattern.

pysb.core.**as_reaction_pattern**(*v*)
Internal helper to 'upgrade' a Complex- or MonomerPattern to a complete ReactionPattern.

pysb.core.**extract_site_conditions**(**args*, ***kwargs*)
Handle parsing of MonomerPattern site conditions.

6.2 BioNetGen integration

6.3 Macros

pysb.macros.**equilibrate**(*s1*, *s2*, *klist*)
Generate the unimolecular reversible equilibrium reaction S1 <-> S2.

Parameters **s1, s2** : Monomer or MonomerPattern

S1 and S2 in the above reaction.

klist : list of 2 Parameters or list of 2 numbers

Forward (S1 -> S2) and reverse rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of S1 and S2 and these parameters will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains one reversible Rule and optionally two Parameters if klist was given as plain numbers.

pysb.macros.**bind**(*s1*, *site1*, *s2*, *site2*, *klist*)
Generate the reversible binding reaction S1 + S2 <> S1:S2.

Parameters **s1, s2** : Monomer or MonomerPattern

Monomers participating in the binding reaction.

site1, site2 : string

The names of the sites on s1 and s2 used for binding.

klist : list of 2 Parameters or list of 2 numbers

Forward and reverse rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of S1 and S2 and these parameters will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains the bidirectional binding Rule and optionally two Parameters if klist was given as numbers.

Examples

Model() Monomer('A', ['x']) Monomer('B', ['y']) bind(A, 'x', B, 'y', [1e-4, 1e-1])

pysb.macros.**bind_table**(*bindtable*, *row_site*, *col_site*)
Generate a table of reversible binding reactions.

Given two lists of species R and C, calls the *bind* macro on each pairwise combination (R[i], C[j]). The species lists and the parameter values are passed as a list of lists (i.e. a table) with elements of R passed as the "row headers", elements of C as the "column headers", and forward / reverse rate pairs (in that order) as tuples in the

"cells". For example with two elements in each of R and C, the table would appear as follows (note that the first row has one fewer element than the subsequent rows):

```
[[              C1,            C2],
 [R1, (1e-4, 1e-1), (2e-4, 2e-1)],
 [R2, (3e-4, 3e-1), (4e-4, 4e-1)]]
```

Each parameter tuple may contain Parameters or numbers. If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of the relevant species and these parameters will be included at the end of the returned component list. To omit any individual reaction, pass None in place of the corresponding parameter tuple.

Parameters **bindtable** : list of lists

Table of reactants and rates, as described above.

row_site, col_site : string

The names of the sites on the elements of R and C, respectively, used for binding.

Returns **components** : ComponentSet

The generated components. Contains the bidirectional binding Rules and optionally the Parameters for any parameters given as numbers.

Examples

Model() Monomer('R1', ['x']) Monomer('R2', ['x']) Monomer('C1', ['y']) Monomer('C2', ['y']) bind_table([[C1, C2],

[R1, (1e-4, 1e-1), (2e-4, 2e-1)], [R2, (3e-4, 3e-1), None]],

'x', 'y')

pysb.macros.**catalyze**(*enzyme*, *e_site*, *substrate*, *s_site*, *product*, *klist*)

Generate the two-step catalytic reaction E + S <> E:S >> E + P.

Parameters **enzyme, substrate, product** : Monomer or MonomerPattern

E, S and P in the above reaction.

e_site, s_site : string

The names of the sites on *enzyme* and *substrate* (respectively) where they bind each other to form the E:S complex.

klist : list of 3 Parameters or list of 3 numbers

Forward, reverse and catalytic rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of enzyme, substrate and product and these parameters will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains two Rules (bidirectional complex formation and unidirectional product dissociation), and optionally three Parameters if klist was given as plain numbers.

Notes

When passing a MonomerPattern for *enzyme* or *substrate*, do not include *e_site* or *s_site* in the respective patterns. The macro will handle this.

