

# Sputnik: a Stochastic Petri Net Library in Python

Philipp Buerger, Erik Clark, Ben Moore, Oliver Palmer

January 15, 2013

# Contents

<b>1</b>	<b>Installation</b>	<b>2</b>
<b>2</b>	<b>Quick start</b>	<b>3</b>
<b>3</b>	<b>Background</b>	<b>4</b>
<b>4</b>	<b>Using the Library - GUI</b>	<b>6</b>
<b>5</b>	<b>Using the Library - Command-line</b>	<b>11</b>
<b>6</b>	<b>Examples</b>	<b>23</b>
<b>7</b>	<b>Extending the Framework</b>	<b>31</b>
	<b>References</b>	<b>35</b>

# 1. Installation

**Sputnik** encompasses a range of tools that allow the design, simulation and analysis of stochastic Petri nets. **Sputnik** is written for use with Python version 2.7 and has been tested on both Linux and OSX operating systems. It has the following dependencies:

**Essential libraries:**

NumPy,<sup>†</sup> Scipy,<sup>†</sup> matplotlib,<sup>†</sup> gtk, pygtk.

**Optional libraries:**

libsbml - Required if SBML import functionality is desired

<sup>†</sup>We recommend using the Enthought Python Distribution which is free for academic use and includes Python 2.7.2, NumPy, Scipy and matplotlib.









Having downloaded the latest version of **Sputnik** from [http://XXX.URL\\_needed/](http://XXX.URL_needed/), first extract the files using the following command (where XX is the version number):

```
> tar -xzf sputnik-XX.tar.gz
```

To run the GUI, simply call `python run-sputnik.py` from the `sputnik` directory. To import the library for use in other Python scripts, move the directory to somewhere on your `PYTHONPATH`. Similarly, to make the GUI accessible system-wide, move the directory to one listed in your shell's `PATH`.

## 2. Quick start

To jump into the program without reading the full documentation, follow this short vignette which is explained in more detail in section 6.1.

1. Open the Sputnik GUI with `python start.py`
2. Use  to open the `repressilator.txt` file found in the `examples/` directory.
3. View the calculated net invariants with .
4. Open the simulation  options, set **Runtime** to, for example, 100000 and **Time-Step** to 500 and click **Start Simulation**.
5. After a few seconds, an info-box will inform you the simulation is finished, click **Close**.
6. Now in the lower-right **Places** pane, uncheck ‘Plot’ for all places except **pA**, **pB** and **pC** — these are the three proteins of interest. Click **Show Plot** to see the results.
7. Optionally, use the various plot settings to add a title, choose custom line colours and edit legend-text and use the plot window dialogue to save the simulation plot.
8. Repeat the simulation by increasing the **Number of Simulations** to 4. Check the **Create subplots** checkbox and enter 2 in **Subplot Rows** and **Columns** to create a  $2 \times 2$  plot of all four simulations — highlighting the stochastic variation between runs.
9. Close the Simulation window and click  to view a token game animation of simulated Petri net events. Click **OK** to use the default parameters.
10. The play  button will then step through the simulation, highlighting firing transitions in **red** and updating the place markings as tokens are exchanged. You can also pause  and step through each event using  and .

For more detailed examples and a full explanation of **Sputnik**’s features and capabilities, consult sections 4, 5 and 6.

### 3. Background

Petri nets are a versatile and intuitive graph notation which can be used to represent many kinds of network. They offer an unambiguous mathematical framework to describe a system, as well as providing the means to test hypotheses regarding its behaviour and properties. Having evolved in mathematics and computer science<sup>[1]</sup>, Petri nets have more recently been used by systems biologists for the purpose of modelling biological systems<sup>[2,3,4,5]</sup>. Stochastic Petri nets are a form of Petri net suitable for simulating systems with stochastic mass action kinetics. They are therefore a convenient way to represent and simulate systems where low species abundances result in appreciable intrinsic noise. Such systems include gene regulatory networks, signalling systems, and many metabolic networks.

A Petri net consists of four types of element: places, transitions, arcs and tokens. **Places** are nodes representing the components of a system, while **transitions** are nodes representing reactions or events. **Arcs** are directed edges that connect a place to a transition (**pre arcs**) or *vice versa* (**post arcs**). Each arc is associated with an integer weight representing the stoichiometry of the transitions. The state of the system is given by the distribution of **tokens** among the places; the number of tokens possessed by a place is known as its **marking**. The firing of a particular transition, known as an **event**, moves tokens between places as directed by the pre and post arcs, changing the **global marking** of the system. A particular transition is **enabled** only if the markings of all its input places at least equal the weights of the corresponding pre arcs. Places may have a **capacity**, a maximum number of tokens that can be possessed by the place. **Test arcs** and **inhibitory arcs**, which represent minimum and maximum marking conditions respectively, may provide additional checks on whether a particular transition is enabled, but do not directly result in the consumption of tokens. The **P-invariants** of a net are combinations of places with a constant total marking, while the **T-invariants** are patterns of transition firings that leave the global marking unchanged<sup>[6]</sup>. The transitions of stochastic Petri nets are associated with stochastic **rate constants** and

fire probabilistically, with exponentially distributed waiting times between events<sup>[7]</sup> .

Here we present a Python library, **Sputnik**, that provides a range of tools for constructing, visualising, analysing and simulating stochastic Petri nets, accessible either from the command-line or from a user-friendly GUI. Petri net models may be imported from a file, or created *de novo* using the Petri net editor. Automatic layouting of the nets is achieved via a choice of drawing algorithms, while net components can also be manually repositioned for fine-tuning. Petri nets may be simulated by either of two stochastic simulation algorithms, the Gillespie or the tau-leap, and the resulting timecourses can be visualised in customisable plots. Net properties such as the  $P$ - and  $T$ -invariants, the stoichiometry matrix and the dependency matrix can also be calculated.

In addition to defining a TXT format for storing Petri net models, **Sputnik** also supports the systems biology exchange formats Systems Biology Markup Language (SBML) and Petri Net Markup Language (PNML). Petri nets may be imported or exported in any of these three file formats, ensuring cross-compatibility with the growing body of systems biology software already available. Petri net visualisations may be exported in a variety of common image formats, and the layout coordinates may additionally be saved to file, to be re-imported the next time the Petri net is loaded. Simulation visualisations may also be saved, or the raw data exported for further analysis.

We advise that you read through the Examples to get a feel for how the library works, from either the GUI or the command-line. For full instructions on using the GUI, see Chapter 4. If you prefer to write your own scripts, see Chapter 5 for documentation covering the relevant methods and variables.

## 4. Using the Library - GUI

The Graphical User Interface (GUI) integrates the different components of the library into a convenient and easy-to-use interface. Table 4.1 provides a short explanation for each of the icon buttons in the main window (Figure 4.1).

