pyboolnet documentation

Release 3.0.10

Hannes Klarner

Sep 11, 2022

CONTENTS

pyboolnet 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* 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 is now being developed in Python 3.8 and is not compatible with Python2. If you experience problems with your version of Python and *pyboolnet* 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

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

```
$ apt install python-pip
```

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

```
$ pip3 install networkx
$ apt install graphviz
$ apt install imagemagick
```

Install *pyboolnet* with *pip*:

```
$ pip3 install git+https://github.com/hklarner/pyboolnet
```

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:

```
$ pip3 install git+https://github.com/hklarner/pyboolnet --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*:

```
$ python
>>> import pyboolnet
```

To remove *pyboolnet* using *pip* run:

```
$ pip3 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:

```
$ easy_install pip
```

or if you do not have super user rights:

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

Download and install Graphviz and ImageMagick from

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

Install *pyboolnet* with *pip*:

```
$ pip3 install git+https://github.com/hklarner/pyboolnet
```

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:

```
$ pip3 install git+https://github.com/hklarner/pyboolnet --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*:

```
$ python
>>> import pyboolnet
```

To remove *pyboolnet* using *pip* run:

```
$ pip3 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* with *pip*:

```
C:\> pip.exe install git+https://github.com/hklarner/pyboolnet
```

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 networkx
```

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*:

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

To remove *pyboolnet* 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* 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 `binaries` of *pyboolnet*. The default location is:

```
/usr/local/lib/python<version>/dist-packages/pyboolnet/binaries/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_linux64
bnet2prime      = ./BNetToPrime/BNetToPrime_linux64
clasp           = ./clasp-3.1.1/clasp-3.1.1-x86_64-linux
espresso        = ./espresso/espresso_linux64
eqntott         = ./eqntott/eqntott_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:

```
$ pip3 install networkx
```

or:

```
$ pip3 install networkx --user
```

if you do not have super user rights.

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* 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* 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* 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:

```
$ apt 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:

```
$ apt install ImageMagick
```

ImageMagick is available at

- http://www.imagemagick.org

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

1.5.7 Matplotlib

The package matplotlib is used to plot pie charts and bar plots and similiar graphs, for example in the function create_barplot and basins_create_piechart of the Basins module. To install it on linux run:

```
$ pip3 install matplotlib
```

matplotlib is available at

- http://matplotlib.org

1.5.8 Espresso

Espresso is required for heuristic minimization Boolean expressions. For more information see

- http://chmod755.tumblr.com/post/31417234230/espresso-heuristic-logic-minimizer

1.5.9 Eqntott

Eqntott is required to converting Boolean expressions into the truth table format required by Espresso. For more information see

- https://code.google.com/archive/p/eqntott

1.6 Related 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* but you need it if you want to convert *SBML-qual* files into *bnet* files. To install it run:

```
$ 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* 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:

```
$ apt 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* (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

QUICK MANUAL

2.1 Boolean Networks

Create a Boolean network from a text based definition file in *boolnet* format. Use &, | and ! for conjunction, disjunction and negation and separate the variable name from its activation expression by a comma. Values for constant activation are 0 and 1. Example of BNET file:

```
 v1,    !v1
v2,    1
v3,    v2 & (!v1 | v3)
```

Read a BNET file or its string contents with the function bnet2primes of the module FileExchange. It returns the prime implicant representation of a network:

```
>>> primes = PyBoolNet.FileExchange.bnet2primes(bnet)
```

Use the module PrimeImplicants to inspect and modify a network. For example, find all constant variables using find_constants:

```
>>> const = PrimeImplicants.find_constants(primes)
>>> print(const)
{'v2': 1}
```

Or create new variables using create_variables:

```
>>> PyBoolNet.PrimeImplicants.create_variables(primes, {"v4": "v4 | v2"})
```

You may also use python functions to define the update of a variable:

```
>>> PyBoolNet.PrimeImplicants.create_variables(primes, {"v5": lambda v1,v2,v3: sum(v1,v2,
↪v3)==1})
```

Convert to primes back to the BNET format using primes2bnet:

```
>>> print(PyBoolNet.FileExchange.primes2bnet(primes))
v2,  1

v1,  !v1
v3,  v2&v3 | v2&!v1
v4,  v4 | v2

v5,  !v2&v3&!v1 | v2&!v3&!v1 | !v2&!v3&v1
```

PyBoolNet has its own network repository. To browse it online, visit

- github.com/hklarner/pyboolnet/tree/master/pyboolnet/Repository

To access a network from the repository use the function get_primes of the module *Repository*:

