PyBoolNet Documentation

Release 2.1

Hannes Klarner

February 24, 2017

PyBoolNet 2.1 is a Python package for the generation, manipulation and analysis of the interactions and state transitions of Boolean networks. The project home page is https://github.com/hklarner/PyBoolNet.

Note: *PyBoolNet 2.1* does not yet have any user friendly error messages. Please post questions, report bugs or suggest features in the issues section of the project's homepage:

- https://github.com/hklarner/PyBoolNet/issues

or contact hannes.klarner@fu-berlin.de

CHAPTER

ONE

INSTALLATION

1.1 Python

PyBoolNet 2.1 was written in Python 2.7 but should be compatible with Python 3. If you experience problems with your version of Python and *PyBoolNet 2.1* please contact *hannes.klarner@fu-berlin.de* or post an issue on the project homepage at

- http://github.com/hklarner/PyBoolNet/issues

1.2 Linux

Download the latest release from

- http://github.com/hklarner/PyBoolNet/releases

64bit and 32bit versions are available. We recommend to install the package using *pip*. If it is not already installed on your computer try:

```
$ sudo apt-get install python-pip
```

Make sure that *NetworkX*, *Graphviz* and *ImageMagick* are installed:

```
$ sudo pip install networkx
$ sudo apt-get install graphviz
$ sudo apt-get install imagemagick
```

Install *PyBoolNet 2.1* with *pip*:

```
$ sudo pip install PyBoolNet-2.1_linux64.tar.gz
```

which should place the package here:

```
/usr/local/lib/python<version>/dist-packages/PyBoolNet
```

Use the option `--user` (literally) if you do not have sudo rights:

```
$ pip install PyBoolNet-2.0.tar.gz --user
```

The package is likely going to be placed here:

```
/home/<user>/.local/lib/python<version>/dist-packages/PyBoolNet
```

where `<user>` is the name you are logged in with (`$ whoami`) and `<version>` is the Python version you are using. To install *PyBoolNet 2.1* using *Distutils* unpack *PyBoolNet-2.0.tar.gz* into a temporary folder and run:

```
$ sudo python setup.py install
```

again, using the `--user` flag if you do not have sudo rights:

```
$ python setup.py install --user
```

The locations should be the same as when installing with *pip*.

You should now be able to import *PyBoolNet 2.1*:

```
$ python
>>> import PyBoolNet
```

To remove *PyBoolNet 2.1* using *pip* run:

```
$ pip uninstall PyBoolNet
```

If you do not have *pip*, all files must be removed manually.

1.3 Mac OS

Download the latest release from

- http://github.com/hklarner/PyBoolNet/releases

We recommend to install the package using *pip*. If it is not already installed on your computer try:

```
$ sudo easy_install pip
```

or if you do not have super user rights:

```
$ easy_install --user pip
```

Install NetworkX with:

```
$ sudo pip install networkx
```

or:

```
$ pip install networkx --user
```

Download and install Graphviz and ImageMagick from

- http://www.graphviz.org/Download.php
- http://www.imagemagick.org/script/binary-releases.php

Install *PyBoolNet 2.1* with *pip*:

```
$ sudo pip install PyBoolNet-2.1_mac64.tar.gz
```

which should place the package here:

```
/usr/local/lib/python<version>/dist-packages/PyBoolNet
```

Use the option `--user` (literally) if you do not have sudo rights:

```
$ pip install PyBoolNet-2.0.tar.gz --user
```

The package is likely going to be placed here:

```
/home/<user>/.local/lib/python<version>/dist-packages/PyBoolNet
```

where `<user>` is the name you are logged in with (`$ whoami`) and `<version>` is the Python version you are using.

You should now be able to import *PyBoolNet 2.1*:

```
$ python
>>> import PyBoolNet
```

To remove *PyBoolNet 2.1* using *pip* run:

```
$ pip uninstall PyBoolNet
```

If you do not have *pip*, all files must be removed manually.

1.4 Windows

Download the latest release from

- http://github.com/hklarner/PyBoolNet/releases

We recommend to install the package using *pip*. If it is not already shipped with your Python version follow the instructions on

- http://pip.pypa.io/en/latest/installing

To install *PyBoolNet 2.1* with *pip*:

```
C:\> pip.exe install PyBoolNet-2.1_win64.tar.gz
```

which should place the package here:

```
C:\Python<version>\Lib\site-packages
```

where `<version>` is the Python version you are using.

Important: Make sure you check the paths to the executables. Locate the file `settings.cfg` in the `Dependencies` folder of your installation and try to run each program.

To install NetworkX use *pip* again:

```
C:\> pip.exe install netwokrx
```

To install Graphviz and ImageMagick download the respective executables from the home pages:

- http://www.graphviz.org/Download_windows.php
- http://www.imagemagick.org/script/binary-releases.php#windows

You should now be able to import *PyBoolNet 2.1*:

```
C:\> python
>>> import PyBoolNet
```

To remove *PyBoolNet 2.1* using *pip* run:

```
C:\> pip.exe uninstall PyBoolNet
```

If you do not have *pip*, all files must be removed manually.

1.5 Dependencies

Most of what *PyBoolNet 2.1* does is written in pure Python but some crucial tasks, for example solving ASP problems or deciding CTL queries, are done using third party software. The file that records the locations to third party binaries is called `settings.cfg` and located in the folder `Dependencies` of *PyBoolNet 2.1*. The default location is:

```
/usr/local/lib/python<version>/dist-packages/PyBoolNet/Dependencies/settings.cfg
```

The file is a standard configuration file of `name = value` pairs. The default for Linux 64 bit is:

```
[Executables]
nusmv           = ./NuSMV-a/NuSMVa_linux64
gringo          = ./gringo-4.4.0/gringo
clasp           = ./clasp-3.1.1/clasp-3.1.1-x86_64-linux
bnet2prime      = ./BNetToPrime/BNetToPrime_linux64
dot             = /usr/bin/dot
neato           = /usr/bin/neato
fdp             = /usr/bin/fdp
sfdp            = /usr/bin/sfdp
circo           = /usr/bin/circo
twopi           = /usr/bin/twopi
convert         = /usr/bin/convert
```

If you want to use your own binaries simply replace the respective paths. Note that `./` indicates a relative path while `/` is an absolute path. Make sure all these paths work and that rights for execution and access are set on linux systems. To test whether the dependencies work correctly, run:

```
$ python
>>> import PyBoolNet
>>> PyBoolNet.Tests.Dependencies.run()
```

If you get fails or errors, read *Troubleshooting* and the issues section of the homepage:

- http://github.com/hklarner/PyBoolNet/issues

where you can also post issues. Also, do not hesitate to contact me at hannes.klarner@fu-berlin.de

1.5.1 NetworkX

NetworkX is a Python package and required for standard operations on directed graphs, e.g. computing strongly connected components, deciding if a path between two nodes exists. The package is available at:

- http://networkx.github.io

To install it on Linux using *pip* run:

```
$ sudo pip install networkx
```

or:

```
$ pip install networkx --user
```

if you do not have super user rights.

Note: *PyBoolNet 2.1* is tested with NetworkX version 1.10 and older versions will almost surely not work.

1.5.2 BNetToPrime

BNetToPrime stands for "Boolean network to prime implicants". It is necessary to compute the prime implicants of a Boolean network. It is included in every release and should work out of the box. The binaries and source are available at:

- http://github.com/xstreck1/BNetToPrime

1.5.3 Potassco

The Potassco answer set solving collection consists of the ASP solver clasp and the grounder gringo, see *Gebser2011*. They are necessary to compute trap spaces by means of stable and consistent arc sets in the prime implicant graph, see *Klarner2015(a)*. They are included in every release and should work out of the box.

Note: The development of the Potassco solving collection is active with frequent releases. *PyBoolNet 2.1* is tested with two specific versions, clasp-3.1.1 and gringo-4.4.0 and we strongly recommend you use them because of syntax differences between versions.

The binaries and source are available at:

- http://sourceforge.net/projects/potassco/files/clasp/3.1.1
- http://sourceforge.net/projects/potassco/files/gringo/4.4.0

1.5.4 NuSMV

NuSMV is a symbolic model checker that we use to decide LTL and CTL queries. *PyBoolNet 2.1* requires the extension NuSMV-a for model checking with accepting states. It is included with every release and should work out of the box.

Note: *PyBoolNet 2.1* is tested with NuSMV-a, an extension of NuSMV 2.6.0. If you do not need to compute accepting states you may use the regular NuSMV 2.6.0.

Binaries and source available at:

- http://github.com/hklarner/NuSMV-a

1.5.5 Graphviz

The program *dot* is part of the graph visualization software Graphviz and available at

- http://www.graphviz.org

It is required to generate drawings of interaction graphs and state transition graphs. To install it on Linux run:

```
$ sudo apt-get install graphviz
```

Make sure to check the paths in `settings.cfg`.

1.5.6 ImageMagick

The program *convert* is part of the ImageMagick software suite. It is required to generate animations of trajectories in the state transition graph. To install it on linux run:

```
$ sudo apt-get install ImageMagick
```

ImageMagick is available at

- http://www.imagemagick.org

Make sure to check the paths in `settings.cfg`.

1.6 Additional Software

1.6.1 BoolNet

BoolNet is a library for R that is used for the construction, simulation and analysis of Boolean networks, see *Müssel2010*. It is not a required dependency of *PyBoolNet 2.1* but you need it if you want to convert *SBML-qual* files into *bnet* files. To install it run:

```
$ sudo R
> install.packages("BoolNet")
```

select a CRAN mirror and wait for the download and installation to finish. BoolNet is available at

- http://cran.r-project.org/web/packages/BoolNet/index.html

1.6.2 GINsim

GINsim is a Java program for the construction and analysis of qualitative regulatory and signaling networks, see *Chaouiya2012*. Like BoolNet, GINsim is not a required dependency of *PyBoolNet 2.1* but it has a useful model repository. To convert GINsim models you need to export them as *SBML-qual* files which can then be converted into *bnet* files using BoolNet. No installation required, just download the latest version (tested with version 2.9) and call:

```
$ java -jar GINsim-2.9.3.jar
```

GINsim is available at

- http://www.ginsim.org

1.7 Troubleshooting

For questions that are not listed here please contact *hannes.klarner@fu-berlin.de* or post an issue on the project homepage at

- http://github.com/hklarner/PyBoolNet/issues

1.7.1 libreadline.so.6

If you get the error message:

```
./NuSMVa: error while loading shared libraries: libreadline.so.6:
cannot open shared object file: No such file or directory
```

then a solution for linux is available at stackoverflow:

- http://stackoverflow.com/questions/23993377/red-language-console-error-libreadline-so-6-cannot-open-shared-object-file

The crucial command:

```
$ sudo apt-get install libreadline6:i386
```

1.7.2 permission denied

If you get *permission denied* erros like:

```
OSError: [Errno 13] Permission denied
```

you might have to change the mode of the files to make sure that they are executable. Locate the directory that contains *PyBoolNet 2.1* (see *Installation of PyBoolNet* above) and run:

```
../PyBoolNet$ chmod -R 744 Dependencies/
../PyBoolNet$ chmod -R +x Dependencies/
../PyBoolNet$
```

1.7.3 no such file or directory

If you get *No such file or directory* errors you might have installed the wrong package for your OS. In particular check whether you are on 32 bit or 64 bit Linux and download the respective files from:

- http://github.com/hklarner/PyBoolNet/releases

CHAPTER

TWO

MANUAL

2.1 Importing Boolean Networks

2.1.1 prime implicants

The prime implicants are a unique representation for Boolean networks that serves as a foundation for tasks like computing the interaction graph or state transition graph and computing steady states or trap spaces. See *Klarner2015(a)* for the background. The 1-implicants of a Boolean expression correspond to those clauses in propositional logic that imply that the expression is true while 0-implicants are those clauses that imply that the expression is false. Prime implicants are the shortest implicants, i.e., a clause is prime if removing any literal results in the negation of the original implication.

Consider the expression:

```
v2 & (!v1 | v3)
```

where `&`, `|` and `!` represent conjunction, disjunction and negation, respectively. One of its 1-implicants is:

```
v1 & v2 & v3
```

because:

```
(v1 & v2 & v3) => (v2 & (!v1 | v3))
```

is valid. But it is not prime since removing the literal *v1* is a shorter 1-implicant:

```
(v2 & v3) => (v2 & (!v1 | v3))
```

is also valid. In Python we represent prime implicants as nested dictionaries and lists. The prime implicants of a network with three components *v1*, *v2*, *v3* and three update functions *f1*, *f2*, *f3* that are defined by:

```
f1 := v2 & (!v1 | v3)
f2 := !v3
f3 := v2 | v1
```

is represented by a dictionary, say *primes*, whose keys are the names of the components, here *"v1"*, *"v2"* and *"v3"*. The values of each name are lists of length two that contain the 0 and 1 prime implicants. To access the 1-prime implicants of *v1* use:

```
>>> primes["v1"][1]
[{'v2':1,'v1':0},{'v2':1,'v3':1}]
```

The returned list states that *f1* has two 1-prime implicants and each consists of two literals. Clauses are therefore represented by dictionaries whose keys are names of components and whose values are either 0 or 1, depending on whether the corresponding literal is negative or positive.

It can be difficult to enumerate all prime implicants of a network and *PyBoolNet 2.1* uses the program *BNetToPrime* to do it. As a user you define a network in terms of Boolean expressions, Python functions or you import it from other tools, like GINsim. The steps in each case are explained in the following sections.