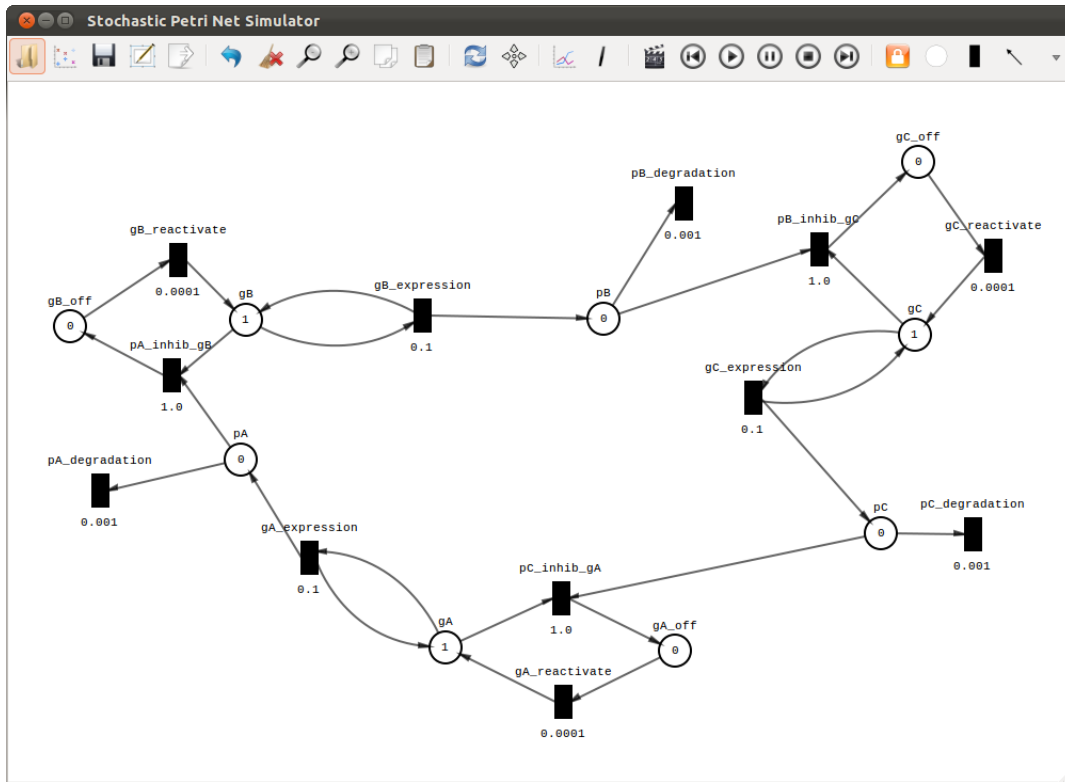


Figure 4.1: The main GUI window containing an example model, the Repressilator<sup>[8]</sup>. The function of each of the buttons of the main toolbar is described in Table 4.1.


































Icon	Shortcut	Description
	Ctrl + O	Open Petri net
	Ctrl + I	Open layout
	Ctrl + S	Save Petri net
	Ctrl + E	Save layout
	Ctrl + P	Export Petri Net
	Ctrl + Z	Undo
	Del	Delete
	- (minus)	Zoom out
	+ (plus)	Zoom in
	Ctrl + C	Copy
	Ctrl + V	Paste
	F5	Refresh
	Alt + L	Layout algorithms
	Alt + D	Simulation plot
	Alt + I	Calculate invariants
	Alt + G	Token Game simulation
	B	Previous step
	R	Play
	P	Pause
	S	Stop
	F	Next step
	Ctrl + L	Lock
	F6	Add Place
	F7	Add Transition
	F8	Add Standard Arc
	F9	Add Inhibitory Arc
	F10	Add Test Arc
	Alt + F4	Exit
	Esc	Abort

Table 4.1: Summary of the button icons found in the GUI main toolbar








## 4.1 Basic Functionality






**Loading and saving Petri nets** A previously saved Petri net can be loaded () from or saved () to a *TXT*, *PNML* or *SBML* file. The current layout can also be saved () to or loaded () from a separate file, which is useful if a net has been manually arranged.

**Petri net layout** Upon opening a Petri net, the positions of the graph elements are laid out by default using a spectral algorithm. To change the layout parameter values, open the layout window () where the **width**, **height**, and **border** size of the drawing area can be set. The **displacement radius** parameter value controls the minimum distance between components.


**Layout refinement** Drag-and-drop functionality can be used to reposition Petri net elements manually. By holding **Ctrl** and dragging across an area, multiple Petri net components can be selected and moved simultaneously.


**Locking the Petri net** Locking the active net () prevents components from being moved or deleted. Additionally, unwanted changes can be reversed using the undo feature ()

**Editing the Petri net** Components can be copied and pasted either using keyboard shortcuts or using the  and  buttons. Newly-copied components will initially be placed at the same location as the original component. Components can be deleted using **Del** or clicking . Deleted components that have connected arcs will also have their arcs removed.

**Adding components** New Petri net elements can be added by clicking the appropriate icon (place , transition , standard arc , test arc  or inhibitory arc ) and placing the component on the drawing area. When adding new arcs, valid connections will be highlighted **green**, invalid connections **red**.

**Modifying component properties** To modify a component's properties, double click the component to bring up the properties window.

**Exporting and saving images** The Petri net can be exported () in a range of formats including: *PDF*, *PS* and *SVG*.

**Petri net animation** A Token Game simulation of the active Petri net can be viewed by clicking  (Figure 4.2). The animation is controlled by a player-like menu and the progress bar can be used to jump to a specific position in the simulation. Note that the animation feature is only available when simulating with the Gillespie algorithm, since the tau-leap algorithm does not explicitly simulate individual events.

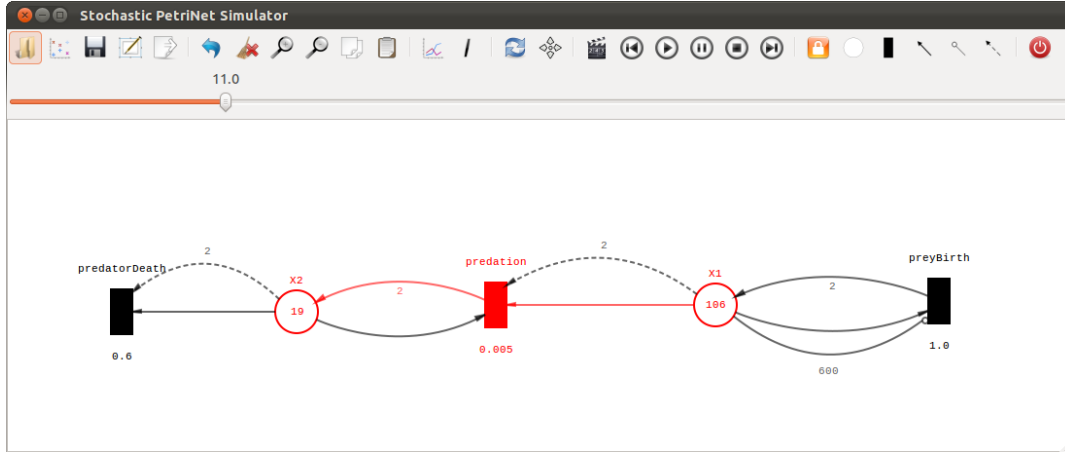



Figure 4.2: The Token Game animation. Petri net objects highlighted in red are those taking part in an event.

## 4.2 Simulating a Petri net

In order to simulate a Petri net, click  to bring up the simulation configuration window. Enter the appropriate parameter values and click **Start Simulation**. For information on selecting a simulation algorithm and simulation parameters please see section 5.4. We recommend using the Gillespie algorithm for most purposes, so it is selected by default.

**N.B. it is only possible to simulate Petri nets that have rates specified for all transitions. Many biological Petri net models available on the internet are incomplete and will require rate constant data to be added before they may be simulated.**

After the simulation is finished, the results can be visualised by clicking **Show Plot**. There are a variety of plot customisation options (Figure 4.3):

**Plot options** — parameters for the whole plot, such as title, line width and axes labels.

**Subplot options** — parameters for each individual subplot. Select the **Create Subplots** checkbox to apply. If the simulation consisted of multiple simulation runs, each individual

simulation will be rendered as a subplot. If the simulation consisted only of a single run, subplots will be generated for each individual species.

