BioNetGen

Ali Sinan Saglam & Caleb Armstrong

Jul 08, 2022

CONTENTS:

BioNetGen is software designed for modular, structure-based modeling of biochemical reaction networks. It can be applied to many other types of modeling as well. It provides a simple, graph-based syntax that lets users build reaction models out of structured objects that can bind and undergo modification.

Visit https://bionetgen.org/ for more information.

Under the **Contents** directory below, you will find more information about the current software for rule-based modeling in BioNetGen. The **VS Code BNGL extension** includes many helpful features, including a GUI, to aid in using BioNetGen in the VS Code editor. **PyBioNetGen** provides a lightweight CLI and Python library for programmatic manipulation of BioNetGen models. Installation instructions for both can be found at the link below.

CHAPTER

ONE

INSTALLATION

1.1 Installing under VS Code

You will need to do the following to install the extension:

1. Download and install VS Code
2. Install BioNetGen extension for VS Code
3. Download and install Anaconda python
4. (Windows users only) Install Perl
5. Open a `.bngl` file in VS Code

First, download and install VS Code. The suggested way to install this VS Code extension is from the VS Code marketplace. Simply open the Extensions tab (or press `CTRL/CMD + SHIFT + X`) after opening VS Code and search for BioNetGen. This package will show up, click install. If this step doesn't work, please make sure you have the latest version.

Next, you will need Python installed, and you need to have your terminal environment set correctly. We recommend Anaconda Python. Download and install Anaconda Python. During the installation, we recommend adding Anaconda to your PATH environment variable when given the option. In VS Code, open a new terminal (`CTRL/CMD + `` ` `` or under `Terminal -> New Terminal`) and test if you have access to Python package manager `pip` (by running `pip -h`, for example). If you don't, you need to setup your environment so that the terminal has access to `pip`; this is OS dependent and there are various guides you can find online.

Windows users should also install Perl at this point if not already installed. You can use `conda install -c conda-forge perl` to do this. If you do not wish to use Anaconda, or if you do not wish to install Perl through Anaconda, we suggest Strawberry Perl.

Finally, open a `.bngl` file in VS Code; this should check if PyBioNetGen is installed, and will automatically install it if not. Once complete, make sure it's installed correctly by running `bionetgen -h`. If this doesn't work, you can open the command palette (`CTRL/CMD + SHIFT + P`) and run the `BNG setup` command. Alternatively, you can open a terminal and run the install command (`pip install bionetgen`) yourself.

1.1.1 Additional Setup Advice

You may need to specify which installation of Python that VS Code uses. To do so, open the command palette (`CTRL/CMD + SHIFT + P`) and select the command `Python:  Select Interpreter`, which will allow you to make your selection.

CHAPTER

TWO

VS CODE BNGL EXTENSION

This is a VS Code extension for BioNetGen language. It can be found in VS Code marketplace.

2.1 Usage

BioNetGen modelling language is a language for writing rule-based models of biochemical systems, including signal transduction, metabolic, and genetic regulatory networks, see here for more information.

This VS Code extension is designed to help write BNGL models by adding syntax highlighting and snippet support, do rapid tests of the model as you write with the help of a built-in run button and basic plotting features.

2.1.1 Syntax highlighting and snippets

Once the extension is installed you can create a new file with `.bngl`. This file extension will be automatically detected and you should see a run button at the top right corner of the file if the extension is running correctly. This extension will also do syntax highlighing on files with `.net` extension.

Next you can start writing your model. This VS Code extension supports a large list of snippets that can help you write your model. For a full list, see here, we will update this with a snippet guide in the future.

2.1.2 Using the correct theme

If you notice that there is no highlighting on certain parts of the model or if the colors don't match with the figures presented here, please make sure you have the `dark-bngl` theme activated (see here to learn how to select color themes). Currently only a dark theme is supported, we will include a light version in the future.

2.1.3 Running a model

Once you finished writing the model, you can try running it. For the run button to work, the default terminal window VS Code opens should have access to Perl, Python3 (preferably anaconda python) and the PyBioNetGen library. See *Installation* for more instructions on how to install the library. You can test if you have the library correctly installed by opening a new terminal and running `bionetgen -h`.

Once you press the run button (or use the shortcut `CTRL/CMD+SHIFT+F1`), the extension should create a new folder with the same name as the model. A time stamped folder will also be created and the current model will be copied under there and the extension will use the terminal and the PyBioNetGen library to run the model. Once the run completes, if the run created a `.gdat` file, it should open automatically.

2.1.4 Plotting results

Once you have some `gdat/cdat/scan` files to look at, you can open one and you should see two new buttons instead of the run button. The white and red button on the right should plot your file into a png with some basic defaults (options to change these options will be provided in future releases).

The blue button on the left should open a new window with an interactive plot (thanks to plotly.js). You can also use the `CTRL/CMD+SHIFT+F1` shortcut. The plotly window also allows you to save the image as a png as well as a svg file and has interactive features. You can also change the plotting type to markers or lines+markers and use one of the selection tools to sub-select time series. If you select a region on the plot with no data points, every time series in the original dataset will be shown again.

2.1.5 Visualization

You can also click on the visualize button that can be found to the left of the run button. This will generate all possible visualizations that's available to BioNetGen in a separate folder as GraphML files. These files are designed to be used in conjunction with yEd. If you have yEd installed, you can associate yEd with GraphML files in your OS (example for windows 10). If you also install the open in external app extension for VS Code you can right click the GraphML files and click on *Open in external App* to open the graph in yEd.

2.2 Options

There are various options for the BioNetGen extension that users can set to their preference. To access them, open the command palette (`CTRL/CMD + SHIFT + P`) and select `Preferences: Open User Settings`. Then select `BioNetGen` under the `Extensions` dropdown, or simply use the search bar.

See the descriptions of the options below:

2.2.1 `General: Auto_install`

The extension checks if BioNetGen is installed when a `.bngl` file is opened, and installs it if not found. Can be enabled/disabled here.

2.2.2 General: `Auto_open`

The extension can automatically open files such as `.gdat`, `.cdat`, or `.scan` that are generated by running a model via the `Run BNG` button or the `bionetgen run` subcommand. Can be enabled/disabled here.

2.2.3 General: `Result_folder`

The extension automatically provides results in the current folder, unless otherwise specified. Default can be set in the `settings.json` provided here.

2.2.4 Plotting: `Legend`

The `Built-in plotting` method provides an interactive legend. Can be enabled/disabled here.

2.2.5 Plotting: `Menus`

The `Built-in plotting` method provides a menu with various options. Can be enabled/disabled here.

2.2.6 Plotting: `Max_series_count`

Set the maximum number of time series to plot using the `Built-in plotting` method. Default 100; can be set here.

2.3 For Developers

2.3.1 Debug mode

Please note that this section is only if you can't use the method described in *Installation* or are a developer working on the extension. There are two other ways to use the extension:

- Cloning the repository and placing it under your VSCode extensions folder
- Cloning the repository and using the extension in debug mode. This is only really useful for development purposes, most users won't need this.

To use the extension in debug mode:

1. Download and install VS Code from https://code.visualstudio.com
2. Open VS Code and open a new terminal
   - Terminal -> New Terminal, or
   - `CTRL/CMD + ``
3. In the terminal, run this line: `git clone https://github.com/RuleWorld/BNG_vscode_extension.git` to clone the repository in the desired directory
4. File -> Open to open the repository folder (BNG_vscode_extension)
5. To open up a new window running the extension
   - Run -> Start Debugging, or
   - `F5`

6. Open an existing `.bngl` file or create a new file with `.bngl` extension

CHAPTER

THREE

PYBIONETGEN

3.1 A lightweight BioNetGen CLI

PyBioNetGen is a lightweight command line interface (CLI) for BioNetGen. PyBioNetGen comes with a command line entry point as well as a library with useful functions. Please see *Quickstart* to learn how to install and use PyBioNetGen.

3.1.1 Quickstart

Installation

See *Installation* to learn how to get PyBioNetGen.