```
>>> primes = get_primes("remy_tumorigenesis")
>>> print(PyBoolNet.FileExchange.primes2bnet(primes))
DNA_damage,          DNA_damage
EGFR_stimulus,       EGFR_stimulus
FGFR3_stimulus,      FGFR3_stimulus
...
Growth_arrest,       p21CIP | RBL2 | RB1
Proliferation,       CyclinE1 | CyclinA
```

Read the references for FileExchange and PrimeImplicants for more on this topic. For a tutorial script on this topic see *01_boolean_networks.py* in the tutorials folder at

- github.com/hklarner/pyboolnet/tree/master/Tutorials

2.2 Interaction Graphs

The basic function for drawing interaction graphs is igs_create_image of the module InteractionGraphs:

```
>>> primes = get_primes("remy_tumorigenesis")
>>> PyBoolNet.InteractionGraphs.create_image(primes, "igraph.pdf")
created igraph.pdf
```

The look of the interaction graph may be modified by one of the predefined styles:

```
>>> PyBoolNet.InteractionGraphs.create_image(primes, "igraph2.pdf", Styles=["sccs",
↪"anonymous"])
created igraph2.pdf
```

The next level of customizing the look of an interaction graph is to style the interaction graph with graphviz properties. Examples of node properties are *shape*, *label*, *style*, *width*, *color* and *fillcolor*. Examples of edge properties are *arrowhead*, *label* and *color*. To obtain the interaction graph use primes2igraph of the module InteractionGraphs. To draw a styled igraph use igraph2image:

```
>>> for x in igraph.nodes():
...         if "GF" in x:
...          igraph.node[x]["shape"] = "square"
...             igraph.node[x]["fillcolor"] = "lightblue"

>>> PyBoolNet.InteractionGraphs.igraph2image(igraph, "igraph3.pdf")
created igraph3.pdf
```

You may also compute the local interaction graph of a given state. To generate a random state, use random_state of the module StateTransitionGraphs. To compute the local interaction graph use local_igraph_of_state of InteractionGraphs:

```
>>> state = PyBoolNet.StateTransitionGraphs.random_state(primes)
>>> local_igraph = PyBoolNet.InteractionGraphs.local_igraph_of_state(state)
>>> PyBoolNet.InteractionGraphs.add_style_interactionsigns(local_igraph)
```

(continues on next page)

(continued from previous page)

```
>>> PyBoolNet.InteractionGraphs.igraph2image(local_igraph, "local_igraph.pdf")
created local_igraph.pdf
```

Read the references for InteractionGraphs for more on this topic. For a tutorial script on this topic see *02_interaction_graphs.py* in the tutorials folder at

- github.com/hklarner/pyboolnet/tree/master/Tutorials

2.3 State Transition Graphs

Drawing and styling state transition graphs is analogous to interaction graphs:

```
>>> primes = get_primes("xiao_wnt5a")
>>> PyBoolNet.StateTransitionGraphs.create_image(primes, "asynchronous", "stg.pdf")
created stg.pdf
```

The update may be either *asynchronous* or *synchronous*. You may specify initial states as a single state or several states, a subspace or a python indicator function:

```
>>> PyBoolNet.StateTransitionGraphs.create_image(primes, "asynchronous", "stg2.pdf",␣
↪InitialStates={"x6":0, "x7":0}, Styles=["anonymous", "tendencies", "mintrapspaces"])
created stg2.pdf
```

To draw paths use add_style_path. A random walk ma be computed using random_walk:

```
>>> path = PyBoolNet.StateTransitionGraphs.random_walk(primes, "asynchronous",␣
↪InitialState="11---00", Length=4)
>>> stg = PyBoolNet.StateTransitionGraphs.primes2stg(primes, "asynchronous")
>>> PyBoolNet.StateTransitionGraphs.add_style_path(stg, path, Color="red")
>>> PyBoolNet.StateTransitionGraphs.stg2image(stg, "stg3.pdf")
created stg3.pdf
```

The module StateTransitionGraphs contains also functions to compute factor graphs in which the state space is partitioned into classes. The classical example is the SCC graph:

```
>>> primes = get_primes("randomnet_n7k3")
>>> stg = PyBoolNet.StateTransitionGraphs.primes2stg(primes, "asynchronous")
>>> scc_graph = PyBoolNet.StateTransitionGraphs.stg2sccgraph(stg)
>>> PyBoolNet.StateTransitionGraphs.sccgraph2image(scc_graph, "scc_graph.pdf")
```

The condensation graph is like the scc graph but transient states are removed:

```
>>> cograph = PyBoolNet.StateTransitionGraphs.stg2condensationgraph(stg)
>>> PyBoolNet.StateTransitionGraphs.sccgraph2image(cograph, "cograph.pdf")
```

The hierarchical transition graph is even further compressed by considering the attractors of the stg:

```
>>> htg = PyBoolNet.StateTransitionGraphs.stg2htg(stg)
>>> PyBoolNet.StateTransitionGraphs.htg2image(htg, "htg.pdf")
```

Reachability questions of the form "Is there a path from an initial state X to a goal state Y?" may be answered by a best first search algorithm with the function best_first_reachability. The search is guided by reducing the hamming distance to the goal space:

```
>>> X = "0--1-1-"
>>> Y = "1--0---"
>>> path = PyBoolNet.StateTransitionGraphs.best_first_reachability(primes,␣
↪InitialSpace=X, GoalSpace=Y)
>>> if path:
...         for x in path:
...             print(x)
... else:
...         print("no path found")

0011011
...
1000001
```

Finally, you may compute the integer-valued energy of a state, based on "frozen variables" with the function energy:

```
>>> x = "0001011"
>>> e = PyBoolNet.StateTransitionGraphs.energy(primes, x)
>>> print(e)
```

Read the references for StateTransitionGraphs for more on this topic. For a tutorial script on this topic see *03_state_transition_graphs.py* in the tutorials folder at

- github.com/hklarner/pyboolnet/tree/master/Tutorials

2.4 Model Checking

The basic function for LTL and CTL model checking is check_primes of the module ModelChecking. The initial states and the specification must be given in NuSMV syntax. For LTL specs use the keyword *LTLSPEC*, for CTL specs use *CTLSPEC* and for initial states use *INIT*:

```
>>> primes = get_primes("remy_tumorigenesis")
>>> init = "INIT TRUE"
>>> spec = "CTLSPEC DNA_damage -> AG(EF(Apoptosis_medium))"
>>> answer = PyBoolNet.ModelChecking.check_primes(primes, "asynchronous", init, spec)
```

For model checking with accepting states use check_primes_with_acceptingstates. The function returns a dictionary of further information regarding the initial and accepting states:

```
>>> answer, accepting = PyBoolNet.ModelChecking.check_primes_with_acceptingstates(primes,
↪ "asynchronous", init, spec)
>>> for key, value in accepting.items():
...         print("{} = {}".format(key, value))

INIT_SIZE = 8153726976
INITACCEPTING_SIZE = 8153726976
INIT = E2F1_medium & (ATM_medium & (CHEK1_2_medium ... | !(Apoptosis_high)))))))))
ACCEPTING = TRUE
ACCEPTING_SIZE = 34359738368
INITACCEPTING = E2F1_medium & (ATM_medium & (CHEK1_2_medium & (E2F3_medium & (Apoptosis_
↪medium | ... !(Apoptosis_high)))))))))
```

Finally, the function check_primes_with_counterexample returns a CTL or LTL counter example, if the query is false:

```
>>> spec = "CTLSPEC DNA_damage -> AG(EF(Proliferation))"
>>> answer, counterex = PyBoolNet.ModelChecking.check_primes_with_counterexample(primes,
↪"asynchronous", init, spec)
>>> print(answer)
>>> if counterex:
...     for state in counterex:
...         print(state)
{'RB1': 0, 'GRB2': 0, 'RAS': 0, 'p16INK4a': 0, 'Proliferation': 0, ..., 'DNA_damage': 1,
↪'FGFR3': 0, 'Apoptosis_high': 0, 'CyclinD1': 0, 'p21CIP': 0}
```

Read the references for StateTransitionGraphs for more on this topic. For a tutorial script on this topic see *04_model_checking.py* in the tutorials folder at

- github.com/hklarner/pyboolnet/tree/master/Tutorials

2.5 Attractors

The two basic functions for finding attractors are Tarjan's algorithm and the random walk algorithm. Tarjan's algorithm is exact but requires the full STG as input and is therefore limited to smaller networks:

```
>>> primes = get_primes("tournier_apoptosis")
>>> stg = PyBoolNet.StateTransitionGraphs.primes2stg(primes, "asynchronous")
>>> steady, cyclic = PyBoolNet.Attractors.compute_attractors_tarjan(stg)
>>> steady
['000101010000', '011000010000']
>>> cyclic
[{'011000010011', '111010010111', ..., '111010001111', '011010000011', '011010001001'}]
```

The random walk algorithm works for larger networks but it finds only a single attractor:

```
>>> state = PyBoolNet.Attractors.find_attractor_state_by_randomwalk_and_ctl(primes,
↪"asynchronous")
>>> print(state)
```

To compute all attractors with an advanced model checking algorithm use the function attractors_compute_json:

```
>>> attrs = PyBoolNet.Attractors.compute_json(primes, "asynchronous", FnameJson="attrs.
↪json")
```

Inspect the json structure with e.g. http://jsonviewer.stack.hu/. Iterate of the attractors and print representative states:

```
>>> attrs["is_complete"]
yes
>>> for x in attrs["attractors"]:
...         print(x["is_steady"])
...         print(x["state"]["str"])
```

2.6 Basins

To compute the weak, strong and cycle-free basins of a subspace use the functions weak_basins, strong_basin and cyclefree_basin:

```
>>> primes = get_primes("tournier_apoptosis")
>>> attrs = PyBoolNet.Attractors.compute_json(primes, "asynchronous")
>>> state = attrs["attractors"][0]["state"]["str"]
>>> weak = PyBoolNet.Basins.weak_basin(primes, "asynchronous", state)
>>> for key, value in weak.items():
...         print("{} = {}".format(key, value))

formula = !(TNF | !(NFkB | (NFkBnuc | ((IKKa | ((CARP | (IAP | !(C8a))) | !C3a)) | !
↪IkB))))
size = 2040
perc = 49.8046875

>>> strong = PyBoolNet.Basins.strong_basin(primes, "asynchronous", state)
>>> for key, value in strong.items():
...         print("{} = {}".format(key, value))

size = 704
perc = 17.1875
formula = !(TNF | (IKKa | (C3a | !(T2 & (CARP & (IAP | !(C8a)) | !CARP & (IAP)) | !T2 &␣
↪(IAP | !(C8a))))))

>>> cycfree = PyBoolNet.Basins.cyclefree_basin(primes, "asynchronous", state)
>>> for key, value in cycfree.items():
...         print("{} = {}".format(key, value))

formula = !(TNF | (IKKa | (C3a | !(T2 & (CARP & (IAP | !(C8a)) | !CARP & (IAP)) | !T2 &␣
↪(IAP | !(C8a))))))
size = 352
perc = 8.59375
```

To plot the sizes of the basins as a bar plot or a pie chart use compute_basins. It extends the attrs data obtained from compute_json.

```
>>> PyBoolNet.Basins.compute_basins(attrs)
>>> PyBoolNet.Basins.create_basins_of_attraction_barplot(attrs, "basin_barplot.pdf")
>>> PyBoolNet.Basins.create_phenotypes_piechart(attrs, "basin_piechart.pdf")
```

2.7 Commitment

To compute the commitment diagram use the function create_diagram of the module Commitment. It requires the attractors data as input:

```
>>> primes = get_primes("tournier_apoptosis")
>>> attrs = PyBoolNet.Attractors.compute_json(primes, "asynchronous")
>>> diag = PyBoolNet.Commitment.compute_diagram(attrs)
```

Commitment diagrams may be visualized as graphs or piecharts:

```
>>> PyBoolNet.Commitment.diagram2image(diag, "commitment_diag.pdf")
>>> PyBoolNet.Commitment.create_piechart(diag, "commitment_pie.pdf")
```

2.8 Phenotypes

To compute phenotypes you need the attractors data and define markers:

```
>>> primes = get_primes("arellano_rootstem")
>>> attrs = PyBoolNet.Attractors.compute_json(primes, "asynchronous")
>>> markers = ["WOX", "MGP"]
>>> phenos = PyBoolNet.Phenotypes.compute_json(attrs, markers)
```

Inspect the json structure with e.g. http://jsonviewer.stack.hu/. Access the existing marker patterns via:

```
>>> for pheno in phenos["phenotypes"]:
...     print(pheno["name"])
...     print(pheno["pattern"])

Pheno A
00
Pheno B
10
Pheno C
01
```

And draw the diagram with phenotypes_compute_diagram:

```
>>> PyBoolNet.Phenotypes.compute_diagram(phenos, FnameImage="phenos_diagram.pdf")
```

2.9 Trap spaces

The main function for computing minimal and maximal trap spaces is trap_spaces:

```
>>> primes = get_primes("remy_tumorigenesis")
>>> mints = PyBoolNet.AspSolver.trap_spaces(primes, "min")
>>> len(mints)
25
>>> maxts = PyBoolNet.AspSolver.trap_spaces(primes, "max")
>>> len(maxts)
8
```

As a special case use the ASP solver to compute all steady states:

```
>>> steady = PyBoolNet.AspSolver.steady_states(primes)
>>> len(steady)
20
```

CHAPTER

THREE

MANUAL

3.1 Importing Boolean Networks

3.1.1 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* 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.

3.1.2 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]
```

3.1.3 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
>>> primes = FileExchange.bnet2primes("example01.bnet")
```

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

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

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

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

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

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

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

```
>>> FileExchange.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.

3.1.4 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.

3.1.5 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))
```

3.2 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 = FileExchange.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.adj["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.adj["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*.

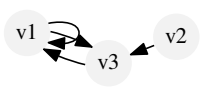


Fig. 1: The interaction graph "*example02_igraph.pdf*" of the network defined above.

3.2.1 graph, node and edge attributes

pyboolnet 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 = FileExchange.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.adj["v3"]["v1"]["arrowhead"] = "inv"
>>> igraph.adj["v3"]["v1"]["color"] = "red"
```

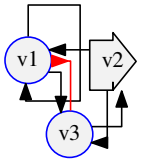
In addition, *dot* graphs have general graph attributes, for example:

- *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

Fig. 2: The interaction graph "*example03_igraph.pdf*".