**Line options** — parameters of each individual timecourse in a plot, for example their colour and legend text. As with individual simulations, checkboxes can be used to select which species are plotted. In order to manually assign line colours, the **automatic colour allocation** checkbox must be deselected.

The screenshot shows the 'Configuration: Simulation' window with the following settings:

- Algorithm:** Gillespie (selected), Tau-Leap (unselected)
- Runtime:** 50
- Time-Step:** 1
- Number of Simulations:** 1
- Epsilon (Tau Leap only):** 0.03
- Control Parameter (Tau Leap only):** 10
- Number of SSA Runs (Tau Leap only):** 100
- General Diagram-Options:**
  - Title:** Simulation, ☒ Show Title
  - X-Label:** Runtime
  - Y-Label:** Marking
  - Linewidth:** 1
  - Legend Position:** 0, ☒ Show Legend
  - ☐ Create Subplots, ☒ Automatic Position Allocation
- Number of Rows and Columns:** 1, 1
- Specific Diagram-Options:**
  - Title:** , ☒ Show Subtitle
  - X-Label:** Runtime
  - Y-Label:** Marking
  - Subplot Position:** 0
  - Legend Position:** 0, ☒ Show Subplot Legend
- Specific Plot-Options:**
  - Legend-Text:**
  - Color:** [Blue color swatch], ☒ Automatic Color Allocation

Buttons at the bottom: Start Simulation, Show Plot, Cancel.

On the right, there are two tables:

**What should be displayed within the diagrams?**

**Simulations:**

Plot	Simulation	Index
<input checked="" type="checkbox"/>	Simulation 1	1

**Places:**

Plot	Place-Key	Place-Label
<input checked="" type="checkbox"/>	X2	X2
<input checked="" type="checkbox"/>	X1	X1

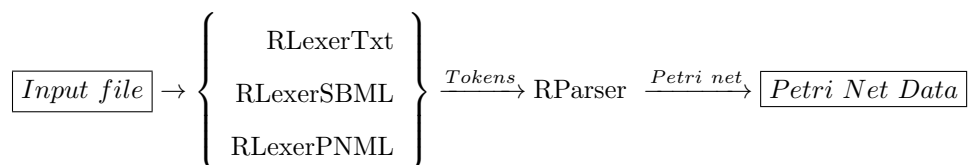
Figure 4.3: The configuration window for choosing simulation parameter values and customising plots of the output.

# 5. Using the Library - Command-line

## 5.1 Import and export of Petri nets

### 5.1.1 Loading a Petri net

In **Sputnik**, file input is parsed to **PetriNetData** via:

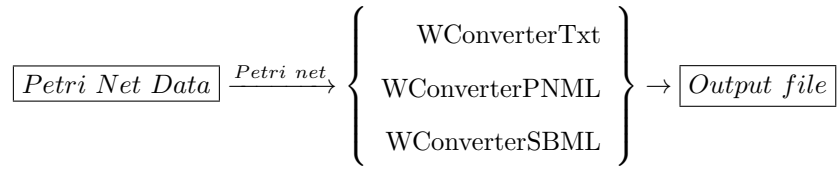


Loading a Petri net in this way, through iPython or using a script, requires the import of two classes, a general parser `rparsen.RParser()` and a lexer that is specific to the file type you are opening, i.e. `txt2_lex.RLexerTxt()`, `sbml_lex.RLexerSBML()` or `pnml_lex.RLexerPNML()`. As an example, the following commands will parse a `.txt` file to a **PetriNetData** object:

```
import spnlib_io 1
2
lexer = spnlib_io.RLexer() 3
parser = spnlib_io.RParser() 4
inputfile = open('examplefile.txt') 5
tokens = lexer.lex(inputfile) 6
parser.data = tokens 7
parser.parse() 8
PetriNet = parser.output 9
```

### 5.1.2 Saving a Petri net

In order to save a Petri net stored in a `PetriNetData` object, the following process is implemented:



To perform these actions as part of a script, for example to save to PNML format, the following commands can be issued:

```
import spnlib_io 1
2
converter = spnlib_io.WConverterPNML(PetriNet) 3
converter.save('outputfilename.pnml') 4
```

### 5.1.3 Text format specification

Writing a plain text file is often the fastest way to define a new Petri net for use with **Sputnik**. The text format closely mirrors a mathematical Petri net definition, and is specified below, with examples given for a simple Lotka-Volterra system<sup>[2]</sup>.

**Places** – a list of strings that represent identifiers for each Petri net place.

Required: Yes

Syntax: `p / places`

Example: `p = [prey,predator]`

**Transitions** – a list of strings identifying transition names.

Required: Yes

Syntax: `t / transitions`

Example: `t = [preyBirth,predation,predatorDeath]`

**Pre** – a matrix describing the weights of arcs connecting places to transitions.

Required: Yes

Syntax: `pre / pre_arcs / pre arcs`

Example: `pre = [[1,0],[1,1],[0,1]]`

**Post** – a matrix describing the weights of arcs connecting transitions to places.

Required: Yes

Syntax: `post / post_arcs / post arcs`

Example: `post = [[2,0],[0,2],[0,0]]`

**Initial marking** – a list of integers that count the number of tokens in each place prior to simulation. Should be the same length as places.

Required: No (set to 0 if absent)

Syntax: `m / markings`

Example: `m = [100,20]`

**Rates** – a list of numbers that represent the stochastic rate constants associated with each transition.

Required: No (set to 0 if absent)

Syntax: `r / rates`

Example: `r = [1,0.005,0.6]`

**Capacities** – a list of integers defining the maximum number of tokens allowed in each place. If present, must have the same length as places.

Required: No

Syntax: `c / capacities`

Example: `c = [500,500]`

**Inhibitory arcs** – a matrix describing a logical test performed on place markings to enable or disable a transition. Transitions will not fire if stated markings are above the values stated in this matrix (zeros are ignored), otherwise they fire as normal.

Required: No

Syntax: `i / inhib / inhibitory_arcs / inhibitory arcs`

Example: `inhib = [[600,0],[0,0],[10,0]]`

**Test arcs** – a matrix describing a logical test performed on place markings to enable or

disable a transition. Transitions will not fire if stated markings are below or equal to the values stated in this matrix, otherwise they fire as normal.

Required: Yes

Syntax: `test / test_arcs / test arcs`

Example: `test = [[2,0],[2,0],[0,2]]`

#### 5.1.4 Standalone file conversion

Standalone file conversion can be performed using `fconvert`. This must be run with two parameters, input and output filenames (relative or absolute paths), in the form:

```
$ ./fconvert inputfile.txt outputfile.xml
```

File extensions `.txt`, `.sbml`, `.pnml` and `.xml` are automatically detected, otherwise the user will be prompted to state the file type.

**Note:** the ability to read write SBML requires the freely available `libSBML` Python API (<http://sbml.org/Software/libSBML/docs/python-api/>).