2.1.2 states, subspaces and paths

Apart from primes, there are three more fundamental data structures: *states*, *subspaces* and *paths*. A *subspace* is a Python dictionary whose items describe which components are fixed at which level, i.e., the keys are component names and the values are the corresponding activities. A *state* is a special case of a subspace. It contains *n* items where *n* is the number of components. The number of components is usually accessible by:

```
>>> n = len(primes)
```

A *path* is sequence of states represented by a Python iterable, usually a tuple or list.

A state and subspace of the example network above are:

```
>>> state = {"v1":0,"v2":1,"v3":0}
>>> subspace = {"v1":0}
```

States and subspaces may also be defined using string representations, i.e., strings of 0s, 1s and dashes:

```
>>> state = "010"
>>> subspace = "0--"
```

String and dictionary representations may be converted into each other using the functions *state2str*, *state2dict* and *subspace2str*, *subspace2dict*.

A path that consists of two states is for example:

```
>>> x = {"v1":0,"v2":1,"v3":0}
>>> y = {"v1":1,"v2":1,"v3":1}
>>> path = [x,y]
```

2.1.3 primes from BNet files

A *bnet* file contains a single line for every component. Each line consists of the name of the variable that is being defined separated by a comma from the Boolean expression that defines its update function. The network above in *bnet* format is:

```
v1,   v2 & (!v1 | v3)
v2,   !v3
v3,   v2 | v1
```

We chose this syntax for its simplicity and to be compatible with BoolNet, see *Müssel2010*. Save it in a text file called *example01.bnet*. To generate its prime implicants use the function *bnet2primes* of the module *FileExchange*:

```
>>> from PyBoolNet import FileExchange as FEX
>>> primes = FEX.bnet2primes("example01.bnet")
```

Instead of a file name the functions also takes string contents of a *bnet* file:

```
>>> bnet = """
...        v1,  v2
...        v2,  v1
...        """
>>> primes = FEX.bnet2primes(bnet)
```

and a second argument can be used for saving the prime implicants as a *json* file:

```
>>> primes = FEX.bnet2primes("example01.bnet", "example01.primes")
```

Saving prime implicants in a separate file is useful in case the network has many components with high in-degrees. For such networks the computation of all primes might take a long time. Previously saved primes can be read with *read_primes*:

```
>>> primes = FEX.read_primes("example01.primes")
```

Previously generated primes can be saved as *json* files using *write_primes*:

```
>>> FEX.write_primes(primes, "example01.primes")
```

You may also want to save primes as a *bnet* file. To do so use *primes2bnet*:

```
>>> FEX.primes2bnet(primes, "example01.bnet")
```

The module *FileExchange* can also export primes to *bns* and *genysis* files to use as inputs for the tools BNS of *Dubrova2011* and GenYsis of *Garg2008*, namely *primes2bns* and *primes2genysis*.

2.1.4 primes from GINsim files

Open the *zginml* or *ginml* file with *GINsim* and generate a *sbml-qual* file, for example *mapk.sbml*, by clicking:

```
File > Export > SBML-qual > run
```

Generate a *bnet* file from *mapk.sbml* with *BoolNet*:

```
$ R
> library(BoolNet)
> net = loadSBML("mapk.sbml")
> saveNetwork(net, "mapk.bnet")
```

Note: In general, GINsim files define multi-valued networks. If you generate primes from a GINsim file be sure that the underlying network is Boolean.

2.1.5 primes from Python functions

An alternative to defining Boolean networks by Boolean expressions and *bnet* files is to create a Python function for every component. This allows the use of arithmetic and arbitrary Python code to express the conditions under which components are activated or inhibited. Suppose, for example, that a gene *v1* is regulated by four transcription factors *v2,...,v5* and that *v1* is activated iff two or more of them are present. It is tedious to express such a condition in terms of a Boolean expression but easy using the Python function *sum*:

```
>>> f1 = lambda v2,v3,v4,v5: sum([v2,v3,v4,v5])>=2
```

Note that due to Python's typecasting we may use *True* and *False* synonymously for 1 and 0:

```
>>> f1(False, True, True, False)
True
>>> f1(0,1,1,0)
True
```

The lambda construct is convenient for single line definitions but more complex functions can be defined using the standard *def* block:

```
>>> def f2(v1,v2,v3):
...     if f1(v2,v3,0,0):
...         return 0
...     else:
...         return sum([v1,v2,v3]) % 2
```

The definition of *f2* involves the variables *v1,v2,v3* and *f1*: it returns 0 if *f1(v2,v3,0,0)* is 1 and otherwise *v1+v2+v3 mod 2*. Note that *f2* returns 1 and 0 instead of *True* and *False*. Function can also you Python logic operators:

```
>>> f3 = lambda v4,v5: not (v4 or not f2(v4,v4,v5))
```

Constant functions always return 1 or 0 and inputs are only regulated by themselves:

```
>>> f4 = lambda: 1
>>> f5 = lambda v5: v5
```

To generate a primes object from these functions use *functions2primes* of the module *QuineMcCluskey*. Its argument is a dictionary of component names and Boolean functions:

```
>>> from PyBoolNet import QuineMcCluskey as QMC
>>> funcs = {"v1":f1, "v2":f2, "v3":f3, "v4":f4, "v5":f5}
>>> primes = QMC.functions2primes(funcs)
```

In case you want to see a minimal *disjunctive normal form* (DNF) of the functions you defined, use *functions2mindnf*:

```
>>> dnf = QMC.functions2mindnf(funcs)
>>> dnf["v1"]
((v4 & v3) | (v5 & v3) | (v5 & v4) | (v5 & v2) | (v4 & v2) | (v3 & v2))
```

2.2 Drawing the Interaction Graph

Prime implicants can be used to derive the *interaction graph* (IG) of a network. The algorithm is based on the fact that a variable *vi* interacts with a variable *vj* if and only if *vj* has a prime implicant whose conjunction involves a *vi* literal. The interaction is positive if and only if there is a 1-prime with a positive literal *vi* or a 0-prime with a negative literal *not vi*. Similarly, the interaction is negative if and only if there is a 1-prime with a negative literal *not vi* or a 0-prime with a positive literal *vi*. To compute the interaction graph use the function *primes2igraph* of the module *InteractionGraphs*. It returns a directed graph in the *NetworkX* format, that is, a `networkx.DiGraph()` object:

```
>>> from PyBoolNet import InteractionGraphs as IGs
>>> bnet = "\n".join(["v1, v1|v3","v2, 1", "v3, v1&!v2 | !v1&v2"])
>>> primes = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> igraph
<networkx.classes.digraph.DiGraph object at 0xb513efec>
```

The nodes and edges of *igraph* can be accessed via the *NetworkX* functions `edges()` and `nodes()`:

```
>>> igraph.nodes()
['v1', 'v2', 'v3']
>>> igraph.edges()
[('v1', 'v1'), ('v1', 'v3'), ('v2', 'v3'), ('v3', 'v1')]
```

The sign of an interaction is either either positive, negative or both. Signs are stored as the edge attribute *sign* and are accessible via the standard *NetworkX* edge attribute syntax:

```
>>> igraph.edge["v3"]["v1"]["sign"]
set([1])
```

Signs are implemented as Python sets and contain 1 if the interaction is positive and -1 if it is negative or both if the interaction is ambivalent, i.e., sometimes positive and sometimes negative:

```
>>> igraph.edge["v1"]["v3"]["sign"]
set([1, -1])
```

To create a drawing of an interaction graph use the function *igraph2image*:

```
>>> IGs.igraph2image(igraph, "example02_igraph.pdf")
```

It uses *Graphviz* and the layout engine *dot* to create the given image file. The result is shown in *the figure below*.

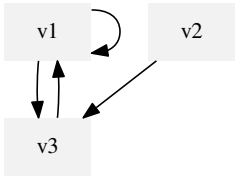


Figure 2.1: The interaction graph "*example02_igraph.pdf*" of the network defined above.

2.2.1 graph, node and edge attributes

PyBoolNet 2.1 generates a *dot* file from an interaction graph by inspecting all its edge, node and graph attributes. Attributes are just dictionaries that are attached to nodes, edges and the graph itself, see *NetworkX* for an introduction. Use these attributes to change the appearance of the graph. The idea is that you either change the appearance of individual nodes and edges using node and edge attributes, or change their default appearance using graph attributes. For a list of all available attributes see

- http://www.graphviz.org/doc/info/attrs.html.

Some node attributes are:

- *shape*: sets the shape of the node, e.g. *"rect"*, *"square"*, *"circle"*, *"plaintext"*, *"triangle"*, *"star"*, *"lpromoter"*, *"rpromoter"*
- *label* (default is the component name): sets the label of a node
- *style*: *"filled"* to fill with a color, *"invis"* to hide or *""* to revert to default
- *fillcolor*: sets the fill color, requires *style="filled"*
- *color*: sets the stroke color
- *fontcolor*: sets the font color
- *fontsize* (default is *14*): sets the font size in pt, e.g. *5*, *10*, *15*
- *fixedsize*: specifies whether the width of a node is fixed, either *"true"* or *"false"*
- *width*: sets the node width, e.g. *5*, *10*, *15*

Colors can be set by names like *"red"*, *"green"* or *"blue"*, or by space-separated HSV values, e.g. *"0.1 0.2 1.0"*, or by a RGB value, e.g *"#40e0d0"*. For a list of predefined color names see for example

- http://www.graphviz.org/doc/info/colors.html.

The basic edge attributes are:

- *arrowhead*: sets the shape of the arrow, e.g. *"dot"*, *"tee"*, *"normal"*
- *arrowsize*: sets the size of the arrow, e.g. *5*, *10*, *15*
- *style*: sets the pen style of the edge, e.g. *"dotted"*, *"dashed"*
- *color*: sets the edge color
- *label*: sets the label of an edge
- *penwidth* (default is *1*): sets the width of an edge, e.g. *5*, *10*, *15*
- *constraint* (default is *"true"*): whether the edge should be included in the calculation of the layout, either *"true"* or *"false"*
- *weight* (default is *1*): sets the cost for stretching the edge during layout computation, for example *"5"*, *"10"*, *"15"*

To set node or edge defaults, add them to the *node* or *edge* attribute of the graph field:

```
>>> bnet = "\n".join(["v1, v2 & (!v1 | v3)","v2, !v3","v3, v2 | v1"])
>>> primes = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> igraph.graph["node"]["shape"] = "circle"
>>> igraph.graph["node"]["color"] = "blue"
```

To change the appearance of specific nodes or edges, add attributes to the node or edge field:

```
>>> igraph.node["v2"]["shape"] = "rpromoter"
>>> igraph.node["v2"]["color"] = "black"
>>> igraph.edge["v3"]["v1"]["arrowhead"] = "inv"
>>> igraph.edge["v3"]["v1"]["color"] = "red"
```

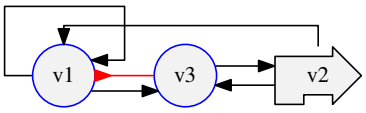
In addition, *dot* graphs have general graph attributes, for example:

- *splines*: sets how edges are drawn, e.g. *"line"*, *"curved"* or *"ortho"* for orthogonal edges
- *label*: adds a label to the graph
- *rankdir* (default is *"TB"*): sets the direction in which layout is constructed, e.g. *"LR"*, *"RL"*, *"BT"*

To change graph attributes, add them to the graph dictionary:

```
>>> igraph.graph["splines"] = "ortho"
>>> igraph.graph["rankdir"] = "LR"
>>> igraph.graph["label"] = "Example 3: Interaction graph with attributes"
>>> IGs.igraph2image(igraph, "example03_igraph.pdf")
```

The result is shown in *the figure below*.



Example 3: Interaction graph with attributes

Figure 2.2: The interaction graph "*example03_igraph.pdf*".

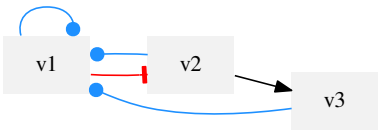
2.2.2 the interaction signs style

PyBoolNet 2.1 has predefined styles for adding attributes to interaction graphs. The function *add_style_interactionsigns* adds or overwrites color and arrowhead attributes to indicate whether an interaction is activating, inhibiting or both. Activating interactions are black with normal arrowheads, inhibiting interactions are red with blunt arrowhead and interactions that are both activating and inhibiting are blue with round arrowheads.

Consider a network with a *exclusive or* regulation:

```
>>> funcs = {"v1":lambda v1,v2,v3: v1+v2+v3==1,
...          "v2":lambda v1: not v1,
...          "v3":lambda v2: v2}
>>> primes = QMC.functions2primes(funcs)
>>> igraph = IGs.primes2igraph(primes)
>>> IGs.add_style_interactionsigns(igraph)
>>> igraph.graph["label"] = "Example 4: Signed interaction graph"
>>> igraph.graph["rankdir"] = "LR"
>>> IGs.igraph2image(igraph, "example04_igraph.pdf")
```

The result is shown in *the figure below*.



Example 4: Signed interaction graph

Figure 2.3: The interaction graph "*example04_igraph.pdf*" with attributes added by *add_style_interactionsings*.

2.2.3 styles for inputs, outputs and constants

Inputs are components that are only regulated by themselves. Usually, inputs regulate themselves positively but we also consider inputs that regulate themselves negatively as inputs. *Outputs* are components that do not regulate other components and *constants* are components whose update function is constant and always returns either *True* or *False*.