Examples

Using distinct Monomers for substrate and product:

```
Model()
Monomer('E', ['b'])
Monomer('S', ['b'])
Monomer('P')
catalyze(E, 'b', S, 'b', P, (1e-4, 1e-1, 1))
```

Using a single Monomer for substrate and product with a state change:

```
Monomer('Kinase', ['b'])
Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
catalyze(Kinase, 'b', Substrate(y='U'), 'b', Substrate(y='P'),
         (1e-4, 1e-1, 1))
```

pysb.macros.**catalyze_state**(*enzyme*, *e_site*, *substrate*, *s_site*, *mod_site*, *state1*, *state2*, *klist*)

Generate the two-step catalytic reaction E + S <> E:S >> E + P. A wrapper around catalyze() with a signature specifying the state change of the substrate resulting from catalysis.

Parameters **enzyme** : Monomer or MonomerPattern

E in the above reaction.

substrate : Monomer or MonomerPattern

S and P in the above reaction. The product species is assumed to be identical to the substrate species in all respects except the state of the modification site. The state of the modification site should not be specified in the MonomerPattern for the substrate.

e_site, s_site : string

The names of the sites on *enzyme* and *substrate* (respectively) where they bind each other to form the E:S complex.

mod_site : string

The name of the site on the substrate that is modified by catalysis.

state1, state2 : strings

The states of the modification site (mod_site) on the substrate before (state1) and after (state2) catalysis.

klist : list of 3 Parameters or list of 3 numbers

Forward, reverse and catalytic rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of enzyme, substrate and product and these parameters will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains two Rules (bidirectional complex formation and unidirectional product dissociation), and optionally three Parameters if klist was given as plain numbers.

Notes

When passing a MonomerPattern for *enzyme* or *substrate*, do not include *e_site* or *s_site* in the respective patterns. In addition, do not include the state of the modification site on the substrate. The macro will handle this.

Examples

Using a single Monomer for substrate and product with a state change:

```
Monomer('Kinase', ['b'])
Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
catalyze_state(Kinase, 'b', Substrate, 'b', 'y', 'U', 'P',
          (1e-4, 1e-1, 1))
```

pysb.macros.**catalyze_table**()

table[0]: [E1, ..., En] table[1]: [S1(site='ssub'), S1(site='sprod'), (kf, kr, kc), ..., (...)] (TABLE, 'sbsite', 'ebsite')

table[0]: [E1, ..., En] table[1]: [S1, 'site', 'ssub', 'sprod' , (kf, kr, kc), ..., (...)] (TABLE, 'sbsite', 'ebsite')

table[0]: [E1, ..., En] table[1]: [S1, (kf, kr, kc), ..., (...)] (TABLE, 'sbsite', 'ebsite' 'smodsite', 'ssub', 'sprod')

pysb.macros.**catalyze_one_step**(*enzyme*, *substrate*, *product*, *kf*)

Generate the one-step catalytic reaction E + S >> E + P.

Parameters **enzyme, substrate, product** : Monomer or MonomerPattern

E, S and P in the above reaction.

kf : a Parameter or a number

Forward rate constant for the reaction. If a Parameter is passed, it will be used directly in the generated Rules. If a number is passed, a Parameter will be created with an automatically generated name based on the names and states of the enzyme, substrate and product and this parameter will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains the unidirectional reaction Rule and optionally the forward rate Parameter if klist was given as a number.

Notes

In this macro, there is no direct binding between enzyme and substrate, so binding sites do not have to be specified. This represents an approximation for the case when the enzyme is operating in its linear range. However, if catalysis is nevertheless contingent on the enzyme or substrate being unbound on some site, then that information must be encoded in the MonomerPattern for the enzyme or substrate. See the examples, below.

If the ability of the enzyme E to catalyze this reaction is dependent on the site 'b' of E being unbound, then this macro must be called as

catalyze_one_step(E(b=None), S, P, 1e-4)

and similarly if the substrate or product must be unbound.

pysb.macros.**catalyze_one_step_reversible**(*enzyme*, *substrate*, *product*, *klist*)

Create fwd and reverse rules for catalysis of the form: E + S -> E + P

P -> S

Parameters **enzyme, substrate, product** : Monomer or MonomerPattern

E, S and P in the above reactions.

klist : list of 2 Parameters or list of 2 numbers

A list containing the rate constant for catalysis and the rate constant for the conversion of product back to substrate (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of S1 and S2 and these parameters will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains two rules (the single-step catalysis rule and the product reversion rule) and optionally the two generated Parameter objects if klist was given as numbers.