## 5.2 Manual Petri net input

A stochastic Petri net is defined by the  $n$ -tuple  $N = \{P, T, Pre, Post, M_0, R, C^*, Test^*, Inhib^*\}$ , where the starred elements are optional. A Petri net model may be input manually, by creating a `spnlib_Petrinet.PetriNetData()` object and then setting its instance variables as the appropriate NumPy matrices and arrays. The *Pre* and *Post* matrices are stored in a `spnlib_Petrinet.Stoich()` object that is set as the `stoichiometry` instance variable of the `PetriNetData()` object (for details of the plaintext format, see section 5.1.3). After the data elements have been entered, certain net properties must be calculated before simulations can be run. A Petri net `p` should be created using the following template:

```
import numpy as np 1
import spnlib_Petrinet 2
3
p = spnlib_Petrinet.PetriNetData() 4
p.stoichiometry = spnlib_Petrinet.Stoich() 5
p.stoichiometry.pre_arcs = np.matrix(<post_arc_matrix>) 6
p.stoichiometry.post_arcs = np.matrix(<post_arc_matrix>) 7
```

```

p.places = np.array(<place_labels>)
p.transitions = np.array(<transition_labels>)
p.rates = np.array(<rates>, dtype=float)
p.initial_marking = np.array(<initial_marking>)

## OPTIONAL

p.capacities = np.array(<capacities>)
p.test_arcs = np.matrix(<post_arc_matrix>)
p.inhibitory_arcs = np.matrix(<post_arc_matrix>)

## Calculate net properties (required for simulations)

p.stoichiometry.calculate_stoichiometry_matrix()
p.stoichiometry.calculate_dependency_matrix()
p.stoichiometry.calculate_consumed()
p.stoichiometry.calculate_species_hors()

```

### 5.3 Calculating the invariants

The  $P$ - and  $T$ -invariants of a net are calculated by a `spnlib_Petrinet.PTInvariants()` object associated with a `spnlib_Petrinet.PetriNetData()` object. The invariants can then be accessed via the `p.invariants` and `t.invariants` instance variables of this object. To calculate and display the invariants of a `PetriNetData()` object, `p`, issue the following commands:

```

import spnlib_Petrinet

i = spnlib_Petrinet.PTInvariants()
i.set_Petri_net(p)
i.calculate_p_invariants()
i.calculate_t_invariants()

print i.p_invariants
print i.t_invariants

```



## 5.4 Running a simulation

Running simulations of a Petri net from the command line is very useful, since all the raw data from each simulation run is accessible for whatever further analysis may be desired by the user.

**Sputnik** can simulate Petri nets using either an exact stochastic simulation algorithm (SSA), the Gillespie, or an approximate SSA, the tau leap. Exact SSAs sample the waiting time to each new reaction explicitly, using exact reaction hazards whereas approximate SSAs use averaged hazards and do not sample every waiting time explicitly, sacrificing some degree of accuracy for increased algorithm speed. The Gillespie SSA can operate on nets including test arcs, inhibitory arcs and capacities, and is a good algorithm for most situations. Further, it must be used if precise event timings are required as output. The tau leap SSA may be used when the model to be simulated involves large species abundances. Under these conditions, the tau leap offers significant speed advantages over the Gillespie, while still offering accurate simulation. **N.B. the tau leap cannot be used with test arcs, inhibitory arcs or capacities.**

`spnlib.simulation.Gillespie()` and `spnlib.simulation.TauLeap()` are subclasses of `spnlib.simulation.Algorithm()`, an abstract base class providing the general instance variables `algorithm`, `num_runs`, `num_iterations`, `run_time` and `time_step`, plus the general interface method `run_simulation()`. When a simulation algorithm subclass is instantiated, `algorithm` is automatically set to the correct algorithm type, causing the command `run_simulation()` to then be assigned to the appropriate simulation function. The simulation may be run for a certain number of iterations (`num_iterations`) or until a certain time  $t$  is reached, with output data being stored at regular timepoints. In the latter case, both `run_time` and `time_step` should be set to the desired values, with `num_iterations` left as its default value, `None`.

Each simulation run initialises an object from the appropriate `spnlib.simulation.SimData()` subclass, `spnlib.simulation.GillespieData()` or `spnlib.simulation.TauLeapData()`. Five types of output data are stored in this class:

- **times:** a 1D NumPy array of timepoints of length  $n$
- **markings:** a 2D NumPy array ( $n \times P$ ) of net markings at each timepoint

- **events:** usually a 2D NumPy array  $((n - 1) \times P)$  of event firing frequencies between each timepoint. For a Gillespie simulation run `by_iteration`, this is a 1D array where each entry gives the index of the event that fired that iteration.
- **event\_freqs:** a list (length  $T$ ) giving the total frequency with which each transition fired
- **iterations:** the number of iterations simulated

Each `SimData()` object is appended to a list, which is set as the `simulation_data` instance variable of the `Gillespie()` or `TauLeap()` object.

### 5.4.1 Running the Gillespie SSA

```

## Initialise a simulation object 1
import spnlib_simulation 2
g = spnlib_simulation.Gillespie() 3
g.Petri_net = <PetriNetData object> 4
5
## OPTIONAL set the number of runs if multiple runs are desired 6
g.num_runs = <num_runs> 7
8
## EITHER choose a number of iterations 9
g.num_iterations = <num_iterations> 10
11
## OR set a run time and time step 12
g.run_time = <run_time> 13
g.time_step = <time_step> 14
15
## Run the simulation 16
g.run_simulation() 17
output = g.simulation_data 18
19
## You may print the raw data for a particular run (uses zero 20
indexing)
print output[<run>].times 21
print output[<run>].markings 22
print output[<run>].events 23

```

```
print output[<run>].event_freqs
```

24

```
print output[<run>].iterations
```

25

## 5.4.2 Running the tau leap SSA

There are three additional parameters for the tau leap algorithm compared to the Gillespie. The default values will be suitable in most circumstances. For a full explanation of the algorithm and its parameters, consult Cao *et al.* (2006)<sup>[9]</sup>.

1. **epsilon** (default = 0.03; sensible range = 0-0.1 ) is a value between 0 and 1 that adjusts the stringency of the algorithm,  $\epsilon$ . The smaller the value of  $\epsilon$ , the smaller the calculated values of  $\tau$ , since  $\epsilon$  is the maximum permitted relative change in any reaction hazard each timestep.
2. **control\_parameter** (default = 10; sensible range = 2-20) determines the threshold place marking that will cause the Gillespie algorithm to be invoked.
3. **num\_ssa\_runs** (default = 100) determines how many iterations of the Gillespie algorithm should be run each time the marking of a place drops below the control parameter value.

```
## Initialise a simulation object 1
import spnlib_simulation 2
t = spnlib_simulation.TauLeap() 3
t.Petri_net = <PetriNetData object> 4
5
## OPTIONAL set the number of runs if multiple runs are desired 6
t.num_runs = <num_runs> 7
8
## EITHER choose a number of iterations 9
t.num_iterations = <num_iterations> 10
11
## OR set a run time and time step 12
t.run_time = <run_time> 13
t.time_step = <time_step> 14
15
## OPTIONAL change the simulation parameters from default 16
t.epsilon = <epsilon> 17
t.control_parameter = <control_parameter> 18
t.num_ssa_runs = <num_ssa_runs> 19
20
## Run the simulation 21
t.run_simulation() 22
```

```

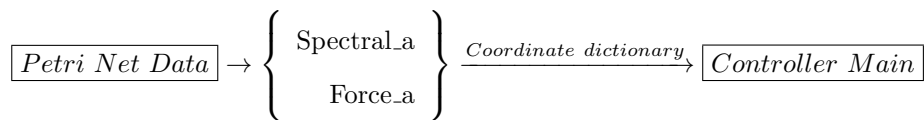
output = t.simulation_data 23
24
## You may print the raw data for a particular run (uses zero 25
indexing)
print output[<run>].times 26
print output[<run>].markings 27
print output[<run>].events 28
print output[<run>].event_freqs 29
print output[<run>].iterations 30

```

## 5.5 Petri net Layout

### 5.5.1 Obtaining Petri net layout coordinates

In **Sputnik**, a Petri net is laid out in the following way:



Laying out a Petri net in this way, through iPython or using a script, requires the import of 2 classes, **Petri\_net\_data** class and a layout algorithm selection e.g. **spectral.a** or **force.a**.