Basic Usage

PyBioNetGen's CLI can be used to simply run a BNGL model

```
bionetgen run -i mymodel.bngl -o output_folder
```

which will create `output_folder` and run `mymodel.bngl` inside that folder. For other subcommands or more information on how to use PyBioNetGen's CLI, please see *Command Line Interface*.

PyBioNetGen's library can also be used to run a BNGL model

```
import bionetgen
result = bionetgen.run("mymodel.bngl", output="myfolder")
```

which will create numpy record arrays. For other methods or more information on how to use PyBioNetGen's library, please see *Library*.

3.1.2 Tutorials

Various tutorials for using PyBioNetGen can be found here.

Sample Model

These tutorials use a simple BNGL model as an example. The `SIR.bngl` file can be found `here`.

CLI Tutorial

This tutorial shows how to use PyBioNetGen's command line interface to run and plot a simple BNGL model, as well as how to create a simple Jupyter notebook containing the model, using the simple "SIR.bngl" model as an example.

Getting Started

Make sure you have PyBioNetGen properly installed by running

```
bionetgen -h
```

If this command doesn't print out help information, install PyBioNetGen with

```
pip install bionetgen
```

Running a Model

To first run your model in a new or existing folder called "SIR_folder", use the `run` subcommand:

```
bionetgen run -i SIR.bngl -o SIR_folder
```

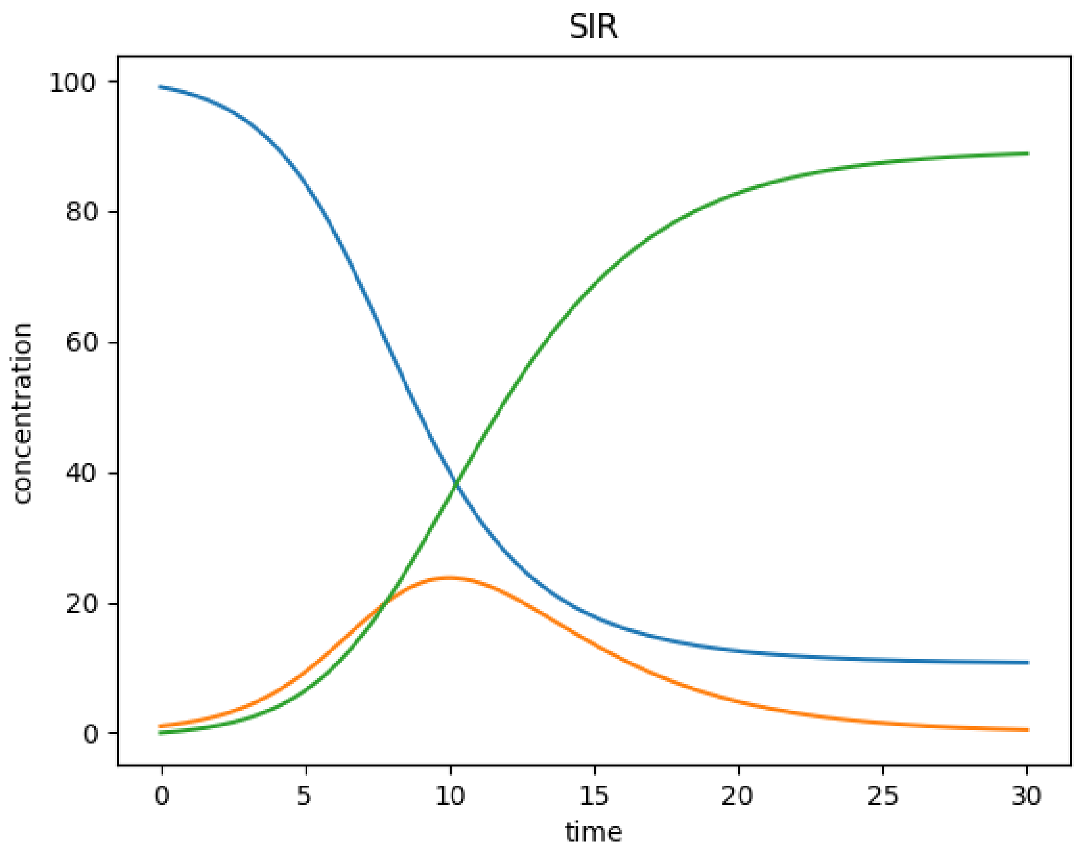
This will run the model and save the results under the specified folder, allowing for further analysis.

Plotting a Model

To simply plot the gdat or cdat file, use the `plot` subcommand with the appropriate file:

```
bionetgen plot -i SIR.gdat
```

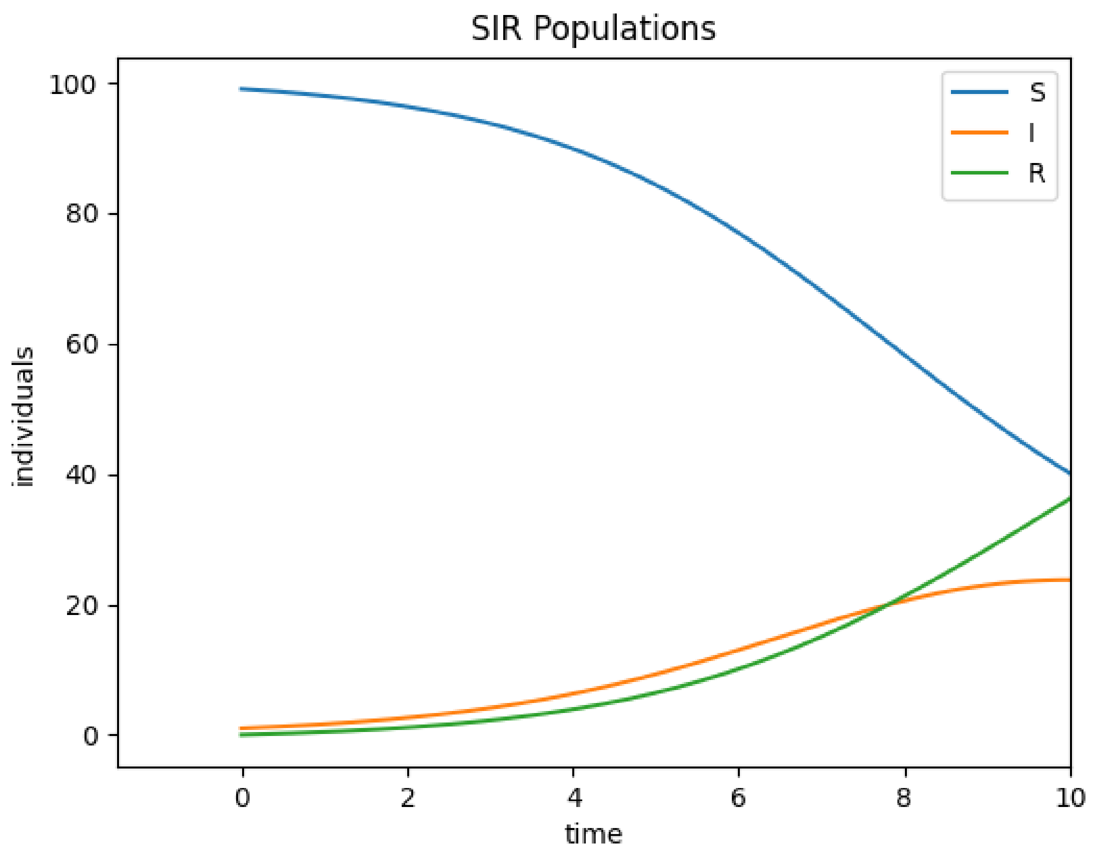
The resulting gdat plot should look like this:



However, there are many optional arguments, such as including a legend or changing axes. Use `bionetgen plot -h` to see them. For example, if we wanted to look at a smaller timeframe, change some labels, and include a legend, we would run:

```
bionetgen plot -i SIR.gdat --legend --xmax 10 --ylabel "individuals" --title "SIR␣
↪Populations"
```

The updated plot should look like this:



Creating a Notebook

Finally, use the `notebook` subcommand to create a Jupyter notebook:

```
bionetgen notebook -i SIR.bngl -o SIR_notebook.ipynb
```

This subcommand currently has limited functionality, and will only create a simple notebook that can run and plot the model.