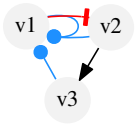
3.2.2 the interaction signs style

pyboolnet 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

Fig. 3: The interaction graph "*example04_igraph.pdf*" with attributes added by add_style_interactionsigns.

3.2.3 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 = FileExchange.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*.

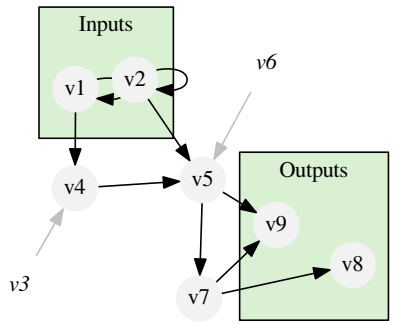
3.2.4 the SCCs style

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 = FileExchange.bnet2primes(bnet)
```

(continues on next page)



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

Fig. 4: The interaction graph "*example05.pdf*" with styles added by add_style_inputs, add_style_outputs and add_style_constants.

(continued from previous page)

```
>>> 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*.

3.2.5 the subgraphs style

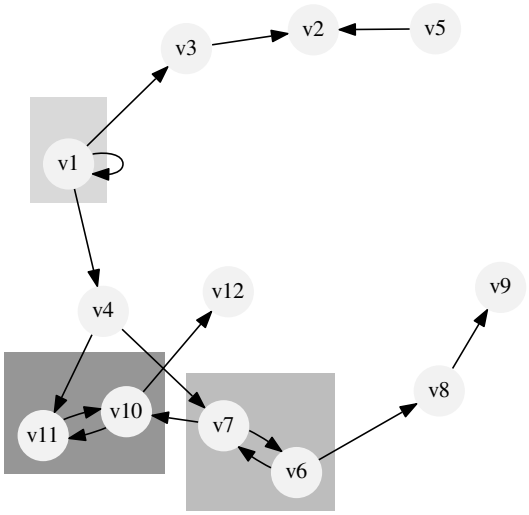
The function add_style_subgraphs allows you to specify subsets of nodes that will be added to a *dot* subgraph. The subgraphs may be specified as a list of pairs that consist of a list of names and a dictionary of *dot* attributes for that subgraph, e.g., a label or background color.

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:

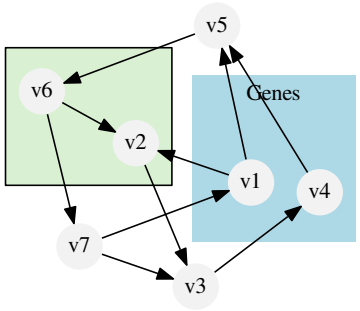
```
>>> bnet = ["v1, v7", "v2, v1 & v6", "v3, v2 | v7", "v4, v3",
...         "v5, v1 | v4", "v6, v5", "v7, v6"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> subgraphs = [(["v2","v6"],{}),
...              (["v1","v4"],{"label":"Genes", "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*.



Example 6: Interaction graph with SCC style

Fig. 5: The interaction graph "*example06_igraph.pdf*" with attributes added by add_style_sccs.



Example 8: Interaction graph with a subgraph style

Fig. 6: The interaction graph "*example08_igraph.pdf*" with attributes added by add_style_subgraphs.

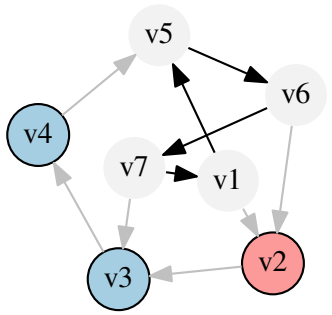
3.2.6 the activities style and animations

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 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 = FileExchange.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

Fig. 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")
```

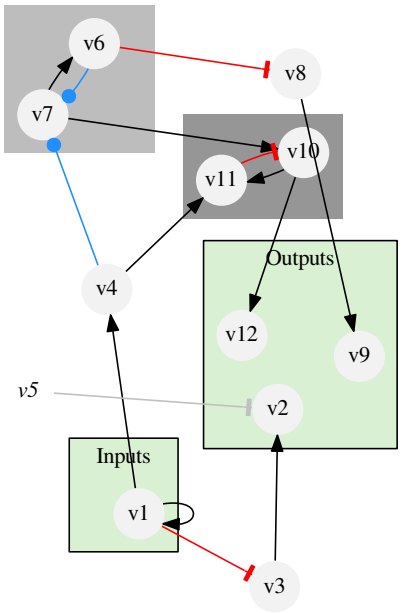
3.2.7 the default style

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 = FileExchange.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

Fig. 8: The interaction graph "*example10_igraph.pdf*" with attributes added by add_style_default.