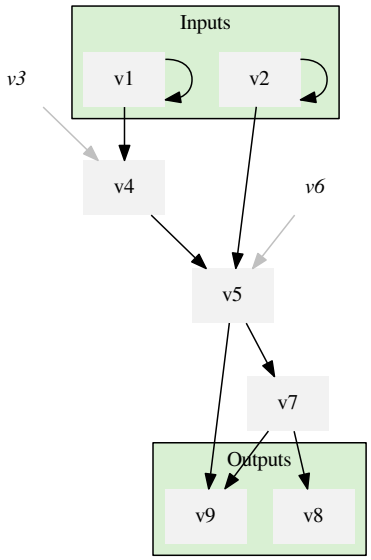
To highlight inputs and outputs by grouping them inside a box use the functions *add_style_inputs* and *add_style_outputs*. They add *dot* subgraphs that contain all components of the respective type and add the label *"inputs"* or *"outputs"*. The function *add_style_constants* changes the shape of constants to *"plaintext"*, their font to *"Time-Italic"* and the color of all interactions involving constants to *"gray"*.

Consider this example:

```
>>> bnet = ["v1, v1", "v2, v2", "v3, 1", "v4, v1 | v3",
...         "v5, v4 & v2 | v6", "v6, 0", "v7, !v5",
...         "v8, v7", "v9, v5 & v7"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> IGs.add_style_inputs(igraph)
>>> IGs.add_style_constants(igraph)
>>> IGs.add_style_outputs(igraph)
>>> igraph.graph["label"] = "Example 5: Interaction graph with styles for"+
```

```
...                          "inputs, outputs and constants"
>>> IGs.igraph2image(igraph, "example05_igraph.pdf")
```

The result is shown in *the figure below*.



Example 5: Interaction graph with styles for inputs, outputs and constants

Figure 2.4: The interaction graph "*example05.pdf*" with styles added by *add_style_inputs*, *add_style_outputs* and *add_style_constants*.

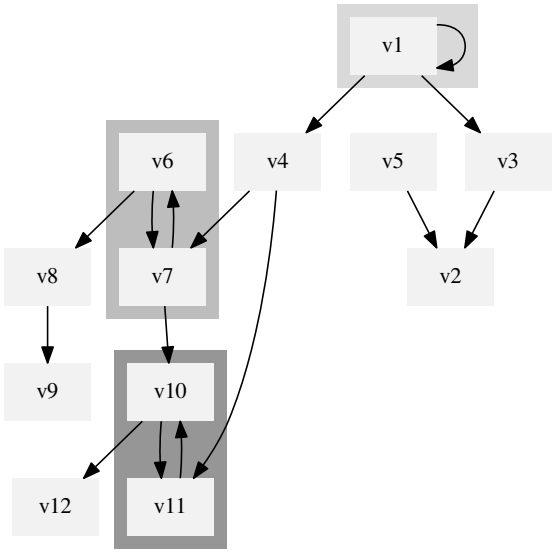
2.2.4 the SCCs style

The function *add_style_sccs* defines a *dot* subgraph for every non-trivial *strongly connected component* (SCC) of the interaction graph. Each SCC subgraph is filled by a shade of gray that indicates the longest path of SCCs leading to it, a unique number that intuitively represents "the depth in the SCC hierarchy", see *Klarner2015(b)* for a formal definition. The further down the hierarchy, the darker the shade of gray will be. Shades of gray repeat after a depth of nine.

Consider the network:

```
>>> bnet = ["v1, v1", "v2, v3 & v5", "v3, v1", "v4, v1", "v5, 1",
...         "v6, v7", "v7, v6 | v4", "v8, v6", "v9, v8", "v10, v7 & v11",
...         "v11, v10 | v4", "v12, v10"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> IGs.add_style_sccs(igraph)
>>> igraph.graph["label"] = "Example 6: Interaction graph with SCC style"
>>> IGs.igraph2image(igraph, "example06_igraph.pdf")
```

The result is shown in *the figure below*.



Example 6: Interaction graph with SCC style

Figure 2.5: The interaction graph "*example06_igraph.pdf*" with attributes added by *add_style_sccs*.

2.2.5 the subgraphs style

The function *add_style_subgraphs* allows you to specify subsets of nodes that will be added to a *dot* subgraph. The subgraphs may be specified as a list of pairs that consist of a list of names and a dictionary of *dot* attributes for that subgraph, e.g., a label or background color.

Note: *Subgraphs* must satisfy this property: Any two subgraphs have either empty intersection or one is a subset of the other. The reason for this requirement is that *dot* can not draw intersecting subgraphs.

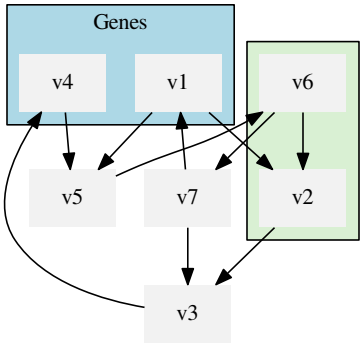
Consider the network:

```
>>> bnet = ["v1, v7", "v2, v1 & v6", "v3, v2 | v7", "v4, v3",
...         "v5, v1 | v4", "v6, v5", "v7, v6"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> subgraphs = [(["v2","v6"],{}),
...              (["v1","v4"],{"label":"Genes", "fillcolor":"lightblue"})]
>>> IGs.add_style_subgraphs(igraph, subgraphs)
>>> igraph.graph["label"] = "Example 8: Interaction graph with a subgraph style"
>>> IGs.igraph2image(igraph, "example08_igraph.pdf")
```

The result is shown in *the figure below*.

2.2.6 the activities style and animations

The function *add_style_activities* colors components according to a given dictionary of activities, i.e., a state or subspace. Components that are active are colored in red, inactive ones blue and the attributes of the remaining components



Example 8: Interaction graph with a subgraph style

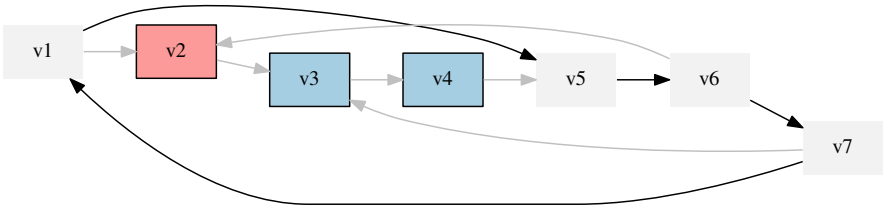
Figure 2.6: The interaction graph "*example08_igraph.pdf*" with attributes added by *add_style_subgraphs*.

are not changed. In addition, interactions that involve activated or inhibited components are grayed out to reflect that they are ineffective.

Here is an example:

```
>>> bnet = ["v1, v7", "v2, v1 & v6", "v3, v2 | v7", "v4, v3",
...         "v5, v1 | v4", "v6, v5", "v7, v6"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> activities = {"v2":1, "v3":0, "v4":0}
>>> IGs.add_style_activities(igraph, activities)
>>> igraph.graph["label"] = "Example 9: Interaction graph with a activities style"
>>> igraph.graph["rankdir"] = "LR"
>>> IGs.igraph2image(igraph, "example09_igraph.pdf")
```

The result is shown in *the figure below*.



Example 9: Interaction graph with a activities style

Figure 2.7: The interaction graph "*example09_igraph.pdf*" with attributes added by *add_style_activities*.

You can also create an animated *gif* from an interaction graph and a sequence of activities. Note that as mentioned in *states, subspaces and paths*, activities may be given as strings that consist of 0s, 1s and dashes using the function *activities2animation*:

```
>>> activities = ["-100", "-110", "-010"]
>>> IGs.activities2animation(igraph, activities, "animation.gif")
```

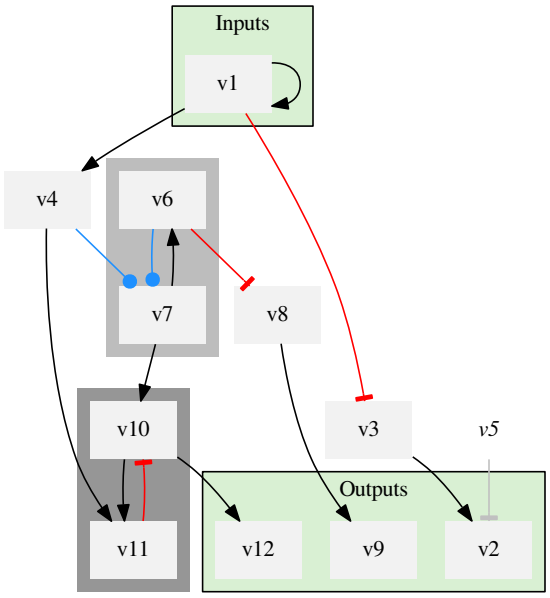
2.2.7 the default style

The default style combines the SCCs, inputs, outputs, constants and interaction sign styles.

Consider the network:

```
>>> bnet = ["v1, v1", "v2, v3 & !v5", "v3, !v1", "v4, v1", "v5, 1",
...         "v6, v7", "v7, v6 & !v4 | !v6 & v4", "v8, !v6", "v9, v8", "v10, v7 & !v11",
...         "v11, v10 | v4", "v12, v10"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> IGs.add_style_default(igraph)
>>> igraph.graph["label"] = "Example 10: Interaction graph with default style"
>>> IGs.igraph2image(igraph, "example10_igraph.pdf")
```

The result is shown in *the figure below*.



Example 10: Interaction graph with default style

Figure 2.8: The interaction graph "*example10_igraph.pdf*" with attributes added by *add_style_default*.

2.3 Drawing the State Transition Graph

Prime implicants can be used to derive the *state transition graph* (STG) of a network. To compute it, use the function *primes2stg* of the module *StateTransitionGraphs*. It returns an instance of the *NetworkX* digraph class:

```
>>> from PyBoolNet import StateTransitionGraphs as STGs
>>> bnet = "\n".join(["v1, v3","v2, v1", "v3, v2"])
>>> primes = FEX.bnet2primes(bnet)
>>> update = "asynchronous"
>>> stg = STGs.primes2stg(primes, update)
>>> stg
<networkx.classes.digraph.DiGraph object at 0xb3c7d64c>
```

The second argument to *primes2stg* is either *"synchronous"* or *"asynchronous"* for the fully synchronous or the fully asynchronous transition relation, see e.g. *Klarner2015(b)* for a formal definition. The nodes of an STG are string representations of states, e.g. *"110"*, see *states, subspaces and paths*. You may use *state2str* to convert a state dictionary into a state string. They are vectors of activities, sorted by component names:

```
>>> stg.nodes()[0]
'010'
```

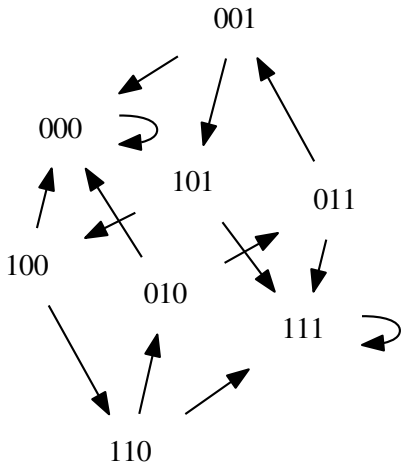
You may use *NetworkX* functions on *stg*, for example networkx.has_path:

```
>>> import networkx
>>> networkx.has_path(stg, "100", "111")
True
```

State transition graphs can be styled in the same way as interaction graphs, see *Drawing the Interaction Graph*. Use the function *stg2image* to generate a drawing of the STG:

```
>>> stg.graph["label"] = "Example 11: The asynchronous STG of a positive circuit"
>>> stg.graph["rankdir"] = "LR"
>>> STGs.stg2image(stg, "example11_stg.pdf")
```

The result is shown in *the figure below*.



Example 11: The STG of a positive circuit

Figure 2.9: The state transition graph "*example11_stg.pdf*" of an isolated feedback circuit.

By default, the full STG is constructed. If you want to draw the part of a STG that is reachable from an initial state or a set of initial states pass a third argument to *primes2stg*. For convenience you may choose one of several ways of specifying initial states. Either a list of states in *dict* or *str* format, see *states, subspaces and paths*:

```
>>> init = ["000", "111"]
>>> init = ["000", {"v1":1,"v2":1,"v3":1}]
```

or as a function that is called on every state and must return either *True* or *False* to indicate whether the state ought to be initial:

```
>>> init = lambda x: x["v1"]>=x["v2"]
```

or by a subspace in which case all the states contained in it are initial:

```
>>> init = "--1"
>>> init = {"v3":1}
```

To construct the STG starting from initial states call:

```
>>> stg = STGs.primes2stg(primes, update, init)
```

Warning: You should not draw asynchronous STGs with more than 2^7=128 states as *dot* will take very long to compute the layout. For synchronous STGs you should not go above 2^12=4096 states. Use different layout engines like *twopi* and *circo* by generating the *dot* file with *stg2dot* and compiling it manually.

2.3.1 the tendencies style

The tendencies style for state transition graphs is similar to the interaction sign style for interaction graphs. It colors state transitions according to whether all changing variables increase (black), or all of them decrease (red) or some increase and some decrease (blue). The latter is only possible for synchronous transition graphs.

Here is an example:

```
>>> bnet = "\n".join(["v1, !v3","v2, v1", "v3, v2"])
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "synchronous")
>>> stg.graph["rankdir"] = "LR"
>>> stg.graph["label"] = "Example 12: The synchronous STG of a negative circuit"
>>> STGs.add_style_tendencies(stg)
>>> STGs.stg2image(stg, "example12_stg.pdf")
```

The result is shown in *the figure below*.