Library Tutorial

These tutorials shows how to use PyBioNetGen's library to run and plot a simple model, as well as create a BNG model object.

Getting Started

Make sure you have PyBioNetGen properly installed by running

```
bionetgen -h
```

If this command doesn't print out help information, install PyBioNetGen with

```
pip install bionetgen
```

Finally, make sure to import the PyBioNetGen library.

```python
import bionetgen
```

Running and Plotting a Model

Use the `run` method to run a BNGL model. Optionally, you can save the results in a new or existing folder.

```python
result = bionetgen.run("SIR.bngl")
# OR
result = bionetgen.run("SIR.bngl", out = "SIR_folder")
```

To view the resulting gdat record array, you can either use the `gdats` attribute or the index of the `result` object:

```python
result.gdats["SIR"][:10]
# OR
result[0][:10]
```

Similarly, to view the resulting cdat record array, use the `cdats` attribute:

```python
result.cdats["SIR"][:10]
```

To plot the gdat record array, we'll need matplotlib.

```python
import matplotlib.pyplot as plt
```

Save the gdat record array as its own object. Then, the values can be plotted.

```python
r = result[0]

for name in r.dtype.names:
    if name != "time":
        plt.plot(r['time'], r[name], label = name)
plt.xlabel("time")
plt.ylabel("species (counts)")
_ = plt.legend(frameon = False)
```

Using the bngmodel object

Use the `bngmodel` method to create a Python representation of a BNGL model.

```
model = bionetgen.bngmodel("SIR.bngl")
```

To view the model, you can `print()` the entire BNGL model or just certain blocks of the model.

```
print(model)
```

```
print(model.parameters)
```

Jupyter Notebooks

Interactive Jupyter notebooks versions of these tutorials can be found here:

- `Running and Plotting`
- `bngmodel`

Atomizer

Atomizer is a translator tool that can convert SBML models to BNGL models. Moreover, atomizer is capable of extracting implicit information found in SBML models using lexical analysis, reaction stoichiometry and annotation information. Using this information, atomizer converts SBML species to structured BNGL complexes.

This in turn allows for easier identification of modification sites, binding interactions while making the model easier to understand and analyze. Combined with BNGL visualization capabilities, atomizer also allows for easier model comparison of the same biological process.

Getting Started

Make sure you have PyBioNetGen properly installed by running

```
bionetgen -h
```

If this command doesn't print out help information, install PyBioNetGen with

```
pip install bionetgen
```

Basics of atomizing a model

Use the `atomize` method to convert a SBML model to BNGL directly (flat translation)

```
bionetgen atomize -i mymodel.xml -o mymodel_flat.bngl
```

Using the *-a* option allows for atomization of the model (atomized translation)

```
bionetgen atomize -i mymodel.xml -o mymodel_flat.bngl -a
```

if the atomization is taking a long time, try the *-mr* option (please note that this might use up a lot of memory)

```
bionetgen atomize -i mymodel.xml -o mymodel_flat.bngl -a -mr
```

If you encounter atomization errors, you can fix it by using a user input JSON file which is given by the *-u* option (see below)

```
bionetgen atomize -i mymodel.xml -o mymodel_flat.bngl -a -u user_input.json
```

Atomizer looks up annotation information on various online resources (namely Pathway Commons, BioGRID and UniProt). This generally allows for easier atomization, reducing user input required. If you don't have internet connection you can turn this off with the *-p* option

```
bionetgen atomize -i mymodel.xml -o mymodel_flat.bngl -a -p
```

Generally a complex model will require several options with a bit of user input in JSON format (see below), for example

```
bionetgen atomize -i mymodel.xml -o mymodel_flat.bngl -a -u user_input.json
```

If the atomizer output is too cluttered you can adjust the output levels with the *-ll* option

```
bionetgen atomize -i mymodel.xml -o mymodel_flat.bngl -a -u user_input.json -ll "ERROR"
```

we suggest using "ERROR" or "WARNING" for *-ll* argument.

User input format

The user input JSON file has 4 potential fields. Empty fields can be omitted.

```
{
    "reactionDefinition" : [
    ],
    "partialComplexDefinition" : [
    ],
    "binding_interactions" : [
        ["Partner1", "Partner2"]
    ],
    "modificationDefinition": {
        "complex":["molecule1", "molecule2"],
    }
}
```

"binding_interactions" field is an array where each element is also an array of two items. Both items should be the names of species in the model, exactly as written in the SBML. This represents that there is a binding interaction between the two items which in turn tells atomizer that there should be a binding component on both molecules for each other.

"modificationDefinition" field is a dictionary where the key is a complex in the model and the value is an array that reflects what the complex is comprised of.

"reactionDefinition" field is…

"partialComplexDefinition" field is…

Examples of atomization

These tutorials shows how to use Atomization tool to make a BNGL Model from SBML format. We will Use various curated models from the biomodels database. You can download the specific SBML files by clicking on the titles.

Biomodels database model 48

Atomizing the Model: Once you download the SBML file of BMD48, you will have an `.xml` file in your directory. Use it as the input to the *atomize* subcommand as shown below. To show the effect of using the web services we'll also add the *-p* option to not use the web serices at first

```
bionetgen atomize -i BIOMD0000000048.xml -o BMD48.bngl -a -p
```

you can name the *bngl* output file whatever you want. This will print out information on the atomization process. If the output is too cluttered you can look at only the major errors using the following command

```
bionetgen atomize -i BIOMD0000000048.xml -o BMD48.bngl -a -p -ll "ERROR"
```

which prints out

```
ERROR:SCT212:['EGF_EGFR2']:EGF_EGFR2_P:Atomizer needs user information to determine␣
↪which element is being modified among component species:['EGF', 'EGF', 'EGFR', 'EGFR
↪']:_p
ERROR:ATO202:['EGF_EGFR2', 'EGF_EGFR2_PLCg_P']:(('EGF', 'EGF'), ('EGF', 'EGFR'), ('EGFR',
↪ 'EGFR')):We need information to resolve the bond structure of these complexes .␣
↪Please choose among the possible binding candidates that had the most observed␣
↪frequency in the reaction network or provide a new one
ERROR:ATO202:['EGF_EGFR2_Shc_Grb2_SOS']:(('EGF', 'Grb2'), ('EGF', 'SOS'), ('EGF', 'Shc'),
↪ ('EGFR', 'Grb2'), ('EGFR', 'SOS'), ('EGFR', 'Shc')):We need information to resolve␣
↪the bond structure of these complexes . Please choose among the possible binding␣
↪candidates that had the most observed frequency in the reaction network or provide a␣
↪new one
Structured molecule type ratio: 0.7
```

the first three "ERROR"s tells us that atomizer needs user input to resolve certain ambiguities in the model. Structured molecule type ratio is the ratio of structured species in the *molecule types* block of the resulting BNGL to the total number of molecule types, to give an idea of how successful atomizer was at inferring structure of the species in the model.

Before we give atomizer more user input, let's try removing the *-p* option to see if atomizer can resolve these automatically

```
bionetgen atomize -i BIOMD0000000048.xml -o BMD48.bngl -a -ll "ERROR"
```

which prints out

```
ERROR:SCT212:['EGF_EGFR2']:EGF_EGFR2_P:Atomizer needs user information to determine␣
↪which element is being modified among component species:['EGF', 'EGF', 'EGFR', 'EGFR
↪']:_p
ERROR:ATO202:['EGF_EGFR2_PLCg_P']:(('EGF', 'PLCg'), ('EGFR', 'PLCg')):We need␣
↪information to resolve the bond structure of these complexes . Please choose among the␣
↪possible binding candidates that had the most observed frequency in the reaction␣
↪network or provide a new one
Structured molecule type ratio: 0.875
```

there were multiple instances of “ERROR:MSC02” that warn the user about issues with connections to the BioGRID service which were removed for clarity. Now we only have two errors left.