3.3 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 = FileExchange.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:

```
>>> list(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*.

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}]
```

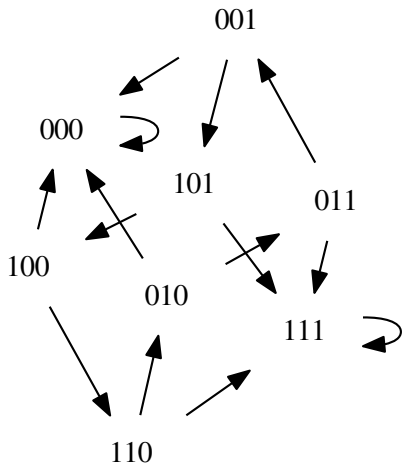
or as a function that is called on every state and must return either *True* or *False* to indicate whether the state ought to be initial:

```
>>> 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:



Example 11: The STG of a positive circuit

Fig. 9: The state transition graph "*example11_stg.pdf*" of an isolated feedback circuit.

```
>>> 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.

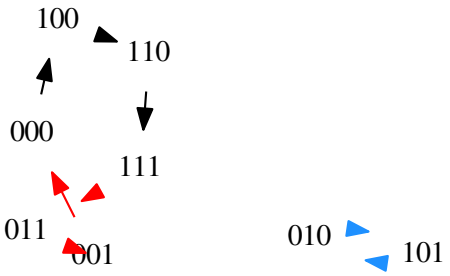
3.3.1 the tendencies style

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 = FileExchange.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*.



Example 12: The synchronous STG of a negative circuit

Fig. 10: The state transition graph "*example12_stg.pdf*" with attributes added by add_style_tendencies.

3.3.2 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 = FileExchange.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 13: STG with path style"
```

Now add the path style:

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

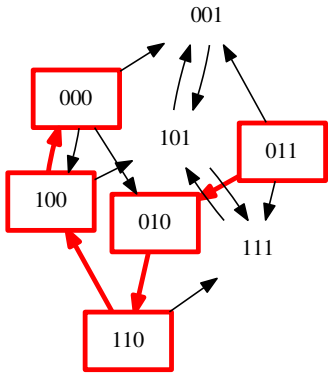
The result is shown in *the figure below*.

3.3.3 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.:

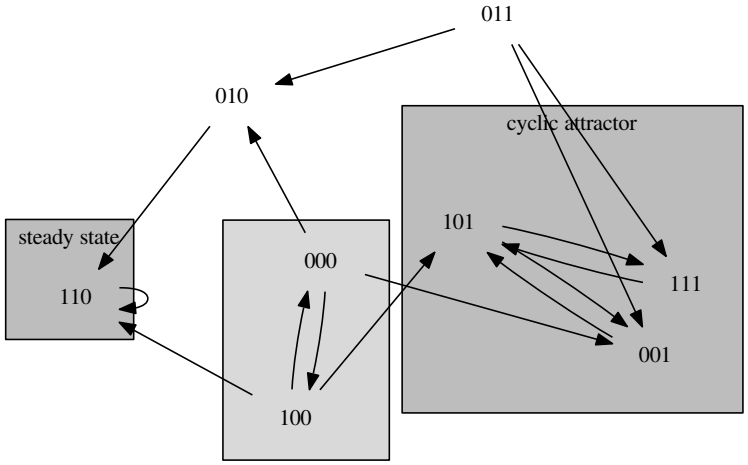
```
>>> bnet = "\n".join(["x, !x|y", "y, x&!y|!z", "z, x&z|!y"])
>>> primes = FileExchange.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "The SCC style"
>>> STGs.add_style_sccs(stg)
>>> STGs.stg2image(stg, "example14_stg.pdf")
```

The result is shown in *the figure below*.



Example 13: STG with path style

Fig. 11: The state transition graph "*example13_stg.pdf*" with attributes added by add_style_path.



Example 14: STG with SCC style

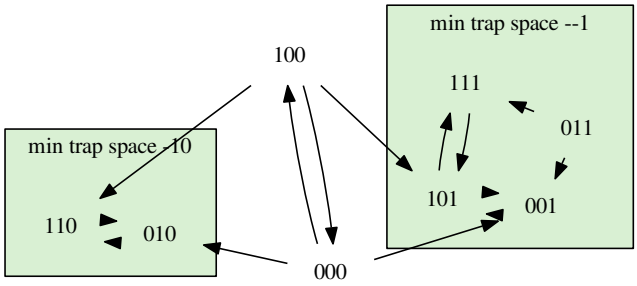
Fig. 12: The state transition graph "*example14_stg.pdf*" with attributes added by add_style_sccs.

3.3.4 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)*:

```
>>> bnet = "\n".join(["x, !x|y&z", "y, x&!y|!z", "z, z|!y"])
>>> primes = FileExchange.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*.