#### Spectral algorithm

To obtain raw layout coordinates for a Petri net in the a **Petri\_net\_data** object named as **p**:

```

import numpy as np 1
import spectral.a 2
3
v = spectral.a.Spectral() 4
v.Petri_net = p 5
v.get_Petri_net() 6
v.width = 1000 7
v.height = 1000 8
v.border = 100 9
v.d_radius = 40 10
coordinates = v.calculate() 11

```

The `width`, `height`, `border` and `grid_size` variables have default values within the `spectral_a` class but they can be overridden as required.

### Force directed algorithm

For the force directed algorithm the process is the same but includes an optional `iterations` variable:

```
import numpy as np 1
import force_a 2
3
v = force_a.ForceDirected() 4
v.Petri_net = p 5
v.get_Petri_net() 6
v.width = 1000 7
v.height = 1000 8
v.border = 100 9
v.iterations = 100 10
coordinates = v.calculate() 11
```

The number of iterations required to reach an acceptable layout varies between graphs. For small graphs (tens of vertices) 100 iterations may be sufficient; for graphs containing hundreds of vertices, 1000+ iterations may be required. Some trial and error may be required.

### 5.5.2 Drawing a Petri net

It is possible to draw a Petri net without using the GUI.

To perform these actions as part of a script for a `Petrinet` object `pn`:

```
import numpy as np 1
import spectral_a 2
3
v = spectral_a.Spectral() 4
v.Petri_net = pn.Petri_net_data 5
v.get_Petri_net() 6
v.width = 1000 7
v.height = 1000 8
v.border = 100 9
v.d_radius = 40 10
```

```

v.calculate() 11
## set the positions and draw the Petri net 12
pn.set_positions(v.node_positions) 13

```

### 5.5.3 Drawing an undirected graph

To draw a graph which is not a Petri net, the graph's *Adjacency* matrix must be available.

```

import numpy as np 1
import spectral_a 2
3
v = spectral_a.Spectral() 4
v.width = 1000 5
v.height = 1000 6
v.border = 100 7
8
## In this context places refers to a unique identifier for each 9
vertex in the graph
v.places = np.array([a, b, c, d, e]) 10
11
## The adjacency matrix is ordered as the places array: 12
##
13
adjacency = np.matrix([
14     [ 0, 0, 1, 1, 0,], #a
15     [ 0, 0, 0, 1, 1,], #b
16     [ 1, 0, 0, 0, 0,], #c
17     [ 1, 1, 0, 0, 0,], #d
18     [ 0, 1, 0, 0, 0,]]) #e
19
20
## Create the degree and Laplacian matrices from the adjacency 20
degree = v.calculate_degree(adjacency) 21
laplacian = v.calculate_laplacian(degree, adjacency) 22
## returns dictionary of coordinates 23
coordinates = v.calculate_eigenvectors(laplacian) 24

```

# 6. Examples

In the following sections, two example systems are analysed using the **Sputnik** framework. Example 1 is run via the graphical user interface while Example 2 is demonstrated through a command-line interface.

## 6.1 Example 1 : Repressilator

The repressilator is a synthetic biochemical network designed for *Escherichia coli* (Figure 6.1). The system acts as a cellular clock, exhibiting oscillatory behaviour involving three protein-coding genes<sup>[10]</sup>.

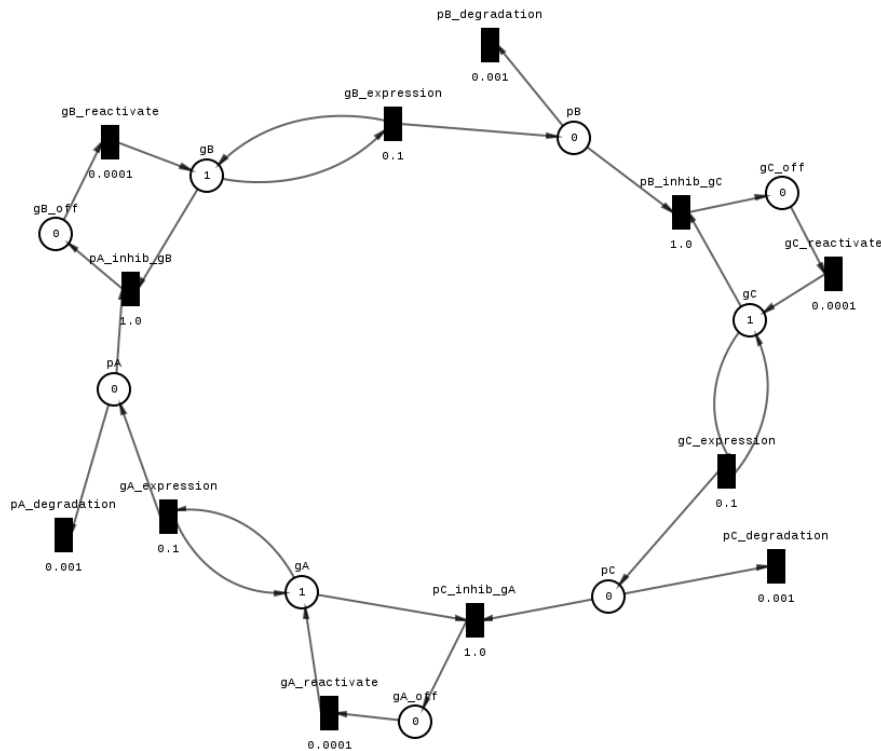


Figure 6.1: The repressilator genetic system.



The program is supplied with the following example file which represents the above Petri net:

```
## Repressilator text file ##

places= [gA, pA, gA_off, gB, pB, gB_off, gC, pC, gC_off]

t = [gA_expression, pA_degradation, pA_inhib_gB, gB_reactivate,
      gB_expression, pB_degradation, pB_inhib_gC, gC_reactivate,
      gC_expression, pC_degradation, pC_inhib_gA, gA_reactivate]


pre_arcs= [[1, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0],
            [1, 0, 0, 0, 0, 0, 0, 1, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0]]


post_arcs =[[1, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 1, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0],
            [1, 0, 0, 0, 0, 0, 0, 0, 0]]

rates = [0.1, 0.001, 1, 0.0001, 0.1, 0.001,
          1, 0.0001, 0.1, 0.001, 1, 0.0001]

marking = [1, 0, 0, 1, 0, 0, 1, 0, 0]
```

Figure 6.2: A plain text file which specifies the repressilator Petri net model.

To load this input file from the GUI, select open file () , then locate and open **repressilator.txt**, included in the **Sputnik** package in the **examples/** directory.

The Petri net will now be displayed. To calculate  $P$ - and  $T$ -invariants, select the invariants icon () . The following results window will then appear:

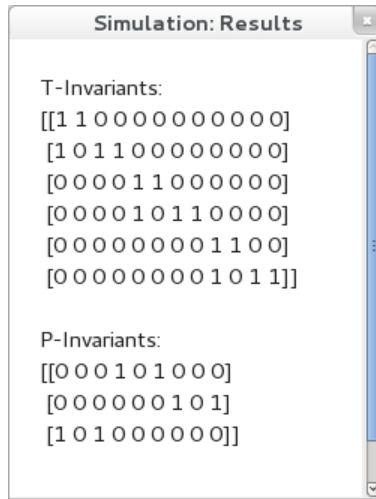



Figure 6.3: Calculated  $T$ - and  $P$ -invariants for the repressilator Petri net model.