Resolving errors

Now let’s take a look at the remaining issues one by one

```
ERROR:SCT212:['EGF_EGFR2']:EGF_EGFR2_P:Atomizer needs user information to determine␣
↪which element is being modified among component species:['EGF', 'EGF', 'EGFR', 'EGFR
↪']:_p
```

atomizer is having trouble figuring out where the modification *_p* is supposed to go, which is a phosphorylation site. We know that EGFR is the molecule that’s being phosphorylated so we make a JSON file (here we call it *user-input_1.json*)

```
{
      "modificationDefinition": {
              "EGF_EGFR2_P": ["EGFR_P", "EGFR", "Epidermal_Growth_Factor", "Epidermal_
↪Growth_Factor"]
      }
}
```

and we rerun atomization with the *-u* option using this JSON file we created

```
bionetgen atomize -i BIOMD0000000048.xml -o BMD48.bngl -a -ll "ERROR" -u user-input-1.
↪json
```

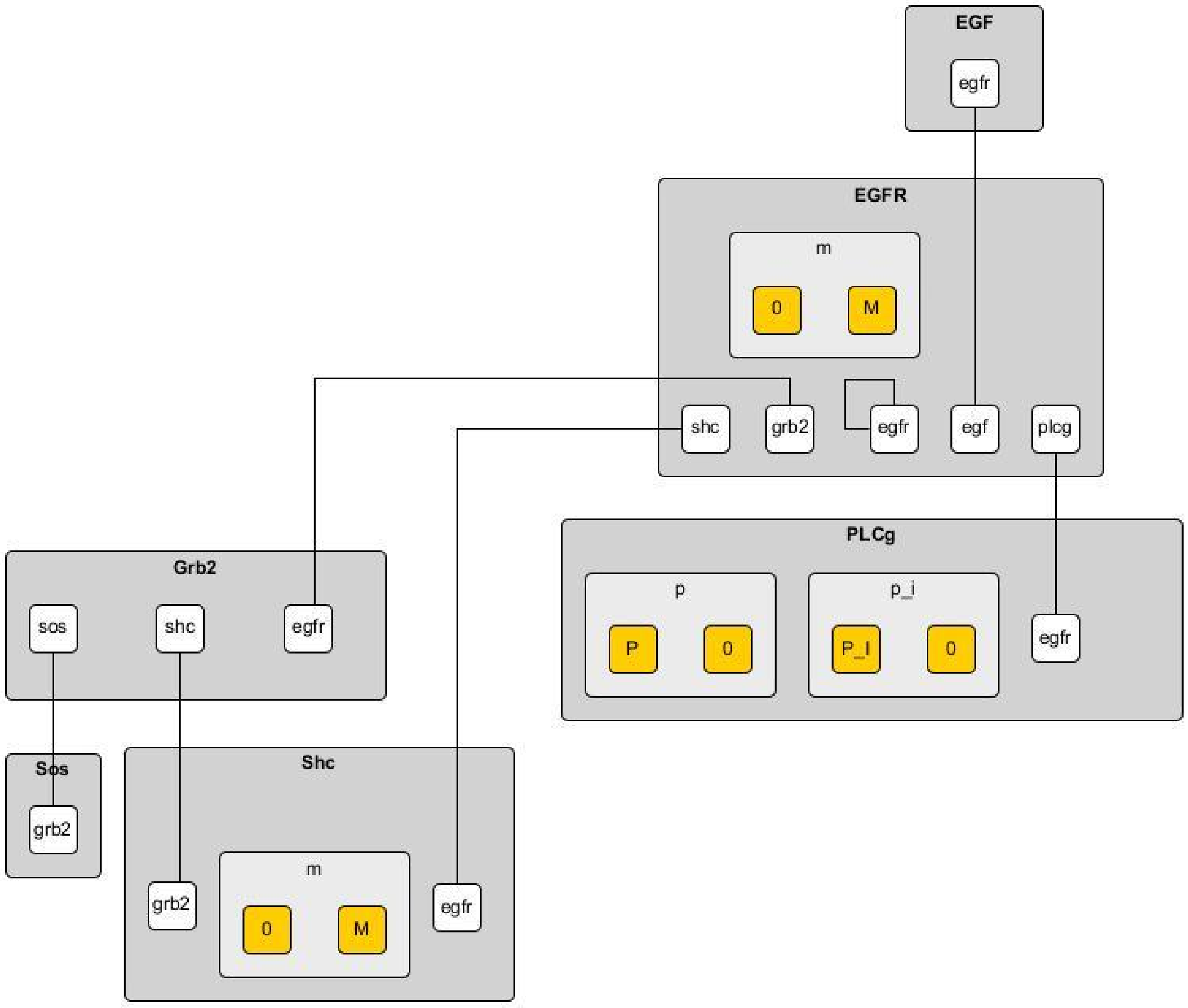
which returns (disregarding connection errors)

```
ERROR:ATO202:['EGF_EGFR2_PLCg', 'EGF_EGFR2_PLCg_P']:(('EGFR', 'PLCg'), ('Epidermal_
↪Growth_Factor', 'PLCg')):We need information to resolve the bond structure of these␣
↪complexes . Please choose among the possible binding candidates that had the most␣
↪observed frequency in the reaction network or provide a new one
```

which tells us that atomizer can’t resolve where *PLCg* is binding, let’s add that to the JSON file

```
{
  "binding_interactions": [
      ["EGFR", "PLCg"]
  ],
      "modificationDefinition": {
              "EGF_EGFR2_P": ["EGFR_P", "EGFR", "Epidermal_Growth_Factor", "Epidermal_
↪Growth_Factor"]
      }
}
```

rerunning atomizer should return no errors and you should now have a fully atomized BNGL model. usage and using yEd to look at the contact map gives us the following



Biomodels database model 19

This model is an expanded version of BioModel 48. Let's follow the same strategy and atomize it without any input first and see what atomizer says.

```
bionetgen atomize -i BIOMD0000000019.xml -o BMD19.bngl -a -ll "ERROR"
```

this returns

```
ERROR:ATO202:['EGF_EGFRm2_GAP_Grb2_Prot', 'EGF_EGFRm2_GAP_Grb2_Sos_Prot',
              'EGF_EGFRm2_GAP_Shcm_Grb2_Prot', 'EGF_EGFRm2_GAP_Grb2_Sos_Ras_GTP_Prot',
              'EGF_EGFRm2_GAP_Shcm_Grb2_Sos_Prot', 'EGF_EGFRm2_GAP_Grb2_Sos_Ras_GDP_Prot
↪',
              'EGF_EGFRm2_GAP_Shcm_Grb2_Sos_Ras_GTP_Prot',
              'EGF_EGFRm2_GAP_Shcm_Grb2_Sos_Ras_GDP_Prot']:
              (('EGF', 'Prot'), ('EGFR', 'Prot'), ('GAP', 'Prot'),
              ('Grb2', 'Prot')):We need information to resolve the bond structure of␣
↪these
```

(continues on next page)

(continued from previous page)

```
              complexes . Please choose among the possible binding candidates that had␣
↪the
              most observed frequency in the reaction network or provide a new one
Structured molecule type ratio: 0.6363636363636364
```

which tells us that atomizer is having trouble figuring out which binding interaction to include for *Prot*. We know that protein binds to EGFR so let's include that in a user input JSON file

```
{
    "binding_interactions": [
        ["EGFR", "Prot"]
    ]
}
```