Notes

Calls the macro catalyze_one_step to generate the catalysis rule.

Examples

Model() Monomer('E', ['b']) Monomer('S', ['b']) Monomer('P') catalyze_one_step_reversible(E, S, P, [1e-1, 1e-4])

pysb.macros.**synthesize**(*species*, *ksynth*)

Generate a reaction which synthesizes a species.

Note that *species* must be "concrete", i.e. the state of all sites in all of its monomers must be specified. No site may be left unmentioned.

Parameters **species** : Monomer, MonomerPattern or ComplexPattern

The species to synthesize. If a Monomer, sites are considered as unbound and in their default state. If a pattern, must be concrete.

ksynth : Parameters or number

Synthesis rate. If a Parameter is passed, it will be used directly in the generated Rule. If a number is passed, a Parameter will be created with an automatically generated name based on the names and site states of the components of *species* and this parameter will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains the unidirectional synthesis Rule and optionally a Parameter if ksynth was given as a number.

Examples

Model() Monomer('A', ['x', 'y'], {'y': ['e', 'f']}) synthesize(A(x=None, y='e'), 1e-4)

pysb.macros.**degrade**(*species*, *kdeg*)

Generate a reaction which degrades a species.

Note that *species* is not required to be "concrete".

Parameters **species** : Monomer, MonomerPattern or ComplexPattern

The species to synthesize. If a Monomer, sites are considered as unbound and in their default state. If a pattern, must be concrete.

kdeg : Parameters or number

Degradation rate. If a Parameter is passed, it will be used directly in the generated Rule. If a number is passed, a Parameter will be created with an automatically generated name based on the names and site states of the components of *species* and this parameter will be included at the end of the returned component list.

Returns **components** : ComponentSet

The generated components. Contains the unidirectional degradation Rule and optionally a Parameter if ksynth was given as a number.

Examples

Model() Monomer('B', ['x']) degrade(B(), 1e-6) # degrade all B, even bound species

pysb.macros.**synthesize_degrade_table**(*table*)

Generate a table of synthesis and degradation reactions.

Given a list of species, calls the *synthesize* and *degrade* macros on each one. The species and the parameter values are passed as a list of lists (i.e. a table) with each inner list consisting of the species, forward and reverse rates (in that order).

Each species' associated pair of rates may be either Parameters or numbers. If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of the relevant species and these parameters will be included in the returned component list. To omit any individual reaction, pass None in place of the corresponding parameter.

Note that any *species* with a non-None synthesis rate must be "concrete".

Parameters **table** : list of lists

Table of species and rates, as described above.

Returns **components** : ComponentSet

The generated components. Contains the unidirectional synthesis and degradation Rules and optionally the Parameters for any rates given as numbers.

Examples

Model() Monomer('A', ['x', 'y'], {'y': ['e', 'f']}) Monomer('B', ['x']) synthesize_degrade_table([[A(x=None, y='e'), 1e-4, 1e-6],

[B(), None, 1e-7]])

pysb.macros.**assemble_pore_sequential**(*subunit*, *site1*, *site2*, *max_size*, *ktable*)

Generate rules to assemble a circular homomeric pore sequentially.

The pore species are created by sequential addition of *subunit* monomers, i.e. larger oligomeric species never fuse together. The pore structure is defined by the *pore_species* macro.

Parameters **subunit** : Monomer or MonomerPattern

The subunit of which the pore is composed.

site1, site2 : string

The names of the sites where one copy of *subunit* binds to the next.

max_size : integer

The maximum number of subunits in the pore.

ktable : list of lists of Parameters or numbers

Table of forward and reverse rate constants for the assembly steps. The outer list must be of length *max_size* - 1, and the inner lists must all be of length 2. In the outer list, the first element corresponds to the first assembly step in which two monomeric subunits bind to form a 2-subunit complex, and the last element corresponds to the final step in which the *max_size'th subunit is added. Each inner list contains the forward and reverse rate constants (in that order) for the corresponding assembly reaction, and each of these pairs must comprise solely Parameter objects or solely numbers (never one of each). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on 'subunit*, *site1*, *site2* and the pore sizes and these parameters will be included at the end of the returned component list.

pysb.macros.**pore_transport**(*subunit*, *sp_site1*, *sp_site2*, *sc_site*, *min_size*, *max_size*, *csource*, *c_site*, *cdest*, *ktable*)

Generate rules to transport cargo through a circular homomeric pore.