To simulate the Petri net, click the simulation icon () . You will then be presented with the simulation configuration options:

**Configuration: Simulation**

Algorithm: ☒ Gillespie ☐ Tau-Leap

Runtime: 300000

Time-Step: 500

Number of Simulations: 1

Epsilon (Tau Leap only): 0.03

Control Parameter (Tau Leap only): 10

Number of SSA Runs (Tau Leap only): 100

**General Diagram-Options:**

Title: Repressilator simulation ☒ Show Title

X-Label: Runtime

Y-Label: Marking

Linewidth: 1

Legend Position: 0 ☐ Show Legend

☐ Create Subplots ☒ Automatic Position Allocation

Number of Rows and Columns: 1 1

**Specific Diagram-Options:**

Title:  ☒ Show Subtitle

X-Label: Runtime


Y-Label: Marking

Subplot Position: 0

Legend Position: 0 ☒ Show Subplot Legend

**Specific Plot-Options:**

Legend-Text: pA

Color:  ☐ Automatic Color Allocation

**What should be displayed within the diagrams?**

**Simulations:**

Plot	Simulation	Index
<input checked="" type="checkbox"/>	Simulation 1	1

**Places:**

Plot	Place-Key	Place-Label
<input checked="" type="checkbox"/>	gC_off	gC_off
<input checked="" type="checkbox"/>	gB_off	gB_off
<input checked="" type="checkbox"/>	pB	pB
<input checked="" type="checkbox"/>	pC	pC
<input checked="" type="checkbox"/>	pA	pA
<input checked="" type="checkbox"/>	gC	gC
<input checked="" type="checkbox"/>	gB	gB
<input checked="" type="checkbox"/>	gA	gA
<input checked="" type="checkbox"/>	gA_off	gA_off

Figure 6.4: Configuration options when simulating the firing .

In this example, the Gillespie algorithm has been chosen, the runtime has been set to 300000 and the Step-Size to 500. The Petri net can now be simulated by clicking **Start**

**Simulation.** An info box will inform the user when the simulation is complete. Before plotting the results, a number of parameters can be adjusted: For the example shown below (Figure 6.6), the **Title** was changed and **Show Legend** was deselected, additionally some of the line colors were set manually, using the **Specific Plot-Options**; to save these changes, click **Save Specific Plot Setting**, then to visualise the plot, click **Show Plot**.

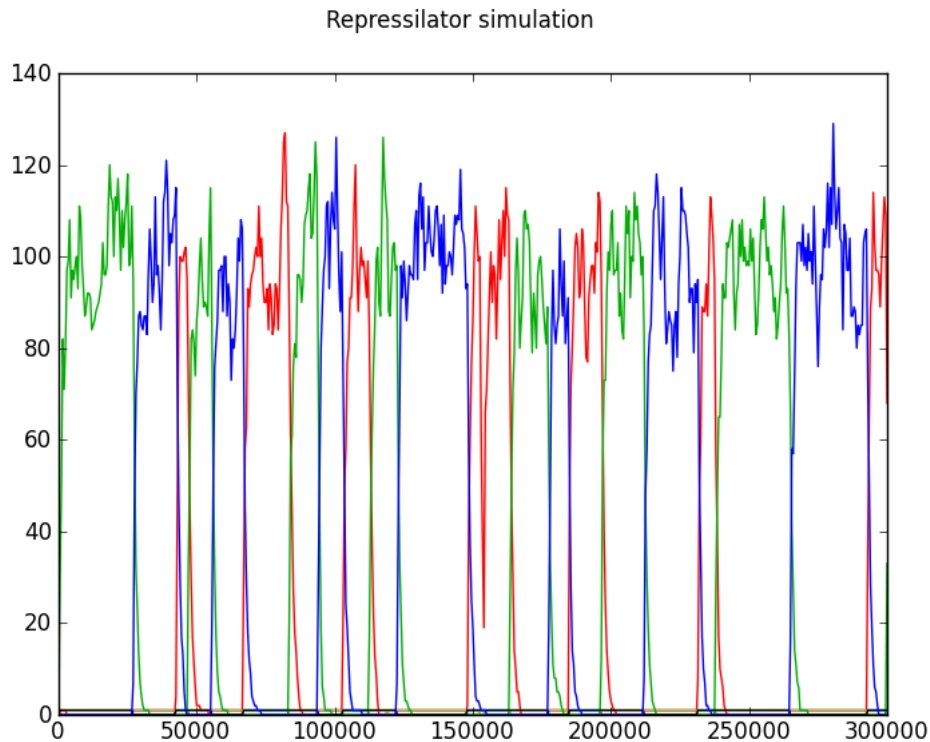


Figure 6.5: Graph output of stochastic simulation results for the repressilator Petri net model. The  $y$ -axis denotes place markings and the  $x$ -axis shows the runtime. The *red*, *green* and *blue* lines represent the three proteins in the repressilator model.

This graph can then be exported to numerous image formats, including **pdf**, **svg** and **png**.

## 6.2 Example 2 : Four-reaction system

A second example system, the four-reaction system<sup>[11]</sup>, will now be entered and simulated using a command-line interface. These commands can be run using iPython or as part of a Python script.

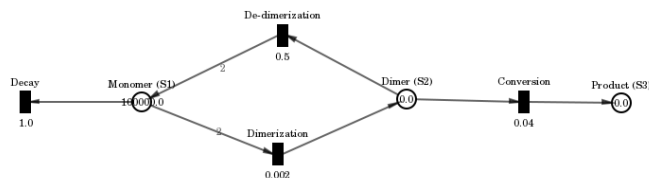


Figure 6.6: A Petri net representing the four-reaction system.

The first step required is to import specific modules of the **Sputnik** framework. In this example we will be using the tau-leap simulation method, due to the large number of molecules involved in this system. The required imports are:

```
import spnlib_Petrimet 1
import Petrimetdata_simulation 2
```

Other imports, required for plotting and data entry, are:

```
import matplotlib.pyplot as plt 1
import numpy as np 2
```

The Petri net can now be specified using the following statements (Note the large number of initial markings used in this example):

```
p = spnlib_Petrimet.PetriNetData() 1
p.places = np.array(['S1', 'S2', 'S3']) 2
p.transitions = np.array(['R1', 'R2', 'R3', 'R4']) 3
p.rates = np.array([1, 0.002, 0.5, 0.04], dtype=float) 4
p.initial_marking = np.array([100000, 0, 0], dtype=int) 5
```

Pre and post arcs are stored in a separate stoichiometry class:

```
s = spnlib_Petrimet.Stoich() 1
s.pre_arcs = np.matrix([[1, 0, 0], [2, 0, 0], [0, 1, 0], [0, 1, 0]], dtype=int) 2
s.post_arcs = np.matrix([[0, 0, 0], [0, 1, 0], [2, 0, 0], [0, 0, 1]], dtype=int) 3
```

```
s.calculate_stoichiometry_matrix() 4
s.calculate_dependency_matrix() 5
```

In preparation for Tau leap simulation, the following extra procedures need to be called:

```
s.calculate_consumed() 1
s.calculate_species_hors() 2
```