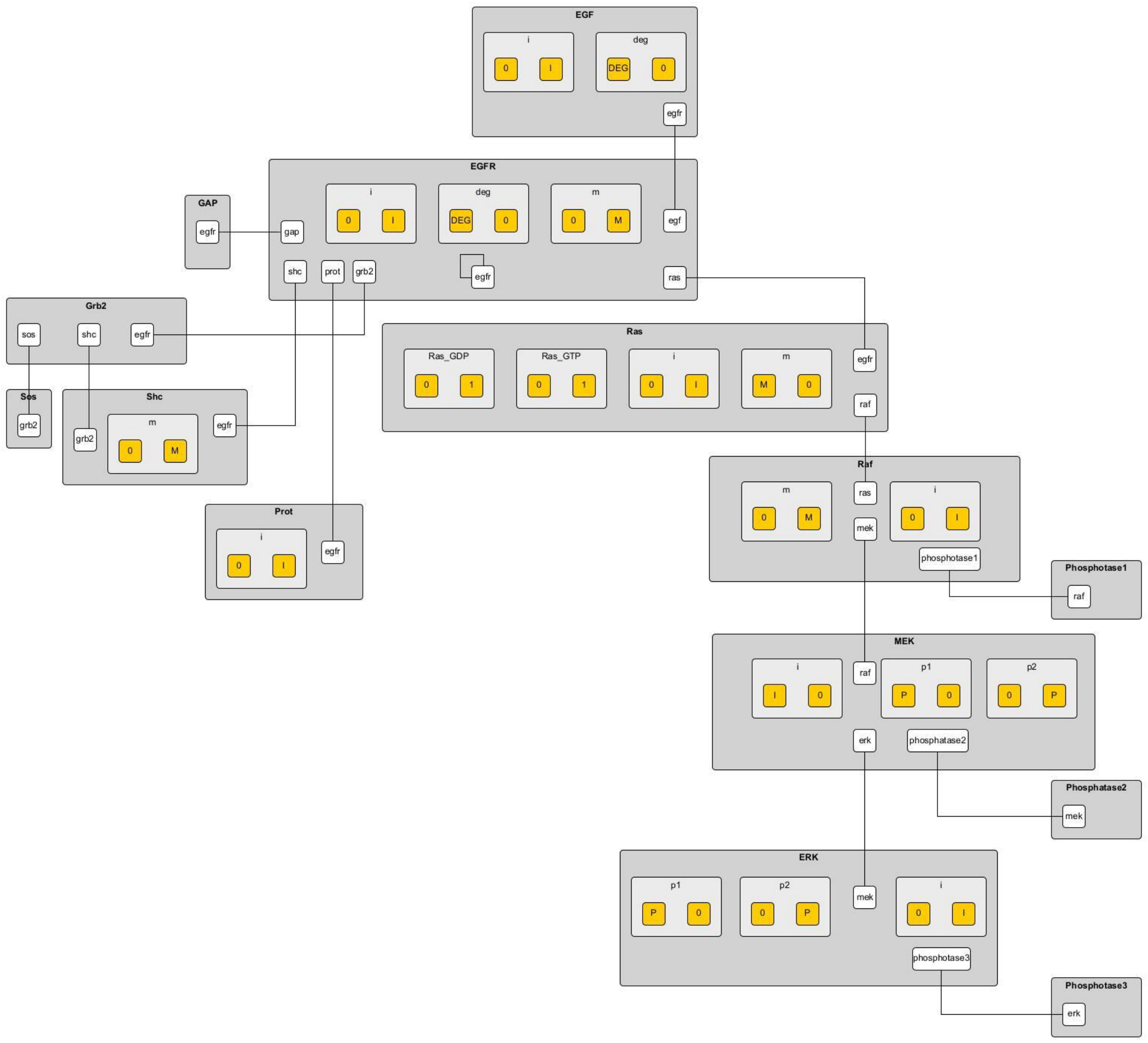
and now rerunning the atomization using this user input file

```
bionetgen atomize -i BIOMD0000000019.xml -o BMD19.bngl -a -ll "ERROR" -u user-input-1.
↪json
```

now returns no errors. However, looking at the resuling BNGL shows *Ras_GTP(Ras_GDP~0~1,egfr,i~0~I,m~0~M,raf)* and we know that Ras should be the base species. We can include that using the *modificationDefinition* section in the user input file

```
{
    "binding_interactions": [
        ["EGFR", "Prot"]
    ],
    "modificationDefinition": {
        "Ras": [],
        "Ras_GTP": ["Ras"],
        "Ras_GDP": ["Ras"]
    }
}
```

rerunning atomization using this user input gives a fully atomized BNGL file. usage and using yEd to look at the contact map gives us the following



Biomodels database model 151

Running atomizer on this model with the following

```
bionetgen atomize -i bmd0000000151.xml -o bmd151.bngl -a -ll "ERROR"
```

we get the following errors

```
ERROR:ANN202:Ras_GTP:Rafast:can be mapped through naming conventions but the annotation␣
↪information does not match
ERROR:ANN202:Ras_GDP:Rafast:can be mapped through naming conventions but the annotation␣
↪information does not match
ERROR:SCT211:IL6_gp80_gp130_JAKast2_STAT3C_SOCS3_SHP2:[['IL6_gp80', 'IL6_gp80', 'gp130_
↪JAK_ast', 'gp130_JAK', 'SHP2', 'SOC
    S3', 'STAT3C'], ['IL6_gp80_gp130_JAKast2_STAT3C_SHP2', 'SOCS3']]:[['IL6_gp80', 'IL6_
↪gp80', 'JAK', 'JAK', 'SHP2', 'SOCS3',
```

(continues on next page)

(continued from previous page)

```
    'STAT3'], ['IL6_gp80', 'SOCS3']]:Cannot converge to solution, conflicting definitions
ERROR:SCT211:IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GDP:[['Grb2', 'IL6_gp80', 'IL6_
↪gp80', 'gp130_JAK_ast', 'gp130_JAK
    ', 'Ras_GDP', 'SHP2ast', 'SOS'], ['Grb2', 'IL6_gp80', 'IL6_gp80', 'gp130_JAK_ast',
↪'gp130_JAK', 'Ras_GTP', 'SHP2ast', 'SOS
    ']]:[['Grb2', 'IL6_gp80', 'IL6_gp80', 'JAK', 'JAK', 'Ras_GDP', 'SHP2', 'SOS'], ['Grb2
↪', 'IL6_gp80', 'IL6_gp80', 'JAK', 'JA
    K', 'Ras_GTP', 'SHP2', 'SOS']]:Cannot converge to solution, conflicting definitions
ERROR:SCT211:IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GTP:[['Grb2', 'IL6_gp80', 'IL6_
↪gp80', 'gp130_JAK_ast', 'gp130_JAK
    ', 'Ras_GDP', 'SHP2ast', 'SOS'], ['Grb2', 'IL6_gp80', 'IL6_gp80', 'gp130_JAK_ast',
↪'gp130_JAK', 'Ras_GTPast', 'SHP2ast', '
    SOS']]:[['Grb2', 'IL6_gp80', 'IL6_gp80', 'JAK', 'JAK', 'Ras_GDP', 'SHP2', 'SOS'], [
↪'Grb2', 'IL6_gp80', 'IL6_gp80', 'JAK',
    'JAK', 'Ras_GTP', 'SHP2', 'SOS']]:Cannot converge to solution, conflicting␣
↪definitions
ERROR:ATO202:['IL6_gp80_gp130_JAK2', 'IL6_gp80_gp130_JAK_ast2', 'IL6_gp80_gp130_JAKast2_
↪JAK', 'IL6_gp80_gp130_JAKast2_SOCS
    3', 'IL6_gp80_gp130_JAKast2_STAT3C', 'IL6_gp80_gp130_JAKast2_SHP2ast', 'IL6_gp80_
↪gp130_JAKast2_STAT3Cast', 'IL6_gp80_gp130
    _JAKast2_SHP2ast_Grb2', 'IL6_gp80_gp130_JAKast2_STAT3C_SOCS3', 'IL6_gp80_gp130_
↪JAKast2_SHP2_Grb2']:(('IL6_gp80', 'IL6_gp80
    '), ('IL6_gp80', 'JAK'), ('JAK', 'JAK')):We need information to resolve the bond␣
↪structure of these complexes . Please cho
    ose among the possible binding candidates that had the most observed frequency in␣
↪the reaction network or provide a new one
```

first two errors show that atomizer is having problems identifying *Ras_GTP*, *Ras_GDP* as *Ras*. We can tell atomizer the basic building blocks for the atomization in the *modificationDefinition* block

```
"modificationDefinition": {
    "Ras": [],
    "Ras_GTP": ["Ras"],
    "Ras_GDP": ["Ras"]
}
```

looking at the other errors we can tall atomizer also is having trouble identifing the binding between *IL6* and *gp80* since it's asking about binding interactions between *IL6_gp80* and other items, let's fix that too

```
{
    "modificationDefinition": {
    "Ras": [],
    "Ras_GTP": ["Ras"],
    "Ras_GDP": ["Ras"],
    "IL6_gp80": ["IL6", "gp80"]
    }
}
```