Example 15: STG with min trap spaces style

Fig. 13: The state transition graph "*example15_stg.pdf*" with attributes added by add_style_mintrapspaces.

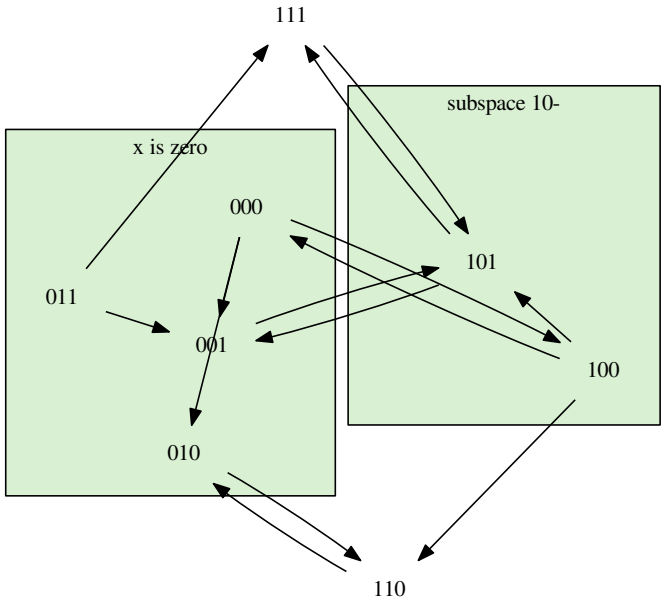
3.3.5 the subspaces style

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 = FileExchange.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 16: STG with subspaces style"
>>> 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.



Example 16: STG with subspaces style

Fig. 14: The state transition graph *"example16_stg.pdf"* with attributes added by add_style_subspaces.

3.3.6 the default style

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

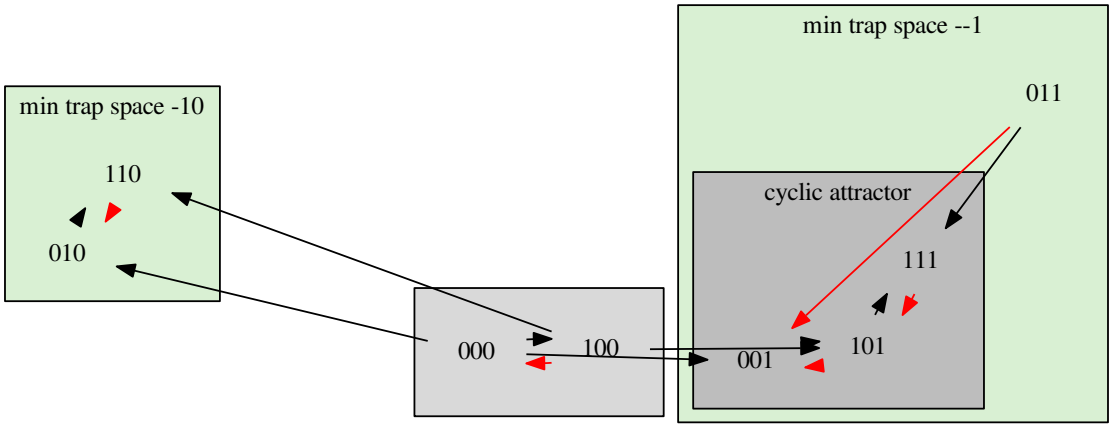
```
>>> bnet = "\n".join(["x, !x|y&z", "y, x&!y|!z", "z, z|!y"])
>>> primes = FileExchange.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 16: STG with default style"
>>> STGs.add_style_default(primes, stg)
>>> STGs.stg2image(stg, "example17_stg.pdf")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

The result is shown in *the figure below*.

3.4 Modifying Networks

3.4.1 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 17: STG with default style

Fig. 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 = FileExchange.bnet2pirmes(bnet)
>>> names = PIs.find_constants(primes)
>>> names
['v1','v2']
>>> PIs.create_blinkers(primes, names)
```

(continues on next page)

(continued from previous page)

```
>>> FileExchange.primes2bnet(primes)
v1,   !v1
v2,   !v2
v3,   v1 & v2 & v3 & v4
v4,   v2 & v3 | v1 & v3
```

Note that *pyboolnet* 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")
>>> FileExchange.primes2bnet(primes)
x,    !x
v2,   !v2
v3,   x & v2 & v3 & v4
v4,   v2 & v3 | x & v3
```

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

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

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

Here, the constants v1 and v3 are removed.

3.4.3 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 = FileExchange.bnet2pirmes(bnet)
>>> PIs.remove_variables(primes, ["v3"])
>>> FileExchange.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 = FileExchange.bnet2primes("v1, v2 \n v2, v1")
>>> create_variables(primes, {"v3": "!v4 | v1", "v4": lambda v1,v2: v1+v2==1})
```

(continues on next page)

(continued from previous page)

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

3.4.4 input combinations

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

```
>>> primes = FileExchange.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}
```

3.5 Model Checking

pyboolnet 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*.