A `TauLeap` object is initialised. In order to run the simulation three parameter values are set, `Petri_net`, `run_time` and `time_step`:

```
t = spnlib_simulation.TauLeap() 1
t.Petri_net = p 2
t.run_time = 40 3
t.time_step = 0.01 4
```

Now you are ready to run the simulation and visualise the output:

```
t.run_simulation() 1
results = t.simulation_data 2
plt.plot(results[0].times, results[0].markings) 3
plt.show() 4
```

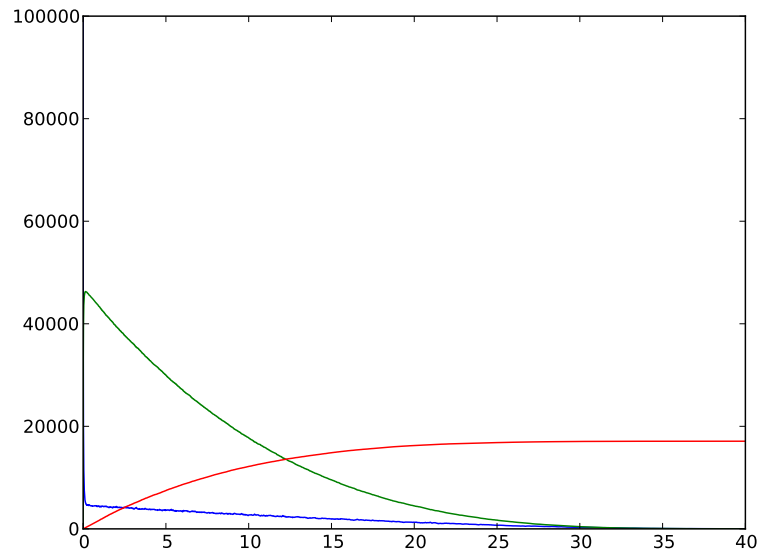


Figure 6.7: The results of Tau-leap simulation of the four-reaction system. The  $x$ -axis represents runtime and the  $y$ -axis shows place markings. Coloured lines are represent places as follows: *blue*: S1, *green*: S2, *red*: S3.

### 6.3 Example 3 : MAP kinase

The library contains two powerful layout algorithms to help the user make sense of Petri nets imported into the program. There are parameters within these algorithms that can be adjusted to optimise for a given scenario, be it the requirements of a specific graph, or particular drawing area size. To demonstrate how these parameters affect the graph layout process, the MAP kinase model will be used, due to its high layout difficulty as a result of its non planar characteristic and high connectivity.

When a new Petri net is loaded to the program it is automatically laid out using the spectral algorithm with default settings. An example of the MAP kinase system under default layout conditions is shown in Figure 6.8.

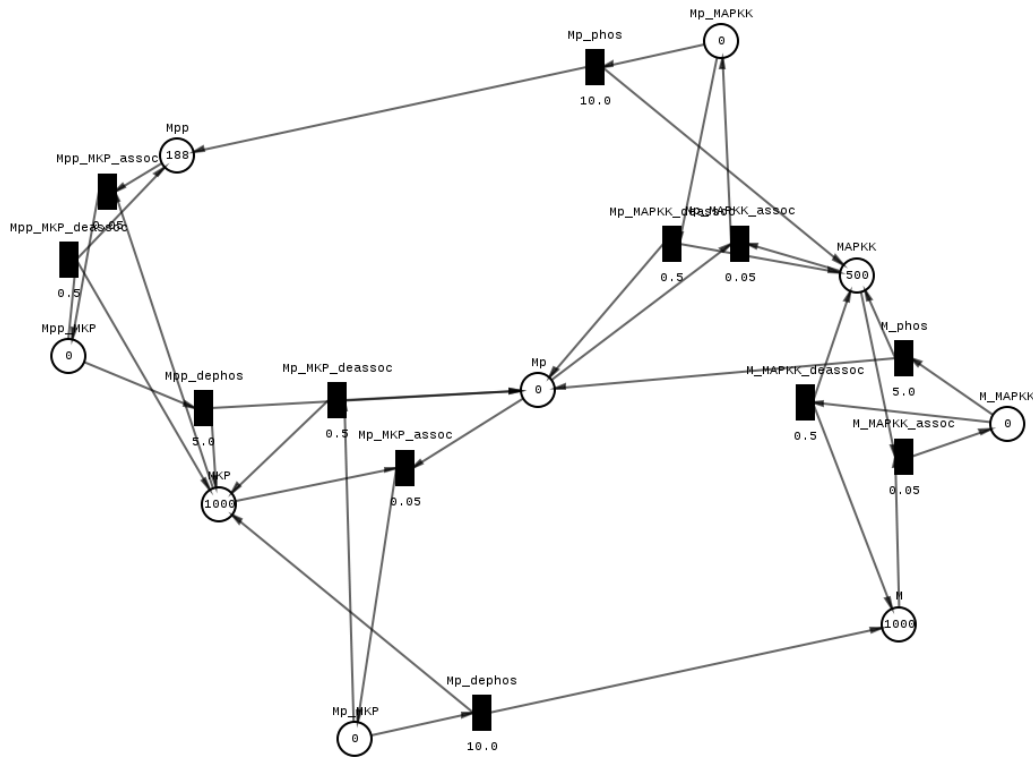


Figure 6.8: The default layout of the MAP kinase system shown here is not ideal, there is some clustering of objects and this has caused some of the object labels to overlap.

By decreasing the border size to provide more space for the layout of objects and increasing the size of the displacement radius to detect more clusters, you can generate a nicer automatic layout. The result of refining the layout parameters for the MAP kinase Petri net is shown in figure 6.9.

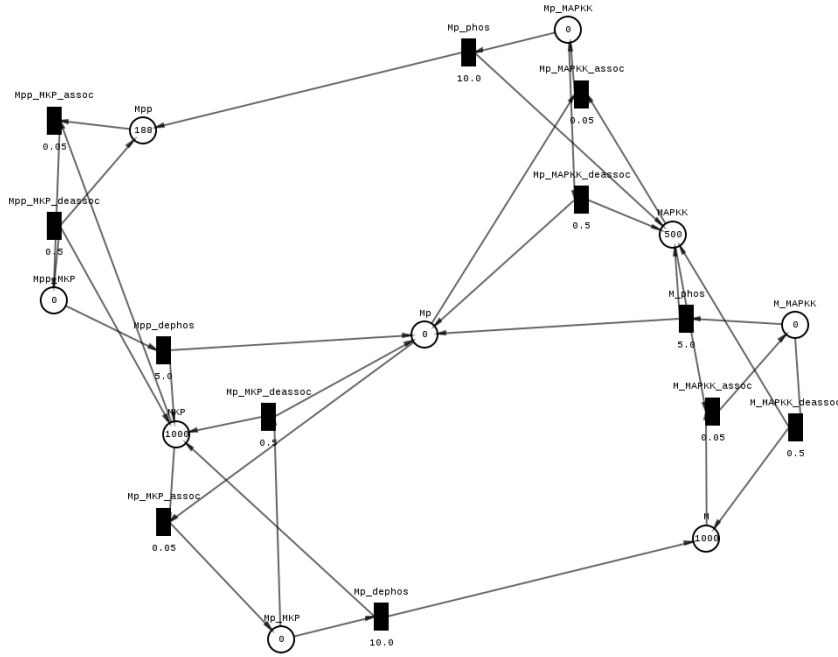


Figure 6.9: The refined layout of the MAP kinase system shown here has solved some of the problems that are present in figure 6.8 such as clustering of objects and the overlap of labels.