Rerunning atomizer with this user input using the following

```
bionetgen atomize -i bmd0000000151.xml -o bmd151.bngl -a -ll "ERROR" -u user_input_151.
↪json
```

gives the following new set of errors

```
ERROR:ATO202:['IL6_gp80_gp130_JAK', 'IL6_gp80_gp130_JAK2', 'IL6_gp80_gp130_JAK_ast2']:
    (('IL6', 'JAK'), ('JAK', 'gp80')):We need information to resolve the bond structure
    of these complexes . Please choose among the possible binding candidates that had the
    most observed frequency in the reaction network or provide a new one
ERROR:ATO202:['IL6_gp80_gp130_JAKast2_JAK', 'IL6_gp80_gp130_JAKast2_SHP2ast',
    'IL6_gp80_gp130_JAKast2_STAT3C_SHP2', 'IL6_gp80_gp130_JAKast2_SHP2ast_Grb2',
    'IL6_gp80_gp130_JAKast2_SHP2_Grb2', 'IL6_gp80_gp130_JAKast2_STAT3C_SOCS3_SHP2',
    'IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GDP',
    'IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GTP']:
    (('IL6', 'SHP2'), ('SHP2', 'gp80')):We need information to resolve the bond structure
    of these complexes . Please choose among the possible binding candidates that had the
    most observed frequency in the reaction network or provide a new one
ERROR:ATO202:['IL6_gp80_gp130_JAKast2_SOCS3', 'IL6_gp80_gp130_JAKast2_STAT3C_SOCS3']:
    (('IL6', 'SOCS3'), ('SOCS3', 'gp80')):We need information to resolve the bond␣
↪structure of
    these complexes . Please choose among the possible binding candidates that had the␣
↪most
    observed frequency in the reaction network or provide a new one
ERROR:ATO202:['IL6_gp80_gp130_JAKast2_STAT3C']:(('IL6', 'STAT3C'), ('STAT3C', 'gp80')):
    We need information to resolve the bond structure of these complexes . Please choose␣
↪among
    the possible binding candidates that had the most observed frequency in the reaction␣
↪network
    or provide a new one
ERROR:ATO202:['IL6_gp80_gp130_JAKast2_STAT3Cast']:(('IL6', 'STAT3Cast'), ('STAT3Cast',
↪'gp80')):
    We need information to resolve the bond structure of these complexes . Please choose␣
↪among
    the possible binding candidates that had the most observed frequency in the reaction␣
↪network
    or provide a new one
```

unfortunately, we know from the model that these are not the correct binding interactions. *JAK* binds to *gp130* and so does *SHP2*. Let's add those in

```
{
    "binding_interactions": [
        ["gp130", "SHP2"],
        ["gp130", "JAK"]
    ],
    "modificationDefinition": {
    "Ras": [],
    "Ras_GTP": ["Ras"],
    "Ras_GDP": ["Ras"],
    "IL6_gp80": ["IL6", "gp80"]
    }
}
```

which gives

```
ERROR:SCT241:IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GDP:
    IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GTP:produce the same translation:
```

(continues on next page)

(continued from previous page)

```
    ['Grb2', 'IL6', 'IL6', 'JAK', 'JAK_ast', 'Ras_GDP', 'SHP2ast', 'SOS', 'gp130', 'gp130
↪',
    'gp80', 'gp80']:IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GTP:was emptied
```

We can tell atomizer the composition of *IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GTP* with

```
{
    "binding_interactions": [
        ["gp130", "SHP2"],
        ["gp130", "JAK"]
    ],
    "modificationDefinition": {
    "Ras": [],
    "Ras_GTP": ["Ras"],
    "Ras_GDP": ["Ras"],
    "IL6_gp80": ["IL6", "gp80"],
    "IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GTP": ["IL6","gp80","gp130","JAK","IL6",
↪"gp80","gp130","JAK","SHP2","Grb2","SOS","Ras"]
    }
}
```

which atomizer without errors. Looking at BNGL, we can see that atomizer misses the interaction between SOS and Ras and instead binds SOS to Grb2, we can fix that with this final JSON addition

```
{
    "binding_interactions": [
        ["gp80", "gp80"],
        ["gp130", "SHP2"],
        ["SOS", "Ras"]
    ],
    "modificationDefinition": {
    "Ras": [],
    "Ras_GTP": ["Ras"],
    "Ras_GDP": ["Ras"],
    "IL6_gp80": ["IL6", "gp80"],
    "IL6_gp80_gp130_JAKast2_SHP2ast_Grb2_SOS_Ras_GTP": ["IL6","gp80","gp130","JAK","IL6",
↪"gp80","gp130","JAK","SHP2","Grb2","SOS","Ras"]
    }
}
```

which gives us a fully atomized model.

Biomodels database model 543

Model 543 is an expanded version of model 151, please look at BMD151 tutorial before starting here. We will start from a subset of the the final user input JSON file we constructed for BMD151.

We can then use the following command to start atomizing with this user input file (which we named *user_input_543*)

```
bionetgen atomize -i bmd0000000543.xml -o bmd543.bngl -a -ll "ERROR" -u user_input_543.
↪json
```

which gives us the following set of errors and a reasonably well atomized model

```
ERROR:ATO202:['IFN_R_JAK2m_SHP2']:(('IFN', 'SHP2'), ('JAK', 'SHP2'), ('R', 'SHP2')):
    We need information to resolve the bond structure of these complexes .
    Please choose among the possible binding candidates that had the most observed
    frequency in the reaction network or provide a new one
ERROR:ATO202:['IFN_R_JAK2m_STAT3C']:(('IFN', 'STAT3C'), ('JAK', 'STAT3C'), ('R', 'STAT3C
↪')):
    We need information to resolve the bond structure of these complexes . Please choose␣
↪among
    the possible binding candidates that had the most observed frequency in the reaction␣
↪network
    or provide a new one
ERROR:ATO202:['IFN_R_JAK2m_STAT3Cm']:(('IFN', 'STAT3Cm'), ('JAK', 'STAT3Cm'), ('R',
↪'STAT3Cm')):
    We need information to resolve the bond structure of these complexes .
    Please choose among the possible binding candidates that had the most observed␣
↪frequency
    in the reaction network or provide a new one
ERROR:SCT241:IL6_gp80_gp130_JAK2m_STAT1:IL6_gp80_gp130_JAK2m_STAT1C:produce the same␣
↪translation:
    ['IL6', 'IL6', 'JAKIL_6', 'JAKIL_6', 'STAT1C', 'gp130', 'gp130m', 'gp80', 'gp80']:
    IL6_gp80_gp130_JAK2m_STAT1C:was emptied
ERROR:SCT241:IFN_R_JAK:IFN_R_JAK2:produce the same translation:['IFN', 'JAK', 'R']:
    IFN_R_JAK2:was emptied
```

for the first two errors, we add binding interactions [“R”, “SHP2”] and [“R”, “STAT3”]. The third error we notice that atomizer thinks *STAT3C* and *STAT3* are different, we’ll address that by adding `modificationDefinition`shell for STAT species and get the following input file

```
{
    "binding_interactions": [
        ["gp80", "gp80"],
        ["gp130", "SHP2"],
        ["SOS", "Ras"],
        ["R","SHP2"],
        ["R","STAT3"]
    ],
    "modificationDefinition": {
    "Ras": [],
    "Ras_GTP": ["Ras"],
    "Ras_GDP": ["Ras"],
    "STAT3": [],
    "STAT3m": ["STAT3"],
    "STAT3C": ["STAT3"],
    "STAT3Cm": ["STAT3"],
    "STAT1": [],
    "STAT1m": ["STAT1"],
    "STAT1C": ["STAT1"],
    "STAT1Cm": ["STAT1"]
    }
}
```