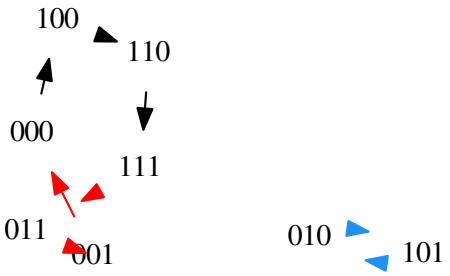
2.3.2 the path style

The path style is used to highlight a path in the state transition graph. It changes the *penwidth* and *color* of transitions.

Consider the following example:

```
>>> bnet = "\n".join(["x, !x|y", "y, !x&!z|x&!y&z", "z, x|!y"])
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 13: STG with path style"
```

Now add the path style:

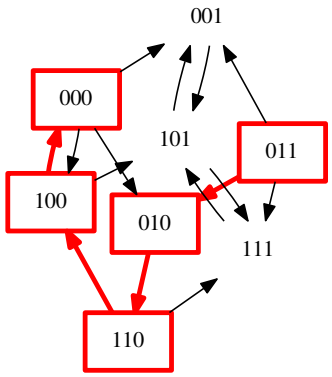


Example 12: The synchronous STG of a negative circuit

Figure 2.10: The state transition graph "*example12_stg.pdf*" with attributes added by *add_style_tendencies*.

```
>>> path = ["011","010","110","100","000"]
>>> STGs.add_style_path(stg, path, "red")
>>> STGs.stg2image(stg, "example13_stg.pdf")
```

The result is shown in *the figure below*.



Example 13: STG with path style

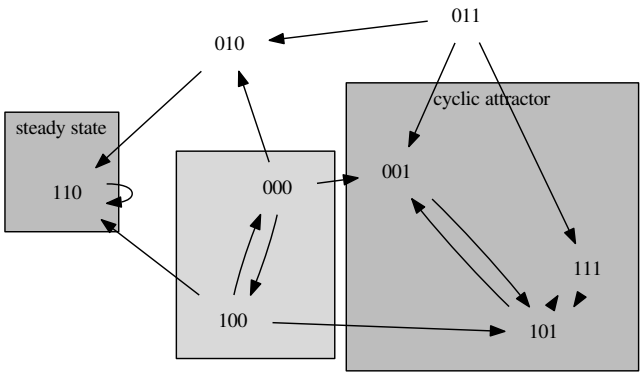
Figure 2.11: The state transition graph "*example13_stg.pdf*" with attributes added by *add_style_path*.

2.3.3 the SCCs style

The SCC style is almost identical to the one for interaction graphs except that it adds a label to the attractors, i.e., steady states and cyclic attractors.:

```
>>> bnet = "\n".join(["x, !x|y", "y, x&!y|!z", "z, x&z|!y"])
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "The SCC style"
>>> STGs.add_style_sccs(stg)
>>> STGs.stg2image(stg, "example14_stg.pdf")
```

The result is shown in *the figure below*.



Example 14: STG with SCC style

Figure 2.12: The state transition graph "*example14_stg.pdf*" with attributes added by *add_style_sccs*.

2.3.4 the min trap spaces style

The min trap spaces style is adds a *dot* subgraph for every minimal trap space of the state transition graph. For an introduction to trap spaces, see *Klarner2015(a)* and also *trap_spaces_and_attractors*:

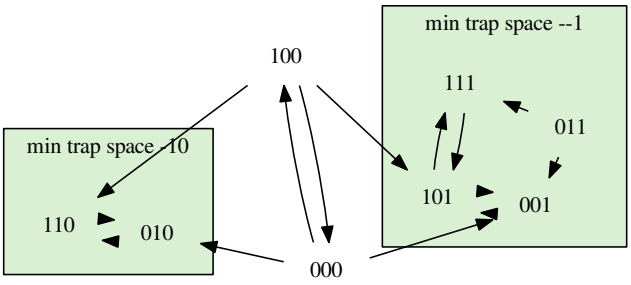
```
>>> bnet = "\n".join(["x, !x|y&z", "y, x&!y|!z", "z, z|!y"])
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 15: STG with min trap spaces style"
>>> STGs.add_style_mintrapspaces(primes, stg)
>>> STGs.stg2image(stg, "example15_stg.pdf")
```

The result is shown in *the figure below*.

2.3.5 the subspaces style

The subspace style is identical to the subgraph style of interaction graphs. It adds a subgraph for every given subspace. As for interaction graphs, you may add pairs of subspace and attribute dictionaries if you want to change the label, or color etc. of the subgraphs:

```
>>> bnet = "\n".join(["x, !x|y&z", "y, x&!y|!z", "z, z|!y"])
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 16: STG with subspaces style"
```



Example 15: STG with min trap spaces style

Figure 2.13: The state transition graph "*example15_stg.pdf*" with attributes added by *add_style_mintrapspaces*.

```
>>> sub1 = ({"x":0},{"label":"x is zero"})
>>> sub2 = {"x":1,"y":0}
>>> subspaces = [sub1, sub2]
>>> STGs.add_style_subspaces(primes, stg, subspaces)
>>> STGs.stg2image(stg, "example16_stg.pdf")
```

The result is shown in *the figure below*.

Note: *Subspaces* must satisfy this property: Any two subspaces have either empty intersection or one is a subset of the other. The reason for this requirement is that *dot* can not draw intersecting subgraphs.

2.3.6 the default style

The default style combines the SCCs with the tendencies and the minimal trap spaces styles:

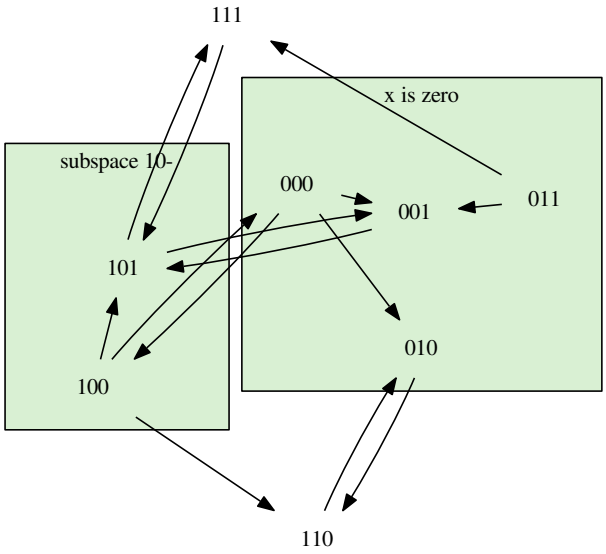
```
>>> bnet = "\n".join(["x, !x|y&z", "y, x&!y|!z", "z, z|!y"])
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 16: STG with default style"
>>> STGs.add_style_default(primes, stg)
>>> STGs.stg2image(stg, "example17_stg.pdf")
```

The result is shown in *the figure below*.

2.4 Modifying Networks

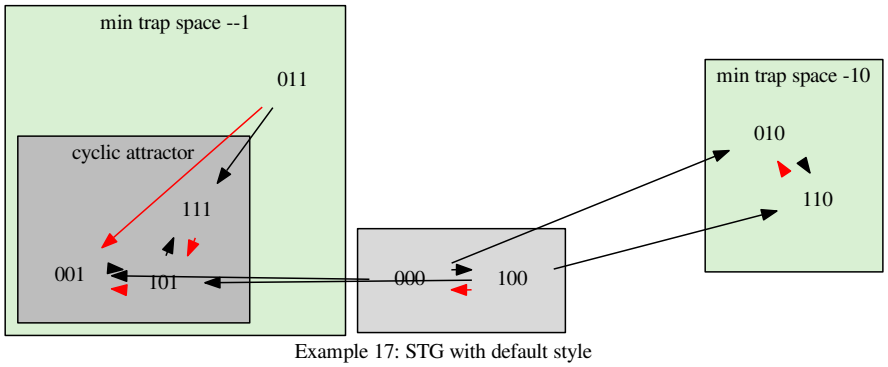
2.4.1 constant, inputs and blinkers

There are various reasons why it may be required to modify an imported Boolean network, i.e., a primes dictionary. A typical example is when the goal is to enumerate a number of variations of a given network structure in order to collect those that satisfy a given specification, i.e., a model checking query. Functions for the modification of networks are contained in the module *PrimeImplicants*. Typically, these functions either find something, e.g. *find_inputs*, or create something, e.g. *create_constants* or remove something, e.g. *remove_variables*. But there are also functions that percolate values, enumerate input combinations and replace update functions.



Example 16: STG with subspaces style

Figure 2.14: The state transition graph *"example16_stg.pdf"* with attributes added by *add_style_subspaces*.



Example 17: STG with default style

Figure 2.15: The state transition graph *"example17_stg.pdf"* with attributes added by *add_style_default*.

As an example consider the task of replacing all constant nodes by so-called *blinkers*, i.e., variables that are negatively auto-regulated and are therefore repetatively changing their activity from *On* to *Off* back to *On*, and so on. A node v1 is constant if in the *bnet* file it is defined by either 0 or 1, e.g.:

```
v1,   0
```

Note that such a node is not an input. A node v2 is an input iff:

```
v2,   v2
```

The difference is also visible in the interaction graph where constants have in-degree 0 and input are only regulated by themselves and the regulation is positive. Finally, a blinker is like an input but with negative auto-regulation, e.g. v3 is a blinker iff:

```
v3,   !v3
```

To replace all constants by blinker first we first need the names of the constants. If they are not known beforehand they may be computed using the function *find_constants*. To create the blinkers use the function *create_blinkers*:

```
>>> from PyBoolNet import PrimeImplicants as PIs
>>> bnet = """
... v1,   0
... v2,   1
... v3,   v1&v2&v3&v4
... v4,   v3 & (v1|v2)"""

>>> primes = FEX.bnet2pirmes(bnet)
>>> names = PIs.find_constants(primes)
>>> names
['v1','v2']
>>> PIs.create_blinkers(primes, names)
>>> FEX.primes2bnet(primes)
v1,   !v1
v2,   !v2
v3,   v1 & v2 & v3 & v4
v4,   v2 & v3 | v1 & v3
```

Note that *PyBoolNet 2.1* modifies the primes object in place rather than creating and returning a modified copy. If you want to keep the original primes and modify a copy you have to create the copy explicitly:

```
>>> newprimes = PIs.copy(primes)
>>> PIs.create_inputs(newprimes, names)
```

Components may be renamed using the function *rename_variable*, e.g.

```
>>> PIs.rename_variable(primes, "v1", "x")
>>> FEX.primes2bnet(primes)
x,    !x
v2,   !v2
v3,   x & v2 & v3 & v4
v4,   v2 & v3 | x & v3
```

2.4.2 percolating constants

A frequently used step in model analysis and model reduction is to compute the set of variables *that will become constant* due the constants already in the model. We call the network obtained by replacing the update functions of the new constants be the respective constant values the *percolated network* because we imagine the values to "trickle through" along cascades in the interaction graph where the original constants are at the top. Consider this example:

```
>>> bnet = """
... v1,   0
... v2,   v2
... v3,   !v1 | v2"""
```

Although v3 is not a constant its update function will be constant at 1 once v1 has attained its constant value of 0. We say that the value of v1 percolates to v3, that is, determines the value of v3 in the long term. Networks with a lot of constants are easier to analyse and understand as these nodes can, for example, be discarded for many model checking queries. There are two functions for computing percolated networks: *percolate_and_keep_constants* and *percolate_and_remove_constants*. The second one removes all variables from the primes dict that became constant during the percolation while the second one keeps them. Both functions return a dictionary of constants. Keeping the constants results in:

```
>>> primes = FEX.bnet2pirmes(bnet)
>>> constants = PIs.percolate_and_keep_constants(primes)
>>> constants
{'v1':0,'v3':1}
>>> FEX.primes2bnet(primes)
v1,   0
v2,   v2
v3,   1
```

Here, v1 and v3 are kept in the model. Removing the constants results in:

```
>>> primes = FEX.bnet2pirmes(bnet)
>>> constants = PIs.percolate_and_remove_constants(primes)
>>> constants
{'v1':0,'v3':1}
>>> FEX.primes2bnet(primes)
v2,   v2
```

Here, the constants v1 and v3 are removed.

2.4.3 removing, adding and creating variables

You can not, in general, remove variables from a model because other variables may depend on the one you want to remove. In the example network below, how would you define the network obtained by removing v1 from it?:

```
v1,   !v1 | v2
v2,   v2 & v1
v3,   v1 & v2 & v3
```

Clearly, you can not simply remove the definition of v1 because:

```
v2,   v2 & v1
v3,   v1 & v2 & v3
```

is not well-defined, since v3 depends on a variable that is not specified. But, you may remove v3 and the result is a well-defined network:

```
v1,   !v1 | v2
v2,   v2 & v1
```

In general, you may remove variables that are *closed under the successor relation* in the interaction graph. That is, any set of variables that contains all its successors may be safely removed. There are two functions for removing variables depending on whether you specify the names of variables to keep or to remove: *remove_variables* and *remove_all_variables_except*. Both functions raise an exception if you try to remove a set of variables that is not closed under the successor relation. Example:

```
>>> bnet = """
... v1,   !v1 | v2
... v2,   v2 & v1
... v3,   v1 & v2 & v3"""
>>> primes = FEX.bnet2pirmes(bnet)
>>> PIs.remove_variables(primes, ["v3"])
>>> FEX.primes2bnet(primes)
v1,   !v1 | v2
v2,   v2 & v1
```

To add a variable use the function *create_variables*. The update functions of new variables may either be specified as *bnet* strings or as Python function with correctly named parameters, see *primes from Python functions* for details on using Python functions to define variables. This function can also be used to modify existing variables as it replaces update functions if they already exist. The function raises an exception if the resulting network contains variables whose update functions are undefined. Example of correct use:

```
>>> primes = FEX.bnet2primes("v1, v2 \n v2, v1")
>>> create_variables(primes, {"v3": "!v4 | v1", "v4": lambda v1,v2: v1+v2==1})
>>> primes FEX.primes2bnet(primes)
v1, v2
v2, v1
v3, !v4
v4, v1&!v2 | !v1&v2
```

An example of violating the condition that all variables must be defined is:

```
>>> primes = FEX.bnet2primes("v1, v1")
>>> create_variables(primes, {"v2":"v3 | v4", "v3":"!v1"})
error: can not add variables that are dependent on undefined variables.
```

2.4.4 input combinations

To enumerate all possible input combinations of a given network use the function *input_combinations*:

```
>>> primes = FEX.bnet2primes("input1, input1 \n input2, input2")
>>> create_variables(primes, {"v1": "input1 & input2"})
>>> create_variables(primes, {"v2": "input1 | input2"})
>>> for x in input_combintations:
...     print x
{'input1':0,'input2':0}
{'input1':1,'input2':0}
{'input1':0,'input2':1}
{'input1':1,'input2':1}
```

2.5 Model Checking

PyBoolNet 2.1 uses *NuSMV* to decide model checking queries for Boolean networks. A model checking problem is defined by a transition system, its initial states and a temporal specification. For a formal introduction to model checking see for example *Baier2008*.