To further improve the layout, you can drag objects to the desired location, save the coordinates and lock the Petri net in place (Fig. 6.10).

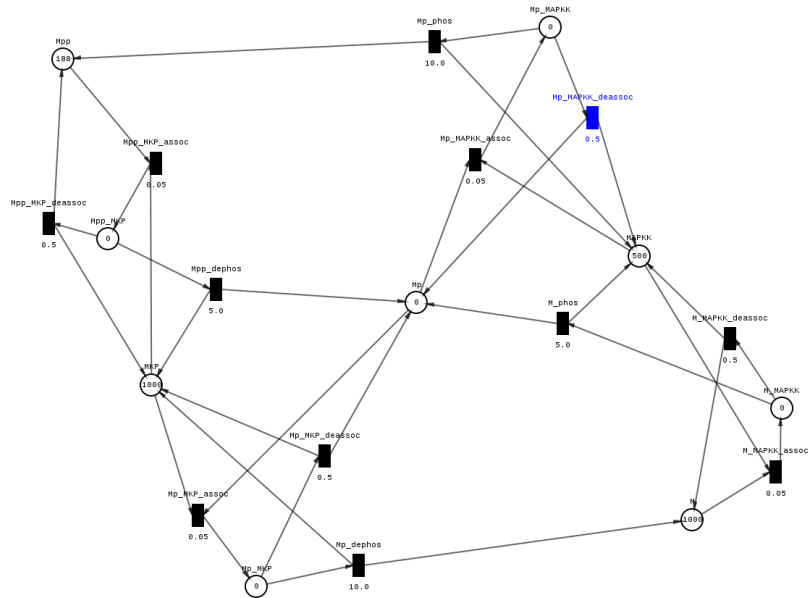


Figure 6.10: This figure demonstrates the improved layout that can be achieved with position refinement. The object highlighted in blue is the object currently being moved by the user.

# 7. Extending the Framework

The library was designed with an explicit object oriented paradigm to allow for simple extension in the future. The Model-View-Controller (MVC) Architecture forms the backbone of the library framework and permits straightforward extensibility of the GUI. Additionally an efficient notification concept operates throughout the architecture which notifies all observers when a data change occurs. The **PetriNet** class forms the centrepiece of the framework and is used to encapsulate single components from each other, allowing the independent extension of each framework component.

## 7.1 Extending the Graphical User Interface

Through the integrated MVC architecture, a new representation of the available data, or view, can be easily added. The general **View** class, which provides basic functionalities, inherits from the **MVCObserver** class and is used as a parent class of any new representation. This is similar to the specific controllers which inherit from the **Controller** class. The **Controller** class itself is again a child class from the **MVCObserver**. The views are needed to implement the graphical representations and the controllers are needed to manage the interactions between the user and the application. Both views and controllers can be registered at its **model**, a child class from **MVCObservable**. The following code presents a sample extension with a new view and controller. Figure 7.1 shows the general concept of the MVC architecture.



```

import gtk
import pygtk

import spnlib_gui

class ViewExtension(spnlib_gui.View):

    def __init__(self):
        ## call constructor of parent class
        spnlib_gui.View.__init__(self)

    def __init__(self, model = None, controller = None):
        ## call constructor of parent class
        spnlib_gui.View.__init__(self, model, controller)

        ## common methods which are used for display and
        notification purposes

    def show(self):
        pass

    def update_component(self, key):
        pass

    def update_output(self):
        pass

    def undo(self):
        pass

    def update(self):
        pass

    def reset(self):
        pass

class ControllerExtension(spnlib_gui.Controller):

```

```

def __init__(self):
    ## call constructor of parent class
    spnlib_gui.Controller.__init__(self)

def __init__(self, model = None, view = None):
    ## call constructor of parent class
    spnlib_gui.Controller.__init__(self, model, view)

## common methods which are used for display and notification
    purposes
def show(self):
    pass

def update_component(self, key):
    pass

def update_output(self):
    pass

def undo(self):
    pass

def update(self):
    pass

def reset(self):
    pass

```

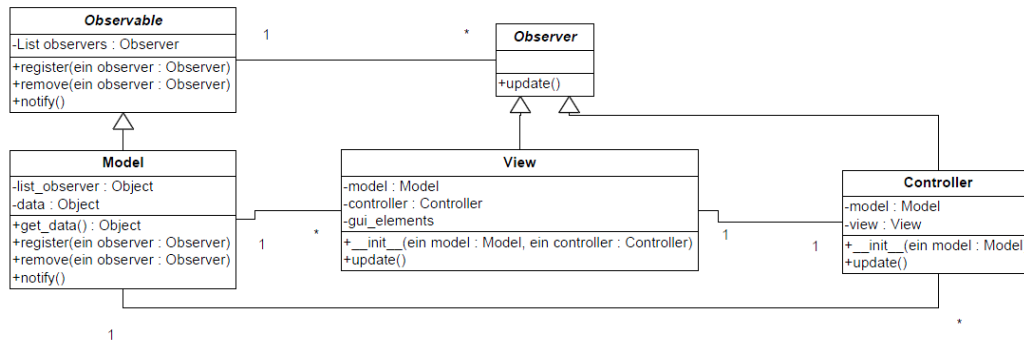


Figure 7.1: This figure shows the general concept of a Model-View-Controller (MVC) Architecture where the observer defines the parent class of the *view* and *controller*. The *view* implements the Graphical User Interface (GUI) and the *controller* handles user interactions from the *view*. Data are stored within the *model* which is a child class of the *Observable* class and implements the notification protocol to inform the registered observers (views and controllers)<sup>[12]</sup>.

# References

- [1] M. Ajmone Marsan. Stochastic petri nets: An elementary introduction. In *In Advances in Petri Nets*, pages 1–29. Springer, 1989.
- [2] D. Wilkinson. *Stochastic Modelling for Systems Biology*. Chapman & Hall/CRC Mathematical & Computational Biology, April 2006.
- [3] J. W. Pinney, D. R. Westhead, and G. A. McConkey. Petri net representations in systems biology. *Biochem Soc Trans*, 31(Pt 6):1513–1515, Dec 2003.
- [4] M. Heiner, D. Gilbert, and R. Donaldson. Petri nets for systems and synthetic biology. In *SFM 2008*, number 5016, pages 215–264. LNCS, 2008.
- [5] T. M. Chin and A. S. Willsky. Stochastic petri net modeling of wave sequences in cardiac arrhythmias. *Comput Biomed Res*, 22(2):136–159, Apr 1989.
- [6] T Toni. *Approximate Bayesian computation for parameter inference and model selection in systems biology*. PhD thesis, Imperial College London, 2010.
- [7] R David and H Alla. *Discrete, Continuous and Hybrid Petri Nets*. Springer, 2005.
- [8] Michael B. Elowitz and Stanislas Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335–338, January 2000.
- [9] Y. Cao, D. T. Gillespie, and L. R. Petzold. Efficient step size selection for the tau-leaping simulation method. *J Chem Phys*, 124(4):044109, Jan 2006.
- [10] M. B. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335–338, Jan 2000.
- [11] T. Tian and B. Burrage. Binomial leap methods for simulating stochastic chemical kinetics. *Journal of Chemical Physics*, 121(21):10356–10364, December 2004.
- [12] Buschmann F, Meunier R, Rohnert H, Sommerlad P, and Stal M. *Pattern - Orientierte Software Architektur: Ein Pattern - System*. Addison - Wesley, 1998.