The pore structure is defined by the *pore_species* macro – *subunit* monomers bind to each other from *sp_site1* to *sp_site2* to form a closed ring. The transport reaction is modeled as a catalytic process of the form pore + csource <> pore:csource >> pore + cdest

Parameters **subunit** : Monomer or MonomerPattern

Subunit of which the pore is composed.

sp_site1, sp_site2 : string

Names of the sites where one copy of *subunit* binds to the next.

sc_site : string

Name of the site on *subunit* where it binds to the cargo *csource*.

min_size, max_size : integer

Minimum and maximum number of subunits in the pore at which transport will occur.

csource : Monomer or MonomerPattern

Cargo "source", i.e. the entity to be transported.

c_site : string

Name of the site on *csource* where it binds to *subunit*.

cdest : Monomer or MonomerPattern

Cargo "destination", i.e. the resulting state after the transport event.

ktable : list of lists of Parameters or numbers

Table of forward, reverse and catalytic rate constants for the transport reactions. The outer list must be of length *max_size* - *min_size* + 1, and the inner lists must all be of length 3. In the outer list, the first element corresponds to the transport through the pore of size *min_size* and the last element to that of size *max_size*. Each inner list contains the forward, reverse and catalytic rate constants (in that order) for the corresponding transport reaction, and each of these pairs must comprise solely Parameter objects or solely numbers (never some of each). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the subunit, the pore size and the cargo, and these parameters will be included at the end of the returned component list.

CHAPTER

SEVEN

ABOUT

7.1 About PySB

PySB is an open source tool developed in the laboratory of Peter K. Sorger at Harvard Medical School by C. F. Lopez, J. L. Muhlich, and J. A. Bachman. *PySB* was motivated by the desire to create a mathematical representation of a "biological word model" in a manner that does not result in significant loss of information. Our approach aimed to embed a rules-based language formalism within a fully developed programming language. We wanted the language to be simple, flexible, extensible, and easy to learn. After much work, the idea to develop *PySB* in the *Python* programming language began to mature. At the time of release *PySB* consists of a set of core algorithms that allow the user to develop programs, similar to the way a user thinks about biology, using *Python* at a biological rules-language level. *PySB* also provides a set of biochemical macros to enable model creation and offers a novel paradigm in biological model building in a modular manner. *PySB*, through in-house developed code or through available *Python* modules, has the capablities to develop models of any arbitrary size, numerical integration using ODEs, numerical model analysis, computer experiment plotting, model visualization, symbolic math manipulations, and provides a base for users to develop interactions with any tools available in the *Python* programming language.

7.2 About the Developers

Carlos F. Lopez received his B.A. and B.S. from University of Miami, Ph. D. from University of Pennsylvania and trained as a postdoctoral fellow with Peter J. Rossky at University of Texas at Austin and Peter K Sorger at Harvard Medical School. He is now an assistant professor at Vanderbilt University, department of Cancer Biology where he continues to contribute to PySB.

Jeremy L. Muhlich received his B.S. from Johns Hopkins University and is a Staff member in the laboratory of Professor Peter K. Sorger. He is an active developer of PySB.

John A. Bachman received his B.S. from Harvard University and is working toward his Ph. D. at Harvard Medical School. He continues to contribute and use PySB for his work.

***Peter K. Sorger* received his A.B. from Harvard College, Ph.D. from** Trinity College Cambridge, U.K. and trained as a postdoctoral fellow with Harold Varmus and Andrew Murray at the University of California, San Francisco. Sorger is the Otto Krayer Professor of Systems Pharmacology at Harvard Medical School (HMS) and co-founder of the Open Microscopy Environment (OME), MIT's Computational and Systems Biology Initiative (CSBi), the Council for Systems Biology in Boston (CSB2; www.csb2.org) and companies that include Merrimack Pharmaceuticals and Glencoe Software and serves on the scientific advisory and corporate boards of several other technology companies. He is director of the Center for Cell Decision Processes, an NIH Center of Excellence in Systems Biology and co-directs a new HMS program in Systems Pharmacology.

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

PYTHON MODULE INDEX

p