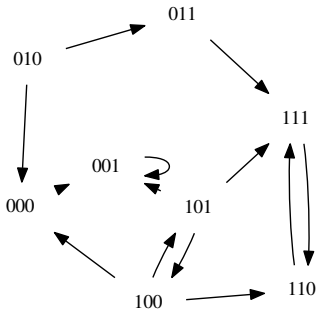
2.5.1 transition systems

Transition systems are very similar to state transition graphs but in addition to states and transitions there are *atomic propositions* which are the statements that are available for specifying states. As an example, consider the following network:

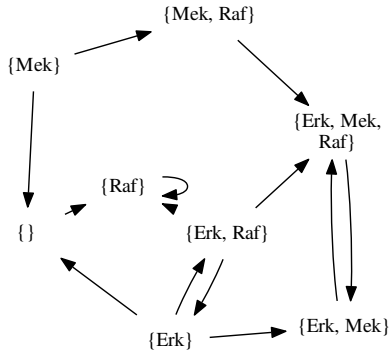
```
>>> bnet = ["Erk,  Erk & Mek | Mek & Raf",
...         "Mek,  Erk | Mek & Raf",
...         "Raf,  !Erk | !Raf"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 18: STG of the Erk-Mek-Raf network"
>>> STGs.stg2image(stg, "example18_stg.pdf")
```

The state transition graph is shown in *the figure below*.

When model checking, *PyBoolNet 2.1* translates state transition graphs into transition systems. The basic approach to doing so is shown in *the figure below*. Here, each state string is replaced by a subset of atomic propositions. The subset is chosen to correspond with the state string, i.e., a state is labeled with *Mek* iff Mek is activated in it which is the case for all states in the subspace *"-1-"*.



Example 18: STG of the Erk-Mek-Raf network



Example 19: Basic transition system of Erk-Mek-Raf

Figure 2.16: The state transition graph *"example18_stg.pdf"* of the Erk-Mek-Raf network on the left and the corresponding basic transition system on the right.

Since the choice of atomic propositions affects the expressiveness and conciseness of the model checking queries that users can formulate we have decided to extend this basic transition system by some *auxiliary variables*. First, we add a proposition that states whether a variable is steady, i.e., whether its activity is equal to the value of its update function. Those propositions add *_STEADY* to each variable, e.g. *Mek_STEADY* for *Mek*. Second, we add a proposition *STEADYSTATE* that is true iff the respective state is a steady state. Finally, we add a proposition *SUCCESSORS=k* where *k* is an integer, that is true iff the respective state has exactly *k* successors (excluding itself). The propositions *SUCCESSORS=0* and *STEADYSTATE* are therefore equivalent.

Note: The *NuSMV* language is case sensitive.

The transition system with the extended set of atomic propositions is shown in *the figure below*.

2.5.2 LTL model checking

Apart from a transition system, a model checking problem requires a *temporal specification*. Since *PyBoolNet 2.1* uses *NuSMV* for solving model checking problems, two specification languages are available: *linear time logic* (LTL)

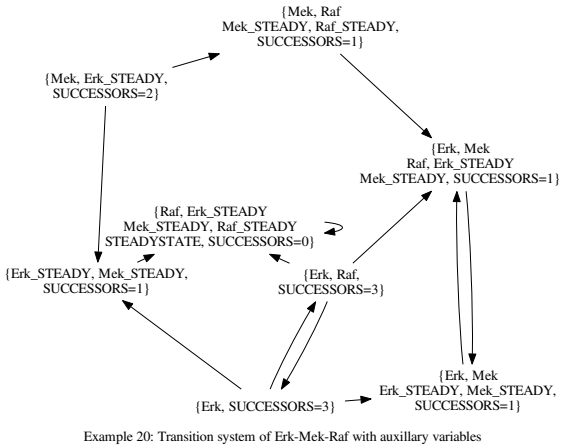


Figure 2.17: The extended transition system for the Erk-Mek-Raf network.

and *computational tree logic* (CTL).

LTL specifications are statements about the sequence of events that are expressed in terms of atomic propositions and temporal operators. A LTL specification is either true or false for a given linear sequence, i.e., infinite path, in a given transition system. The basic temporal operators for LTL are:

- *F(..)* which means *finally*
- *G(..)* which means *globally*
- *[..U..]* which means *until*
- *X(..)* which means *next*

LTL statements may be combined by the usual logical operators which are:

- | which means *disjunction*
- *&* which means *conjunction*
- *!* which means *negation*

in *NuSMV* syntax. For a formal definition of LTL formulas see for example *Baier2008*.

Finally, model checking problems allow the user to specify some states of the transition system to be *initial*. A LTL specification is then defined to be true for a transition system with initial states iff every path that starts from an initial state satisfies the LTL specification.

As an example consider again the Erk-Mek-Raf network *from above*. Let us query whether along every path in its transition system there is eventually a state in which *Raf* is activated:

```
>>> from PyBoolNet import ModelChecking as MC
>>> init = "INIT TRUE"
>>> spec = "LTLSPEC F(Raf)"
>>> update = "asynchronous"
>>> answer = MC.check_primes(primes, update, init, spec)
>>> answer
True
```

The first line imports the module *ModelChecking*. The next line defines the initial states in *NuSMV* syntax with the keyword *INIT* to indicate an initial condition and the expression *TRUE* which evaluates to true in every state. The next

line starts with the keyword *LTLSPEC* which must precede the definition of a LTL specification and the formula *F(Raf)* which states that eventually a state will be reached that is labeled by *Raf*, i.e., in which *Raf* is activated. The fifth line calls the function *check_primes* which constructs the extended transition system and uses *NuSMV* to answer model checking queries. Note that the function requires a parameter that specifies the update rule, i.e., either *"asynchronous"*, *"synchronous"* or *"mixed"* and that it returns a Boolean value.

Even for this small example network it is not trivial to see why *True* is the correct answer, because a brute force approach would require the enumeration of all paths but the transition system contains an infinite number of paths. Convince yourself that every path eventually reaches the state 101 or the state 111 or the state 001. In all cases *Raf*, which is the third digit in the state string, is equal to 1 which is what *F(Raf)* requires. Hence *True* is the correct answer.

The second example is a slightly more complicated *reachability* query:

```
>>> spec = "LTLSPEC F(Raf & F(STEADYSTATE))"
>>> answer = MC.check_primes(primes, update, init, spec)
>>> answer
False
```

The LTL formula queries whether every path will eventually come across a state in which *Raf* is activated followed by a steady state. Note that the formula asserts an order on the sequence of events: first *Raf* and then *STEADYSTATE*. To see why the specification is false we only need to find one infinite path from an initial state that does not satisfy the LTL formula. Since all states are initial the following path will do:

```
101 -> 100 -> 110 -> 111 -> 110 -> 111 -> 110 -> ...
```

The last two states, 111 and 110, are repeated for ever and neither is labeled with *STEADYSTATE* in the extended transition system, see *this figure*. Hence *False* is the correct answer.

The third example specifies a proper subset of states as initial and queries the existence of *sustained oscillations* in *Raf*:

```
>>> init = "INIT Erk & SUCCESSORS<2"
>>> spec = "LTLSPEC G(F(Raf) & F(!Raf))"
>>> answer = MC.check_primes(primes, update, init, spec)
>>> answer
True
```

Here, a state is initial iff *Erk* is activated in it and the number of its successors - with respect to the given the update rule - is less than two. The formula *G((F(Raf) & F(!Raf))* requires that however far down the sequence of states, i.e., *globally*, it is true that *Raf* will eventually be activated and also that *Raf* will eventually be inhibited. The extended transition system, see *this figure*, shows that exactly three state are initial: 110, 011 and 111. Any path starting in one of those state will eventually end in the infinite sequence:

```
111 -> 110 -> 111 -> 110 -> 111 -> ...
```

Hence, any path that starts in one of the initial states satisfies *G((F(Raf) & F(!Raf))*, i.e., a sustained oscillation in *Raf*, and hence the truth of the query.

The fourth example involves another feature: the use of *NuSMV* built-in functions, in this case *count*:

```
>>> init = "INIT Mek"
>>> spec = "LTLSPEC G(count(Erk_STEADY,Mek_STEADY,Raf_STEADY)>=2)"
>>> answer = MC.check_primes(primes, update, init, spec)
>>> answer
False
```

The LTL formula also uses the auxiliary variables *Erk_STEADY*, *Mek_STEADY* and *Raf_STEADY* which are true in states in which the respective variables are equal to the values of their update functions. The formula states that along any path that starts from an initial state at least two of the variables *Erk*, *Mek* and *Raf* are steady. Since the query is false there must be a path that does not satisfy the specifications, for example this one:

```
010 -> 011 -> 111 -> 110 -> 111 -> 110 -> ...
```

It does not satisfy the LTL formula because in the state 010 only *Erk* is steady and hence *count(...)* which counts the number of true expressions is equal to one and hence *G(count(...)>=2)* is false. See the *NuSMV* manual for more built-in functions like *count()*.

The existence of so-called *counterexamples* is essential to LTL model checking and *NuSMV* can be asked to return one if it finds one.

2.5.3 LTL counterexamples

If a LTL query is false then *NuSMV* can return a finite path that proves that the formula is false.

Note: Since the transition systems of Boolean networks are finite, a counterexample will always be a finite sequence of states - possibly ending in a cycle. For a justification, see for example *Baier2008* Sec. 5.2.

To return a counterexample use the function *check_primes_with_counterexample*. The function returns the answer and a counterexample. Reconsider the following query, which we know is false, from above:

```
>>> init = "INIT TRUE"
>>> spec = "LTLSPEC F(Raf & F(STEADYSTATE))"
```

To retrieve the answer and a counterexample call:

```
>>> answer, counterex = MC.check_primes_with_counterexample(primes, update, init, spec)
```

The counterexample is a tuple of state dictionaries (recall *states, subspaces and paths*) if the query is false and *None* in case it is true (in which case no counterexample exists). Hence, a typical way to inspect a counterexample involves a Python if-statement:

```
>>> if counterex:
...     print " -> ".join(STGs.state2str(x) for x in counterex)
100 -> 101 -> 100
```

Here, *state2str* is a "pretty print" function contained in the module *StateTransitionGraphs*. It generates a state string from a state dictionary. An alternative way of inspecting counterexample is by *STGs.add_style_path*:

```
>>> stg = STGs.primes2stg(primes, update)
>>> STGs.add_style_path(stg, counterex, "red")
>>> stg.graph["label"] = "Example 19: A LTL counterexample"
>>> STGs.stg2image(stg, "example19_stg.pdf")
```

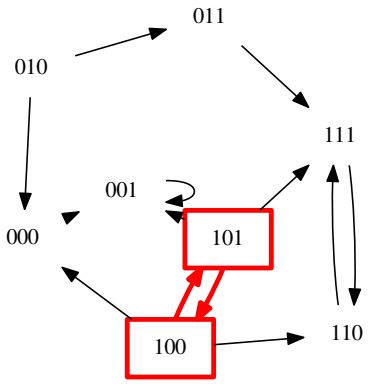
A second alternative is to generate an animated *gif* of the changing activities in each state and using *IGs.activities2animation*:

```
>>> igraph = IGs.primes2igraph(primes)
>>> IGs.activities2animation(igraph, counterex, "counterexample.gif")
```

2.5.4 CTL model checking

NuSMV can also solve model checking problems that involve *computation tree logic* (CTL). CTL formulas are constructed like LTL formulas but the temporal operators *F*, *G*, *X* and *U* must be quantified by *E* which means *for some path* or *A* which means *for all paths*. A CTL formula is not evaluated for paths but for trees of successors rooted in some initial state.

Note: Some properties can be specified in LTL or CTL, other properties can only be stated in either LTL or CTL. See



Example 19: A LTL counterexample

Figure 2.18: The state transition graph *"example18_stg.pdf"* of the Erk-Mek-Raf network with a path style that indicates a counterexample to the LTL query with all states being initial and the formula *F(Raf & F(STEADYSTATE))*.

Sec. 6.3 in *Baier2008* for a discussion of the expressiveness of CTL and LTL.