atomizing again we get the following errors

```
ERROR:ANN202:SOCS1:SOCS3:can be mapped through naming conventions but the annotation␣
↪information
    does not match
ERROR:ANN202:Phosp1:Phosp3:can be mapped through naming conventions but the annotation␣
↪information
    does not match
ERROR:SCT241:IFN_R_JAK:IFN_R_JAK2:produce the same translation:['IFN', 'JAK', 'R']:IFN_R_
↪JAK2:
    was emptied
ERROR:SCT241:IL6_gp80_gp130_JAK2m_STAT1:IL6_gp80_gp130_JAK2m_STAT1C:produce the same␣
↪translation:
    ['IL6', 'IL6', 'JAKIL_6', 'JAKIL_6', 'STAT1C', 'gp130', 'gp130m', 'gp80', 'gp80']:
    IL6_gp80_gp130_JAK2m_STAT1C:was emptied
```

looking at the 3rd and 4th errors, combined with the previous errors, we can see that atomizer is having trouble resolving types of JAK molecules. We can add `modificationDefinition`s to help with those and some downstream products of JAK

```
"JAK": [],
"JAKIFN": ["JAK"],
"JAKIL_6":["JAK"]
"R_JAK": ["R","JAKIFN"],
"IFN_R_JAK": ["IFN","R_JAK"],
"IFN_R_JAK2": ["IFN_R_JAK", "IFN_R_JAK"],
"IFN_R_JAK2m": ["IFN_R_JAK2"],
"gp130_JAK": ["gp130", "JAKIL_6"]
```

adding these to the user input file and rerunning atomization gives

```
ERROR:ANN202:SOCS1:SOCS3:can be mapped through naming conventions but the annotation␣
↪information
    does not match
ERROR:ANN202:Phosp1:Phosp3:can be mapped through naming conventions but the annotation␣
↪information
    does not match
ERROR:SCT241:IL6_gp80_gp130_JAK2m_STAT1:IL6_gp80_gp130_JAK2m_STAT1C:produce the same␣
↪translation:
    ['IL6', 'IL6', 'JAKIL_6', 'JAKIL_6m', 'STAT1C', 'gp130', 'gp130', 'gp80', 'gp80']:
    IL6_gp80_gp130_JAK2m_STAT1C:was emptied
```

looking at the last error, we should give a *modificationDefinition* for *IL6_gp80_gp130_JAK2m_STAT1C*

```
"IL6_gp80_gp130_JAK2m_STAT1C": ["IL6", "gp80", "gp130", "JAK","IL6", "gp80", "gp130",
↪"JAK","STAT1"]
```

Adding this will result in atomization without any errors. However, looking at BMD543, we find some *binding_interactions* that should be added to BMD543 user input that are missing in BMD151. We add the following set of *binding_interactions*

```
["gp130", "SOCS3"],
["gp130","STAT3"]
["R","STAT1"],
["R","R"],
```

(continues on next page)

(continued from previous page)

```
["R","SOCS1"],
["STAT1","gp130"]
```

to get the final input file

```
{
    "binding_interactions": [
        ["gp80", "gp80"],
        ["SOS", "Ras"],
        ["R","SHP2"],
        ["R","STAT3"],
        ["R","STAT1"],
        ["R","R"],
        ["R","SOCS1"],
        ["gp130", "SHP2"],
        ["STAT1","gp130"],
        ["STAT3","gp130"],
        ["gp130", "SOCS3"]
    ],
    "modificationDefinition": {
    "Ras": [],
    "Ras_GTP": ["Ras"],
    "Ras_GDP": ["Ras"],
    "STAT3": [],
    "STAT3m": ["STAT3"],
    "STAT3C": ["STAT3"],
    "STAT3Cm": ["STAT3"],
    "STAT1": [],
    "STAT1m": ["STAT1"],
    "STAT1C": ["STAT1"],
    "STAT1Cm": ["STAT1"],
    "JAK": [],
    "JAKIFN": ["JAK"],
    "JAKIL_6":["JAK"],
    "R_JAK": ["R","JAKIFN"],
    "gp130_JAK": ["gp130", "JAKIL_6"],
    "IFN_R_JAK2": ["IFN","R_JAK", "IFN","R_JAK"],
    "IFN_R_JAK2m": ["IFN_R_JAK2"],
    "IL6_gp80_gp130_JAK2m_STAT1C": ["IL6", "gp80", "gp130", "JAK","IL6", "gp80", "gp130",
↪ "JAK","STAT1"]
    }
}
```

and running with this user input file gives us the fully atomized model.

3.1.3 Command Line Interface

The command line tool comes with several subcommands. For each command, you can see the help text with the command

```
bionetgen subcommand -h
```

Run

This subcommand simply runs a model:

```
bionetgen run -i mymodel.bngl -o output_folder
```

This will run `mymodel.bngl` under the folder `output_folder`. If no output folder is specified, then the temporary folder used while running the subcommand will be deleted upon completion.

Plot

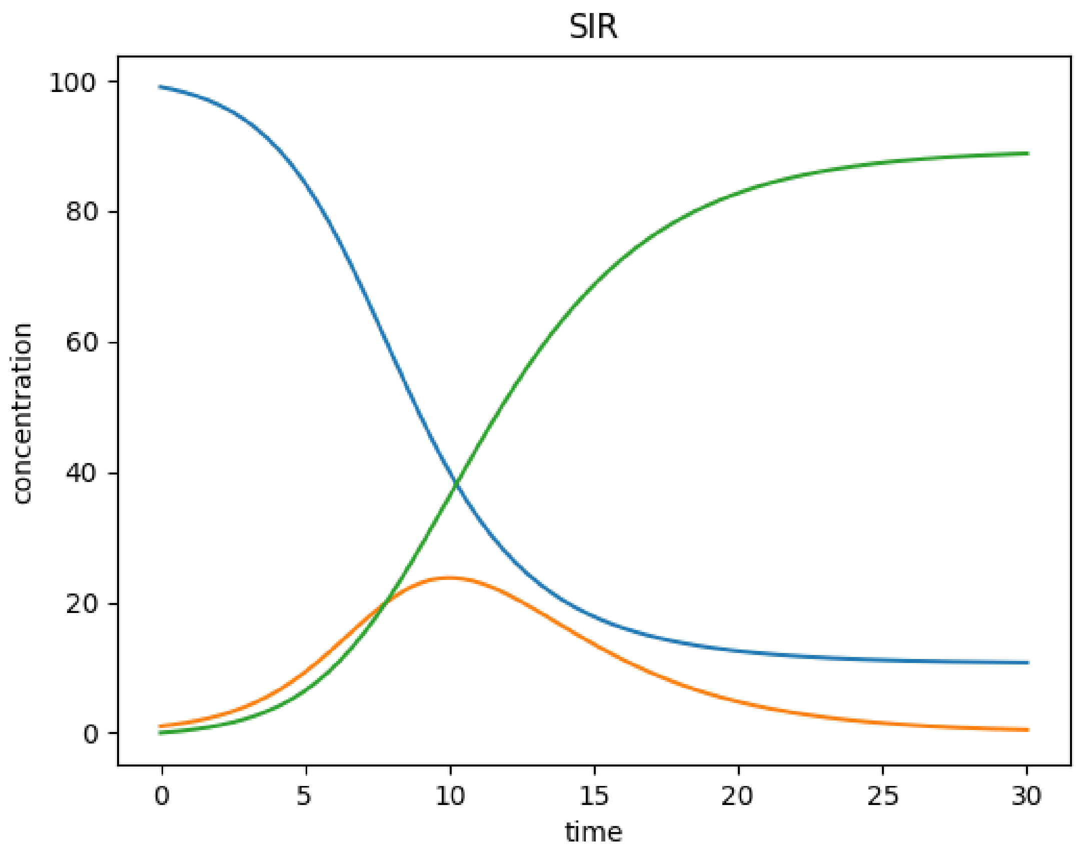
This subcommand allows you to make a simple plot from a gdat/cdat or scan file:

```
bionetgen plot -i mymodel.gdat -o gdat_plot.png
```

You can see all the available options by running `bionetgen plot -h`

```
optional arguments:
   -h, --help            show this help message and exit
   -i INPUT, --input INPUT
                           Path to .gdat/.cdat file to use plot
   -o OUTPUT, --output OUTPUT
                           Optional path for the plot (default:
                           "$model_name.png")
   --legend              To plot the legend or not (default: False)
   --xmin XMIN           x-axis minimum (default: determined from data)
   --xmax XMAX           x-axis maximum (default: determined from data)
   --ymin YMIN           y-axis minimum (default: determined from data)
   --ymax YMAX           y-axis maximum (default: determined from data)
   --xlabel XLABEL       x-axis label (default: time)
   --ylabel YLABEL       y-axis label (default: concentration)
   --title TITLE         title of plot (default: determined from input file)
```

Resulting plots should look similar to this:



Visualize

This subcommand creates .graphml files to be used by an external graph editor (yEd) for model visualization, including contact maps.

```
bionetgen visualize -i mymodel.bngl -o output_folder
```

You can see all the available options by running `bionetgen visualize -h`

```
optional arguments:
   -h, --help            show this help message and exit
-i INPUT, --input INPUT
                        Path to BNGL model to visualize
-o OUTPUT, --output OUTPUT
                        (optional) Output folder, defaults to current folder
-t TYPE, --type TYPE  (optional) Type of visualization requested. Valid options are:
↪'ruleviz_pattern','ruleviz_operation', 'contactmap', 'regulatory' and
                        'atom_rule'. Regulatory and atom rule graphs are the same thing,␣
↪defaults to 'contactmap'.
```

Notebook

This subcommand is in its early stages of development. The subcommand is used to generate a simple Jupyter notebook. You can also give your model as an argument and the resulting notebook will be ready to load in your model using PyBioNetGen library.

```
bionetgen notebook -i mymodel.bngl -o mynotebook.ipynb
```

Info

This subcommand simply prints out information about software versions and installation paths.

```
bionetgen info
```

Atomize

The CLI includes one more subcommand, `atomize`, which is detailed further in *Atomizer*.

Tutorial

For a brief tutorial showing how to use the CLI on a simple BNGL model, please see *CLI Tutorial*.

3.1.4 Library

PyBioNetGen also comes with a library that allows you to programatically run and do simple modifications of BNGL models.

run

This method allows you to do a simple run of a BNGL model and returns the results as numpy record arrays.

```
import bionetgen
result = bionetgen.run("mymodel.bngl", out="myfolder")
result["mymodel"] # this will contain the gdat results of the run
```

bngmodel

This method allows you to load in a model into a python object.

```
import bionetgen
model = bionetgen.bngmodel("mymodel.bngl") # generates BNG-XML and reads it
print(model)
```

Basic Usage

This is designed to be a pythonic object representing the BNGL model given. It currently has some limited options to modify the model. You can load the model object using

```
import bionetgen
model = bionetgen.bngmodel("mymodel.bngl") # generates BNG-XML and reads it
```

The underlying code will attempt to generate a BNG-XML of the model which it then reads to generate this object.

One core principle of this object is that the object and every object associated with it can be converted to a string to get the BNGL string of the object itself. For example

```
print(model) # this prints the entire model
print(model.observables) # prints the observables block
print(model.parameters) # prints the parameters block
model.parameters.k = 10 # sets parameter k to 10
print(model.parameters) # block updated to reflect change
```

The BNGL string is dynamically generated and not just returning the block string from the original model. This allows for making simple changes to your model, e.g.

```
for i in range(10):
    model.parameters.k = i
    with open("param_k_{}.bngl".format(i), "w") as f:
        f.write(str(model))
```

This will write 10 models with different k parameters.

Blocks

All blocks that are active can be seen with print(model.active_blocks). Currently supported blocks are

- Parameters
- Compartments
- Molecule types
- Species (or seed species)
- Observables
- Functions
- Reaction rules

PyBioNetGen bngmodel also recognizes actions within the model but discards them upon loading (this will eventually be optional). All bionetgen features will eventually be supported by this library, including every valid BNGL block.

Blocks also act pythonic and act like other python objects

```
for param in model.parameters:
    print("parameter name: {}".format(param))
    print("parameter value: {}".format(model.parameters[param]))

for obs in model.observables:
    obs_val = model.observables[obs]
```

(continues on next page)

(continued from previous page)

```python
    print("observable name: {}".format(obs))
    print("observable type: {}".format(obs_val[0]))
    print("observable pattern: {}".format(obs_val[1]))

for spec in model.species:
    spec_count = model.species[spec]
    print("species name: {}".format(spec))
    print("species count: {}".format(spec_count))
    print("molecules in species: {}".format(spec.molecules))
```

The following sections will detail how each block behaves

Parameters

Parameters are a list of names and values associated with those names. The parameters block also stores the parameter expressions in case they are written as functions in the original model.

```python
# let's say we have a parameter k
model.parameters["k"] = 10 # this is the parameter value
model.parameters.k = 10 # this is also the parameter value
model.parameters.expressions["k"] # this is the parameter expression
```

Compartments

Compartments are comprised of a compartment name, dimensionality, volume, and an optional parent compartment name

```python
# say we have a compartment string "PM 2 10.0 EC"
# which is a 2 dimensional compartment with 2 dimensions and 10 volume
# and is contained under another compartment EC
comp_name = model.compartments[i] # where i is the index of PM compartment, will return
↪"PM"
comp_list = model.compartments[comp_name] # will return [2, 10.0, "EC"]
print(comp_list[0]) # will print 2
print(comp_list[1]) # will print 10.0
print(comp_list[2]) # will print EC
```

Molecule types

Molecule types contains different components and all possible states of those components

```python
# let's say we have a molecule type "X()" as the first one
X_obj = model.molecule_types[0] # this is the object for "X()" molecule type
print(X_obj) # will print the molecule type string
X_obj.add_component("p", states=["0","1"]) # adds a component with states
print(X_obj) # prints "X(p~0~1)" now
```

Species

Species are made up of molecules and can contain an overall compartment and label.

```
# let's say we have a species with pattern "X()"
species_obj = model.species[0] # this is the species object
print(species_obj) # prints the pattern
count = model.species[species_obj] # this is the starting count of the species
count = model.species["X()"] # this is the starting count of the species
molecules = species_obj.molecules # this is the list of molecules in the pattern
compartment = species_obj.compartment # this is the overall compartment of the species
label = species_obj.label # this is the overall label of the species
```

Observables

Observables are made up of a list of species patterns

```
# let's say we have a observable with string "Molecules X_phos X(p~1)"
obs_obj = model.observables[0] # this is the observable object
print(obs_obj) # prints the observable patterns, in this case X(p~1)
obs_list = model.observables["X_phos"] # this returns a list of two items
type, obs_obj = obs_list # first one is the type, "Molecules" and second one is the␣
↪object again
patterns = obs_obj.patterns # this is the list of patterns in the observable string
```

Functions

Functions are just a tuple of function name and expression

```
# say we have a function f() = 10*kon
func_name = model.functions[0] # this will return function name f()
func_expr = model.functions[func_name] # this will return function expression "10*kon"
```

Reaction rules

Reaction rules consist of two lists of species, one for reactants and one for products as well as a list of rate constants. There is a single rate constant if the rule is unidirectional and two rate constants if the rule is bidirectional.

```
# Let's say we have a rule: R1: A() + B() <-> C() kon,koff
rule_name = model.rules[0] # this will return "R1" string
rule_obj = model.rules[rule_name] # this is the full rule object
print(rule_obj) # prints bngl string
print(rule_obj.reactants) # prints the list [A(), B()], where each item is a species␣
↪object
print(rule_obj.products) # prints the list [C()], where each item is a species object
print(rule_obj.rate_constants) # prints the list ["kon","koff"]
print(rule_obj.bidirectional) # prints true since
```

bngmodel.setup_simulator

This method allows you to get a libroadrunner simulator of the loaded model.

```
import bionetgen
model = bionetgen.bngmodel("mymodel.bngl") # generates BNG-XML and reads it
librr_simulator = model.setup_simulator()
librr_simulator.simulate(0,1,10) # librr_simulator is the simulator object
```

This is an easy way to generate data for analyses of your model using Python.

Tutorials

For a brief tutorial showing how to use the library on a simple BNGL model, please see *Library Tutorial*.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search