Consider the following toy model of cell proliferation:

```
>>> bnet = ["GrowthFactor,  0",
...         "Proliferation, GrowthFactor | Proliferation & !DNAdamage",
...         "DNAdamage,     !GrowthFactor & DNAdamage"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> update = "asynchronous"
```

Suppose we want to find out whether the presence of *GrowthFactor* implies the possibility of *Proliferation*. By "possibility" we mean that there is a path that leads to a state in which proliferation is activated. Let us first determine whether this property holds in the network above by drawing the state transition graph with the initial states and the proliferation states highlighted by filled rectangles and a subgraph, respectively:

```
>>> stg = STGs.primes2stg(primes, update)
>>> for x in stg.nodes():
...     x_dict = STGs.state2dict(primes, x)
...     if x_dict["GrowthFactor"]:
...         stg.node[x]["style"] = "filled"
...         stg.node[x]["fillcolor"] = "gray"
>>> sub = ({"Proliferation":1},{"label":"proliferation"})
>>> STGs.add_style_subspaces(stg, [sub])
>>> stg.graph["label"] = "Example 20: STG of the Proliferation network"
>>> STGs.stg2image(stg, "example20_stg.pdf")
```

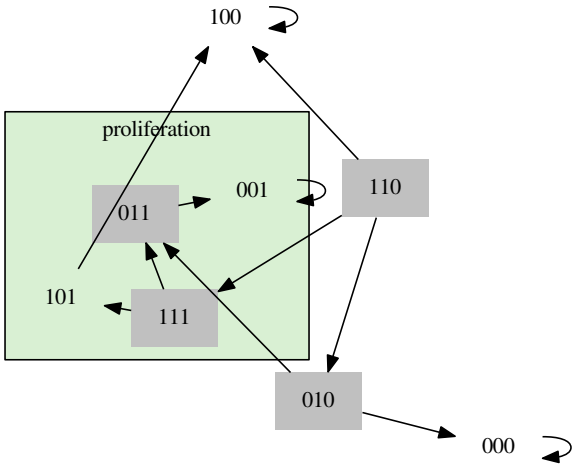
It is easy to see, in the *figure below*, that for every initial state there is a path to a proliferation state. There are two initial states in which *Proliferation* is inhibited, namely *110* and *010*. For each there is a path leading to a state in which *Proliferation* is activated, namely:

```
110 -> 111 and 010 -> 011
```

Perhaps surprisingly, this property can not be formulated in LTL. The LTL formula is *F(Proliferation)*, for example, requires that *all paths* lead to a proliferation state which is not the same as *some paths* lead to proliferation. In fact,

the property *F(Proliferation)* is false, as *the figure below* for the following counterexample demonstrates:

```
>>> init = "INIT GrowthFactor"
>>> spec = "LTLSPEC F(Proliferation)"
>>> answer, counterex = MC.check_primes_with_counterexample(primes, update, init, spec)
>>> answer
False
>>> STGs.add_style_path(stg, counterex, "red")
>>> stg.graph["label"] = "Example 21: Counterexample"
>>> STGs.stg2image(stg, "example21_stg.pdf")
```



Example 20: STG of the Proliferation network

Figure 2.19: The state transition graph *"example20_stg.pdf"* of the Proliferation network with initial states highlighted by gray rectangles and proliferation states gathered in a subgraph.

The property can, however, be formulated in CTL using the existential quantifier:

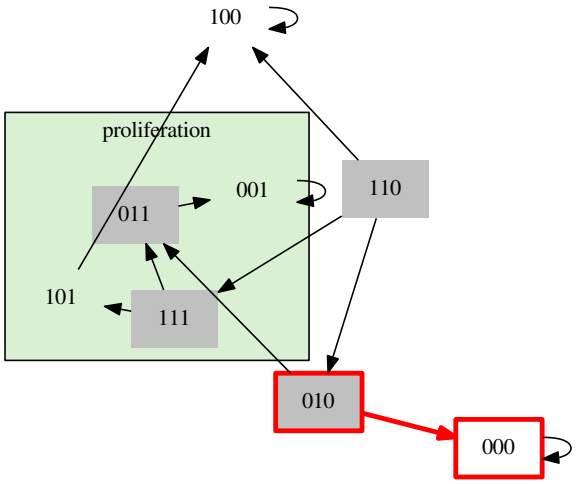
```
>>> spec = "CTLSPEC EF(Proliferation)"
>>> answer = MC.check_primes(primes, update, init, spec)
True
```

Note: The LTL formula *F(Proliferation)* is equivalent to the CTL formula *AF(Proliferation)*. In general, however, there are LTL formulas for which no equivalent CTL formula exists, and vice versa.

CTL model checking is also required when querying properties about the *attractors* of the state transition graph. Attractors are defined to be the *terminal SCCs* of the STG or, equivalently, they are its *minimal trap sets*. For a formal definition see for example *Klarner2015(b)* Sec. 2.2.

Suppose we want to find out whether, for the initial states defined *Proliferation*, all attractors are located in the subset of states that are defined by *!DNAdamage*. In English, this property states that "along any path starting from any initial state it is possible to reach a state from which all reachable states satisfy *!DNAdamage*". In CTL, it can be formulated using the *AG(EF(AG(...)))* query pattern where "..." is the condition that describes the attractor states:

```
>>> init = "INIT Proliferation"
>>> condition = "!DNAdamage"
>>> spec = "CTLSPEC AG(EF(AG(%s)))"%condition
```



Example 21: Counterexample

Figure 2.20: The state transition graph *"example21_stg.pdf"* of the Proliferation network with a counterexample highlighted by path.

```
>>> answer = MC.check_primes(primes, update, init, spec)
True
```

Other frequently used conditions are *STEADYSTATE* to query whether all attractors are steady states:

```
>>> init = "INIT Proliferation"
>>> condition = "STEADYSTATE"
>>> spec = "CTLSPEC AG(EF(AG(%s)))"%condition
>>> answer = MC.check_primes(primes, update, init, spec)
True
```

or disjunctions and conjunctions of basic propositions:

```
>>> init = "INIT Proliferation"
>>> condition = "STEADYSTATE | (!Proliferation & DNAdamage)"
>>> spec = "CTLSPEC AG(EF(AG(%s)))"%condition
>>> answer = MC.check_primes(primes, update, init, spec)
True
```

Note: The CTL formula *AG(EF(AG(STEADYSTATE)))* is equivalent to *AG(EF(STEADYSTATE)* because if a steady is steady then it has no successors.

Note: To query whether *there is* an attractor of a certain type reachable from every initial state, rather than whether *all* attractors are of a certain type, use the query pattern *EF(AG(...))* instead of *AG(EF(AG(...)))*.

2.5.5 CTL counterexamples

If a CTL formula is false then *NuSMV* can return a finite path that starts with an initial state that does not satisfy the formula.

Note: There is a fundamental difference between LTL and CTL counterexamples. LTL counterexamples prove that the formula is false in the sense that any transition system that contains that path will not satisfy the formula. CTL counterexamples, on the other hand, can not be used as general proofs. They merely contain an initial state that does not satisfy the formula *in the given transition system*.

Suppose we want to find out whether each initial states defined by *Proliferation* has a successor state that also satisfies *Proliferation*. To define this property we use the CTL operator *EX*:

```
>>> init "INIT Proliferation"
>>> spec "CTLSPEC EX(Proliferation)"
>>> answer = MC.check_primes(primes, update, init, spec)
False
>>> answer, counterex = MC.check_primes_with_counterexample(primes, update, init, spec)
>>> counterex
({'DNAdamage': 1, 'Proliferation': 1, 'GrowthFactor': 0},)
>>> STGs.state2str(counterex[0])
101
```

The function *check_primes_with_counterexample* returns a counterexample, an initial state, namely 101, that does not satisfy the given CTL spec, i.e., *EX(Proliferation)*. The correctness of this answer can be confirmed by enumerating all successors of 101 (in this case a single successor) by using *STGs.successors_asynchronous*:

```
>>> for x in STGs.successors_asynchronous(primes, "101"):
...     print x
{'DNAdamage': 1, 'Proliferation': 0, 'GrowthFactor': 0}
```

and checking that *Proliferation=0* for all of them.

Note: CTL counterexamples are in general also paths, for an explanation see e.g. *Baier2008*, but the length of the path and which sub-formula is not satisfied by the state it leads to depend on the given formula. It is often easier to just return the initial state that does not satisfy the whole formula, using:

```
>>> answer, counterex = MC.check_primes_with_counterexample(primes, update, init, spec)
>>> state = counterex[0]
```

2.5.6 existential queries

By definition, a LTL query is true iff *all paths* that are rooted in an initial state satisfy the LTL formula. Likewise, a CTL query is true iff *all initial states* satisfy the CTL formula. Without modifying the standard algorithms it is also possible to answer existential queries of the form: "Is there a path rooted in some initial state that satisfies a given LTL formula?" and "Is there an initial state that satisfies a given CTL formula?". The idea is to apply the following logical equivalences:

> There is an initial state that satisfies a given CTL formula iff it is *false* that every initial state satisfies the *negation* of the CTL formula.

and

> There is a path rooted in some initial state that satisfies a given LTL formula iff it is *false* that all paths satisfy the *negation* of the LTL formula.

As an example consider the following network:

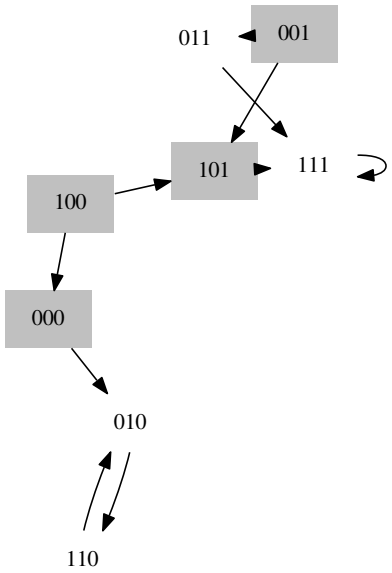
```
>>> bnet = ["x0,   !x0&x1 | x2",
...         "x1,   !x0 | x1 | x2",
...         "x2,   x0&!x1 | x2"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
```

and the queries "Every state that satisfies *x1=0* can reach an attractor in which *x0* is steady" (Q1) and "There is a state that satisfies *x1=0* that can reach an attractor in which *x0* is steady" (Q2). Note that the equivalence from above states that "Q2 is true iff not Q1 is false".

Let us first answer these queries without model checking, that is, by inspecting the state transition graph. As before, we highlight the initial states:

```
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> for x in stg.nodes():
...     if x[1]=="0":
...         stg.node[x]["style"] = "filled"
...         stg.node[x]["fillcolor"] = "gray"
>>> stg.graph["label"] = "Example 22: Existential queries"
>>> STGs.stg2image("example22_stg.pdf")
```

The result is shown in *the figure below*. It is easy to see that the network has two attractors, the steady state 111 (in which *x0* is steady) and a cyclic attractor which consists of the states 010 and 110, in which *x0* is not steady. It is also not hard to confirm that Q1 does not hold, because the initial state 000 can only reach the cyclic attractor, and that Q2 does hold, because 100 is an initial state that can reach the steady state 111.



Example 22: Existential queries

Figure 2.21: The state transition graph *"example22_stg.pdf"* with initial states highlighted by gray rectangles. The attractors are the steady state 111 and the cyclic attractor that consists of the states 010 and 110.

To decide the queries with CTL model checking we use the following encoding:

```
>>> init = "INIT !x1"
>>> specQ1 = "CTLSPEC  EF(AG(x0_STEADY))"
>>> specQ2 = "CTLSPEC !EF(AG(x0_STEADY))"

>>> update = "asynchronous"
>>> Q1 = MC.check_primes(primes, update, init, specQ1)
>>> Q1
False
>>> Q2 = not MC.check_primes(primes, update, init, specQ2)
>>> Q2
True
```

Note that *specQ2* is exactly the negation of *specQ1* and the result of checking *specQ2* has to be negated to obtain the answer to Q2.

Note: The queries *specQ1* and *specQ2* are both false although one is exactly the negation of the other. In LTL and CTL model checking, a formula as well as its negation may be false *simultaneously*. For CTL, this is the case when some initial state satisfy the formula and some other initial state does not. For LTL, this is the case when some admissible path satisfies the formula and some other path does not.

Note also that since *specQ2* is false we can ask *NuSMV* to generate a counterexample, i.e., an initial state that does not satisfy *specQ2*, i.e., a state that satisfies Q2. Counterexamples of existential queries are therefore often also called *witnesses*.

```
>>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

2.5.7 accepting states of CTL queries

Since Version 2.0 PyBoolnet supports model checking with so-called accepting states. That is, PyBoolNet uses NuSMV-a to return a Boolean expression that represents all states that satisfy the CTL spec and another Boolean expression that represents all initial states that satisfy the CTL spec. The functionality of returning accepting states was implemented in NuSMV-a, an extension of NuSMV. To return the accepting states use the function *check_primes_with_acceptingstates* or *check_smv_with_acceptingstates*. It returns a tuple consisting of the answer and a dictionary with the following keys and values:

- `"INIT"`: *str*, a factored formula representing the initial states
- `"INIT_SIZE"`: *int*, the cardinality of the initial states
- `"ACCEPTING"`: *str*, a factored formula representing the accepting states
- `"ACCEPTING_SIZE"`: *int*, the cardinality of the accepting states
- `"INITACCEPTING"`: *str*, a factored formula representing the initial and accepting states
- `"INITACCEPTING_SIZE"`: *int*, the cardinality of the initial and accepting states

Consider the previous network as an example:

```
>>> bnet = ["x0,   !x0&x1 | x2",
...         "x1,   !x0 | x1 | x2",
...         "x2,   x0&!x1 | x2"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
```

We already know that the query with initial states `!x1` and the CTL spec `EF(AG(x0_STEADY))` is false. Using the function *check_primes_with_counterexample* we found an initial state that does not satisfy the specification, i.e., 000. The function *check_primes_with_counterexample* can be used to get a complete picture of the intial states that satisfy the spec:

```
>>> update = "asynchronous"
>>> init = "INIT !x1"
>>> spec = "CTLSPEC EF(AG(x0_STEADY))"
>>> answer, accepting = MC.check_primes_with_acceptingstates(primes, update, init, spec)
>>> accepting["INITACCEPTING"]
'!(x0 & (x1) | !x0 & (x1 | !(x2)))'
```

The result is a *factored formula* that represents the exact set of states that satisfy the spec in NuSMV syntax so that it can be re-used for subsequent queries. The number of initial and accepting states can be obtained by:

```
>>> accepting["INITACCEPTING_SIZE"]
3
```

which explains why the query is false, since there are four initial states, i.e., one that does not satisfy the spec:

```
>>> accepting["INIT_SIZE"]
4
```

It is also possible to obtain the complete set of states that satisfy the spec, i.e., including states that are not initial:

```
>>> accepting["ACCEPTING"]
'x0 & ((x2) | !x1) | !x0 & (x2)'
```

The size of this set tells us that there are two states outside of the initial one that also satisfy the spec:

```
>>> accepting["ACCEPTING_SIZE"]
5
```

Note that *PyBoolNet 2.1* does not currently support the manipulation of Boolean expression. They may however be used in subsequent queries. For example, we may query whether all initial states that satisfy the original spec also satisfy the property `EF(STEADYSTATE)`:

```
>>> prop = accepting["INITACCEPTING"]
>>> init = "INIT %s"%prop
>>> spec = "CTLSPEC EF(STEADYSTATE)"
>>> MC.check_primes(primes, update, init, spec)
True
```

You can use the function *list_states_referenced_by_proposition* to enumerate all states that are referenced by a propositional formula:

```
>>> for x in STGs.list_states_referenced_by_proposition(primes, prop): print x
001
101
100
```

2.6 Computing Trap Spaces

Maximal, Minimal and All Trap Spaces The term *trap space* merges the notions “subspace” and “trap set”. Hence, once a trajectory enters a trap space it can not escape. Trap spaces have a number of interesting properties: they are independent of the update strategy, i.e., they are identical for all state transition graphs, they satisfy a partial order defined by set inclusion of the respective states contained in them and they can be computed efficiently for networks with hundreds of components. Intuitively, trap spaces can be seen as generalizations of steady states (note that steady

states have the same three properties). For a formal introduction, an algorithm for computing trap spaces and a benchmark see *Klarner2015(a)*.

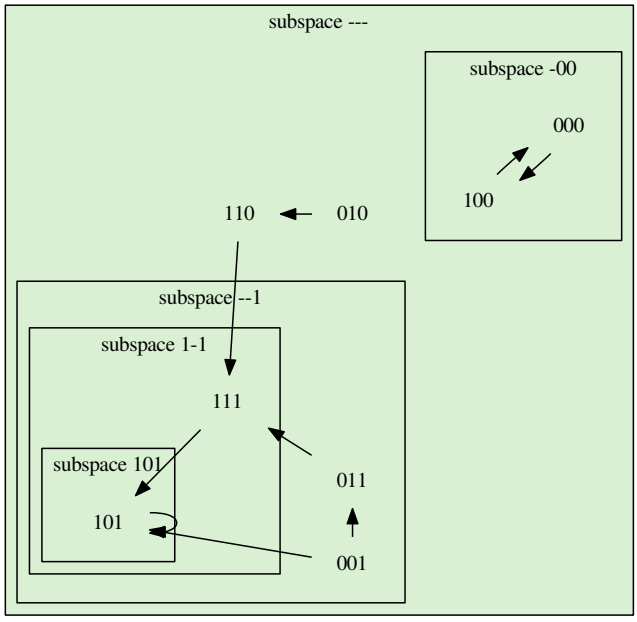
PyBoolNet 2.1 uses the module *TrapSpaces* and the function *trap_spaces* to compute trap spaces. As an example, consider the following network which has five trap spaces:

```
>>> from PyBoolNet import TrapSpaces as TS
>>> bnet = ["x, !x | y | z",
...         "y, !x&z | y&!z",
...         "z, x&y | z"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> tspaces = TS.trap_spaces(primes, "all")
>>> print ", ".join(STGs.subspace2str(primes, x) for x in tspaces)
---, --1, 1-1, -00, 101
```

The trap space `---`, i.e., the full state space, is also called the trivial trap space. `101` is a steady state and there are three more trap spaces, `--1`, `1-1` and `-00`. Note that some trap spaces are comparable using subset inclusion, i.e., `1-1` < `--1` because the two states contained in `1-1` are also contained in `--1`, while others are not comparable, for example `--1` and `-00`.

The trap spaces are illustrated in *the figure below* using the subspaces style:

```
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> STGs.add_style_subspaces(primes, stg, tspaces)
>>> stg.graph["label"] = "Example 23: All trap spaces"
>>> STGs.stg2image(stg, "example23_stg.pdf")
```



Example 23: All trap spaces

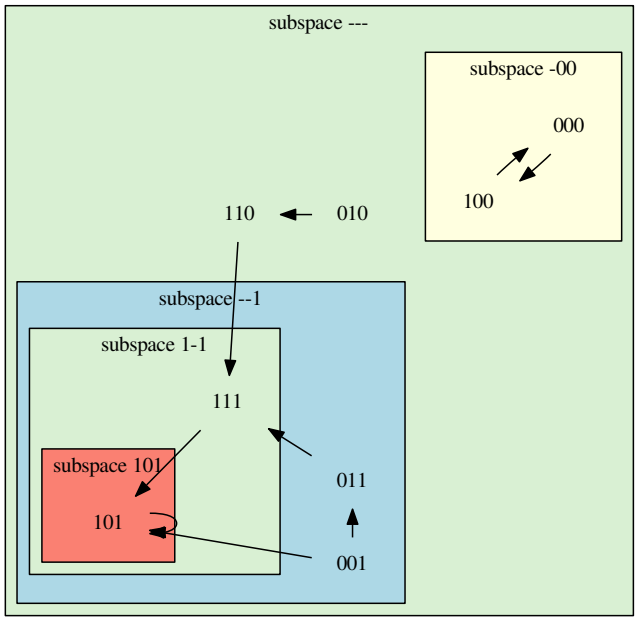
Figure 2.22: The state transition graph *"example23_stg.pdf"* with every trap space highlighted by its own subgraph.

The number of all trap spaces of a network can be very large and one is often only interested in the subset of minimal or maximal trap spaces. These can also be computed using *trap_spaces* by passing *"min"* or *"max"* instead of the

previously used value *"all"* for the second parameter:

```
>>> mintspaces = TS.trap_spaces(primes, "min")
>>> for x in mintspaces:
...     sub = (x,{"fillcolor":"salmon"})
...     STGs.add_style_subspaces(primes, stg, [sub])
>>> maxtspaces = TS.trap_spaces(primes, "max")
>>> for x in maxtspaces:
...     if x in mintspaces:
...         sub = (x,{"fillcolor":"lightyellow"})
...         STGs.add_style_subspaces(primes, stg, [sub])
...     else:
...         sub = (x,{"fillcolor":"lightblue"})
...         STGs.add_style_subspaces(primes, stg, [sub])
>>> stg.graph["label"] = "Example 24: Minimal and maximal trap spaces"
>>> STGs.stg2image(stg, "example24_stg.pdf")
```

The result is shown in *the figure below* in which `-00` is minimal and maximal (yellow), `--1` is maximal (blue), `1-1` is neither maximal nor minimal (green), and `101` is minimal (red).



Example 24: Minimal and maximal trap spaces

Figure 2.23: The state transition graph *"example24_stg.pdf"* with minimal trap spaces in red, maximal trap spaces in blue, trap spaces that are minimal and maximal at the same time in yellow and the remaining trap spaces in green.

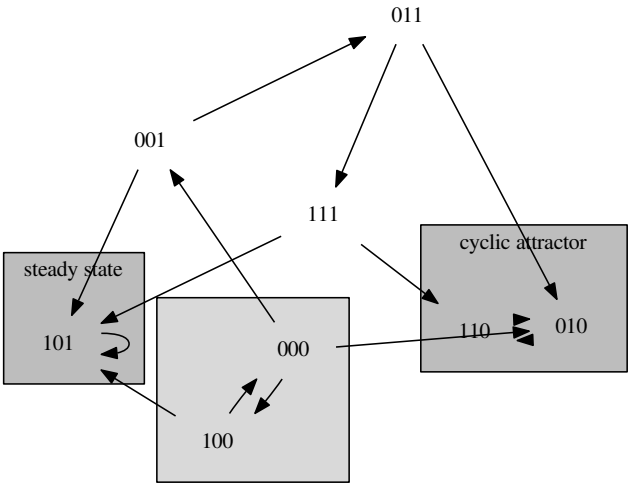
Note: It is possible that two non-minimal trap spaces intersect in which case the intersection is again a trap space. Since *Graphviz* can not draw intersecting subgraphs it is therefore not always possible to draw all trap spaces. Minimal trap spaces on the other hand, can not intersect and can always be drawn in the same STG.

2.7 Attractors

2.7.1 attractor detection

Attractors capture the long-term activities of the components of Boolean networks. Two different types of attractors are possible: either all activities stabilize at some values and the network enters a steady state or at least one component shows sustained oscillations and the network enters a cyclic attractor. Formally, attractors are defined as the inclusion-wise minimal trap sets of a given STG which is equivalent to the so-called terminal SCCs of the state transition graph. One approach to computing the attractors is to use Tarjan's algorithm for computing the SCCs of a directed graph, see *Tarjan1972* and then to select those SCCs that are terminal, i.e., those for which there is no path to another SCC. This approach is implemented in the function *compute_attractors_tarjan*. As an example for computing attractors with this algorithm consider the following network and its asynchronous STG which is given in *the figure below*:

```
>>> import AttractorDetection as AD
>>> bnet = ["v1, !v1 | v3",
...         "v2, !v1 | v2&!v3",
...         "v3, !v2"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> STGs.add_style_sccs(stg)
>>> stg.graph["label"] = "Example 25: A network with a cyclic attractor and a steady state."
>>> STGs.stg2image(stg, "example25_stg.pdf")
>>> attractors = AD.compute_attractors_tarjan(stg)
>>> len(attractors)
2
>>> for A in attractors:
...     print [STGs.state2str(x) for x in A]
['101']
['010', '110']
```



Example 25: A network with a cyclic attractor and a steady state.

Figure 2.24: The asynchronous STG *"example25_stg.pdf"* of a network with a steady state and a cyclic attractor.

Due to the state space explosion problem, the approach of computing the terminal SCCs by explicitly constructing the underlying STG as a digraphis limited to networks with less than 15~20 components.

There are algorithms for larger networks, but the "best" algorithm for solving the detection problem will depend on the chosen update strategy. For synchronous STGs we suggest to use an approach that was suggested by *Dubrova2011* and involves a SAT solver and bounded LTL model checking. It has been implemented as a tool called *bns* which is available at https://people.kth.se/~dubrova/bns.html.

Note: Boolean networks can be converted into *bns* file format with *primes2bns*.

Note: Whereas the steady states of the synchronous and asynchronous STGs are identical, the number and composition of cyclic attractors depends, in general, on the chosen update strategy.

A fairly efficient approach to detecting at least some attractors or larger networks is mentioned in *Klarner2015(a)* and based on the idea of generating a random walk in the STG, starting from some initial state, and then testing with CTL model checking whether the final state is indeed part of an attractor. This approach is based on the observation that, in practice, a random walk will quickly reach states that belong to an attractor. It is implemented in the function *find_attractor_state_by_randomwalk_and_ctl*:

```
>>> state = AD.find_attractor_state_by_randomwalk_and_ctl(primes, "asynchronous")
>>> STGs.state2str(state)
110
```

The function takes three optional parameters: *InitialState* which allows to specify a subspace from which to sample the initial state, *Length* which is an integer that specifies the number of transitions for the generation of the random walk, and *Attempts* which is the maximal number of random walks that are generated if each time the last state does not belong to an attractor. Though unlikely, it is possible that the function will not find an attractor in which case it will raise an exception. Hence, *find_attractor_state_by_randomwalk_and_ctl* should always be encapsulated in a *Try-Except* block:

```
>>> try:
...     state = AD.find_attractor_state_by_randomwalk_and_ctl(primes, "asynchronous")
...     print STGs.state2str(state)
... except:
...     print "did not find an attractor. try increasing the length or attempts parameters"
```

2.7.2 attractor basins

The module *AttractorBasins* contains functions for constructing diagrams that illustrate the basins of attraction of a given STG. In non-deterministic STGs there are usually states from which more than one attractor is reachable. But, not every combination of attractors has states that can reach exactly that subset of attractors. The function *basins_diagram* checks for each possible combination of attractors whether the set of corresponding basin states is empty or not. If there are states a basin node is created. An edge between basin nodes indicates the existence of a transition between two states of the respective sets of states. The nodes of a basin diagram have the following attributes:

- `"formula"` (str), the factored formula representing the states in that basin
- `"size"` (int), the number of states in that basin
- `"attractors"` (list), the list of attractors that define that basin (represented by individual states or subspaces)

The edges of a basin diagram have the following attributes:

- `"finally_formula"` (str), the factored formula that represents the states that can reach a state that can make the transition

- `"finally_size"` (int), the number of such states

Basin diagrams can be visualized with the function *diagram2image*. The function takes the primes, diagram and file name of the image as parameters. In addition you may specify *FnameATTRACTORS* if you want a separate file for the specification of the attractors. Otherwise they are included in the diagram image. Two parameters for styling the diagram are provided. *StyleInputs* highlight the basin nodes that belong to the same input combination and *StyleAdvanced* modifies the node and edge styles to highlight nodes and transition that are *homogeneous*. For details see the upcoming publication *Klarner2016*.

Consider the following example:

```
>>> primes = REPO.get_primes("arellano_rootstem")
>>> diagram = AB.basins_diagram(primes, "asynchronous")
>>> AB.diagram2image(primes, diagram, "diagram.pdf")
```

The output is given in *the figure below*. It uses the following convention: basin nodes that belong to the same input combination are grouped as light green subgraphs. The fillcolor of a basin node reflects the proportion of states that belong to it: the darker the more states. Nodes are labeled by the attractors they can reach which are enumerated by A0, A1, etc. Cyclic attractros are represented by minimal trap spaces.

Note that the function *basins_diagram* either requires a list of states representing attractors (given via the parameter *Attractors*), or it will compute the minimal trap spaces and *assume* that they are complete and univocal. You should make sure that the minimal trap spaces are indeed complete and univocal using the functions *completeness* and *univocal*.

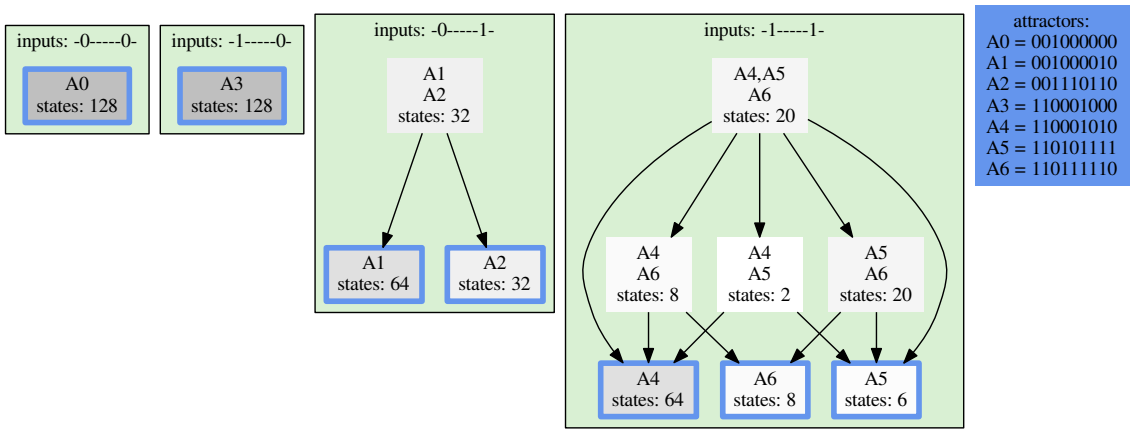


Figure 2.25: The basin diagram of the network *arellano_rootstem* from the repository.

If you want to create a separate image for the basin and the states representing the attractors use the parameter *FnameATTRACTORS* of the function *diagram2image*:

```
>>> AB.diagram2image(primes, diagram, "diagram.pdf", "attractors.pdf")
```

Finally, in particular for diagrams with many nodes you may want to generate the so-called aggregate diagram in which nodes with the same number of reachable attractors are combined. That is, all nodes that can reach, for example, exactly two attractors (irrespective of which two attractors) are represented by a single node. An example is given in *the figure below*:

```
>>> AB.diagram2aggregate_image(diagram, "aggregate.pdf")
```

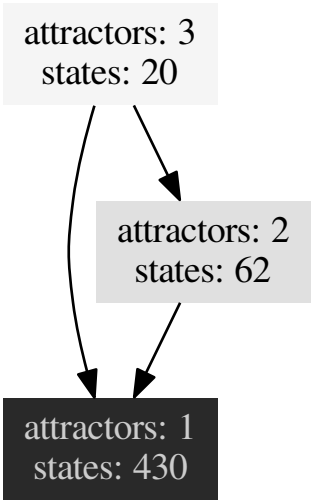


Figure 2.26: The aggregated basin diagram of the network *arellano_rootstem* from the repository.

2.7.3 attractor approximations

Minimal trap spaces approximate attractors because every trap space contains an attractor. But, there can be attractors outside of minimal trap spaces. And there may be several attractors inside a single minmal trap space. And an attractor inside a minimal trap space may be much smaller than the minimal trap space that contains it. Hence there are three criteria for the quality of minimal trap spaces as approximations for attractors:

1. **completeness**: whether every attractor is contained in one of the network's minimal trap spaces
2. **univocality**: whether each minimal trap spaces contains exactly one attractor
3. **faithfulness**: whether the number of free variables of the minimal trap space is equal to the number of oscillating variables of the attractors contained in it

If the minimal trap spaces of a network are complete, univocal and faithful then we say that the approximation is perfect. So far, the minimal trap spaces of every published network we are aware of are a perfect approximation of its attractors. Hence, although computing the attractors of asynchronous STGs is in general a hard problem, in practice we may get away with computing their minimal trap spaces which can efficiently be done for networks with hundreds of components. Note that for limit cycles of synchronous STGs and steady states other algorithms, e.g. *Dubrova2011* and *Naldi2007* are more suitable.

In *Klarner2015(a)* we demonstrate that completeness, univocality and faithfulness can all be decided using CTL model checking. The functions *completeness*, *univocal* and *faithful* automatically generate and test the respective queries, which are defined in Sections 4.1, 4.2 and 4.3 of *Klarner2015(a)*.

As an example of a network whose minimal trap spaces are complete, univocal and faithful consider again the network defined in *the figure above*. The functions *univocal* and *faithful* each require the primes, update strategy and a trap space:

```
>>> update = "asynchronous"
>>> mintspaces = TS.trap_spaces(primes, "min")
```

```
>>> for x in mintspaces:
...     answer_univocal = AD.univocal(primes, update, x)
...     answer_faithful = AD.faithful(primes, update, x)
...     print "min trap space:", STGs.subspace2str(primes, x)
...     print "  is univocal:", answer_univocal
...     print "  is faithful:", answer_faithful
min trap space: -10
  is univocal: True
  is faithful: True
min trap space: 101
  is univocal: True
  is faithful: True
```

The function for deciding whether the minimal trap spaces are complete requires only two arguments, the primes and the update strategy, because it is implied that the trap spaces must be all minimal ones. See *completeness* for details.

```
>>> AD.completeness(primes, update)
True
```

Since *univocality* is based on detecting at least one attractor, via the random walk algorithm explained above, and since a counterexample to the univocality query contains information about additional attractors, there is a second function, called *univocality_with_counterexample'returns a triplet consisting of the answer, an attractor state and a counterexample (if the trap space is not univocal), see :ref:'univocality* for details. The function *faithfulness_with_counterexample* returns a tuple that consists of the answer and a counterexample if it exists.

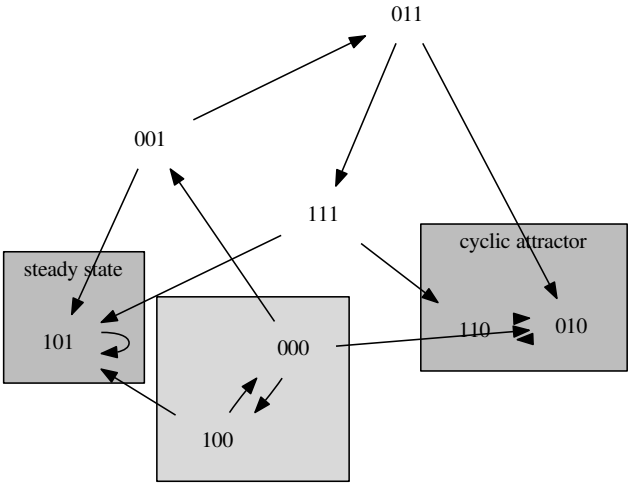
As an illustration, consider network (A) given in Figure 1 of *Klarner2015(a)*. It is defined by the following functions:

The resulting STG is shown in *the figure below*.

Its STG contains two cyclic attractors and its minimal trap space −−− contains two cyclic attractors and it therefore not univocal.

```
>>> bnet = ["v1, !v1&!v2&v3 | !v1&v2&!v3 | v1&!v2&!v3 | v1&v2&v3",
...         "v2, !v1&!v2&!v3 | !v1&v2&v3 | v1&!v2&v3 | v1&v2&!v3",
...         "v3, !v1&!v2&v3 | !v1&v2&!v3 | v1&!v2&!v3 | v1&v2&v3"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> mintspaces = TS.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = TS.trap_spaces(primes, "min")
>>> print [STGs.subspace2str(primes, x) for x in mintspaces]

>>> STGs.add_style_sccs(stg)
>>> STGs.add_style_subspaces(primes, stg, mintspaces)
```



Example 25: A network with a cyclic attractor and a steady state.

Figure 2.27: The state transition graph *"example25_stg.pdf"* in which the minimal trap space "—" is not univocal.

```
>>> mintspaces = TS.trap_spaces(primes, "min")
>>> print [STGs.subspace2str(primes, x) for x in mintspaces]
['---']
>>> STGs.add_style_subspaces(stg, mintspaces)
>>> stg.graph["label"] = "Example 26: An STG whose minimal trap space '---' is not univocal"
```

CHAPTER

THREE

REFERENCE

3.1 FileExchange

3.1.1 bnet2primes

3.1.2 primes2bnet

3.1.3 write_primes

3.1.4 read_primes

3.1.5 primes2genysis

3.1.6 primes2bns

3.1.7 primes2eqn

3.2 PrimeImplicants

3.2.1 copy

3.2.2 are_equal

3.2.3 find_inputs

3.2.4 find_outputs

3.2.5 find_constants

3.2.6 create_constants

3.2.7 create_inputs

3.2.8 create_blinkers

3.2.9 create_variables

3.2.10 create_disjoint_union

3.2.11 remove_variables

3.2.12 remove_all_variables_except

CHAPTER

FOUR

BIBLIOGRAPHY

Arellano2011: Antelope: a hybrid-logic model checker for branching-time Boolean GRN analysis. G. Arellano, et al. BMC bioinformatics, 2011.

Baier2008: Principles of Model Checking. C. Baier and J.-P. Katoen. The MIT Press, 2008.

Chaouiya2012: Logical modelling of gene regulatory networks with GINsim. C. Chaouiya, A. Naldi, D. Thieffry. Bacterial Molecular Networks, p.463-479, Springer, 2012.

Clarke2002: Tree-like counterexamples in model checking. E. Clarke, S. Jha, Y. Lu, and H. Veith. 17th annual IEEE symposium on logic in computer science, 2002.

Dubrova2011: A SAT-based algorithm for finding attractors in synchronous Boolean networks. E. Dubrova and M. Teslenko. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2011, volume 8, number 5, pages 1393-1399.

Garg2008: Synchronous versus asynchronous modeling of gene regulatory networks. A. Garg, A. Di Cara, I. Xenarios, L. Mendoza and G. De Michli. Bioinformatics, 2008, volume 24, number 17, pages 1917-1925.

Grieco2013: Integrative modelling of the influence of MAPK network on cancer cell fate decision. Grieco L., Calzone L., Bernard-Pierrot I., Radvanyi F., Kahn-Perlès B. and Thieffry D. PLoS computational biology, 2013, volume 9, issue 10.

Gebser2011: Potassco: The Potsdam Answer Set Solving Collection. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub and M. Schneider. AI Communications, 2011, volume 24, number 2, pages 107-124.

Klarner2014: Computing symbolic steady states of Boolean networks. H. Klarner, H. Siebert and A. Bockmayr. Lecture Notes in Computer Science, 2014, volume 8751, pages 561-570.

Klarner2015(a): Computing maximal and minimal trap spaces of Boolean networks. H. Klarner, A. Bockmayr and H. Siebert. Natural computing, 2015, volume 14, issue 4, pages 535-544.

Klarner2015(b): Approximating attractors of Boolean networks by iterative CTL model checking. H. Klarner and H. Siebert. Frontiers in Bioengineering and Biotechnology, 2015, volume 3, number 130.

Klarner2016: in preparation.

Müssel2010: BoolNet: An R package for generation, reconstruction and analysis of Boolean networks. C. Müssel, M. Hopfensitz and H. Kestler. Bioinformatics, 2010, volume 26, number 10, pages 1378-1380.

– _Naldi2007:

Naldi2007: Decision diagrams for the representation and analysis of logical models of genetic networks. A. Naldi, D. Thieffry and C. Chaouiya. Computational Methods in Systems Biology, 2007, Springer, pages 233-247.

Prekas2012: Quine-McCluskey algorithm. G. Prekas. https://github.com/prekageo/optistate/blob/master/qm.py

Tarjan1972: Depth-first search and linear graph algorithms R. Tarjan. SIAM Journal of Computing, 1972, 1(2):146-160.

Berenguier2013: Dynamical modeling and analysis of large cellular regulatory networks D. Berenguier, C. Chaouiya, P. Monteiro, A. Naldi, E. Remy, D. Thieffry and L. Tichit Chaos: An Interdisciplinary Journal of Nonlinear Science, 2013.

Tournier2009: Uncovering operational interactions in genetic networks using asynchronous Boolean dynamics L. Tournier and M. Chaves. Theoretical Biology, 2009