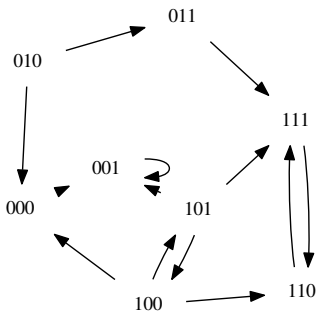
3.5.1 transition systems

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:

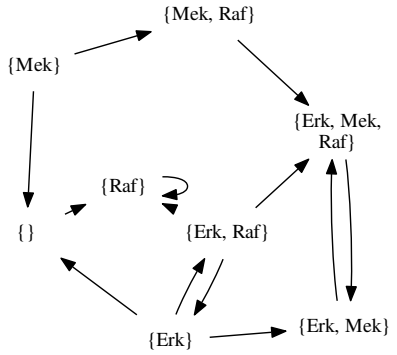
```
>>> bnet = ["Erk,  Erk & Mek | Mek & Raf",
...         "Mek,  Erk | Mek & Raf",
...         "Raf,  !Erk | !Raf"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "Example 18: STG of the Erk-Mek-Raf network"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

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

When model checking, *pyboolnet* 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



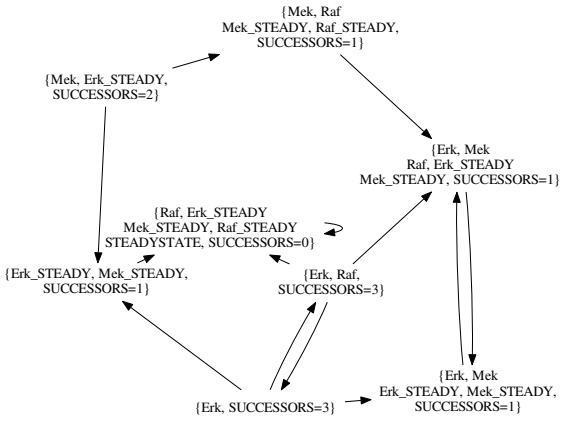
Example 19: Basic transition system of Erk-Mek-Raf

Fig. 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*.



Example 20: Transition system of Erk-Mek-Raf with auxillary variables

Fig. 17: The extended transition system for the Erk-Mek-Raf network.

3.5.2 LTL model checking

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

(continues on next page)

(continued from previous page)

```
>>> 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.

3.5.3 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:

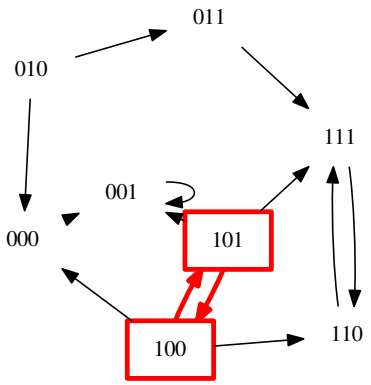
```
>>> answer, counterex = MC.check_primes_with_counterexample(primes, update, init, spec)
```

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")
```



Example 19: A LTL counterexample

Fig. 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))*.

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")
```

3.5.4 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 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 = FileExchange.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:

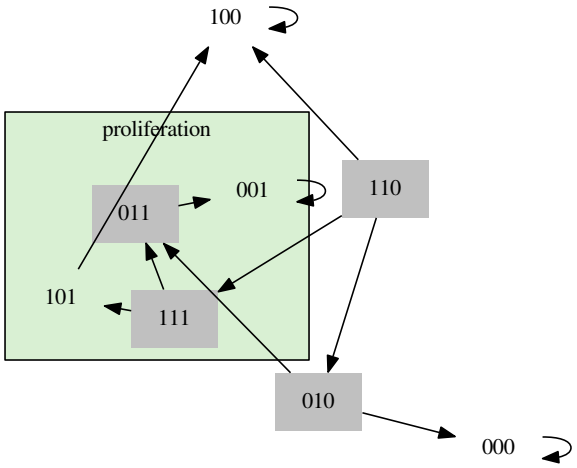
```
>>> stg = STGs.primes2stg(primes, update)
>>> for x in stg.nodes():
...     x_dict = STGs.state2dict(primes, x)
...     if x_dict["GrowthFactor"]:
...         stg.node[x]["style"] = "filled"
...         stg.node[x]["fillcolor"] = "gray"
>>> sub = ({"Proliferation":1},{"label":"proliferation"})
>>> 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

Fig. 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.

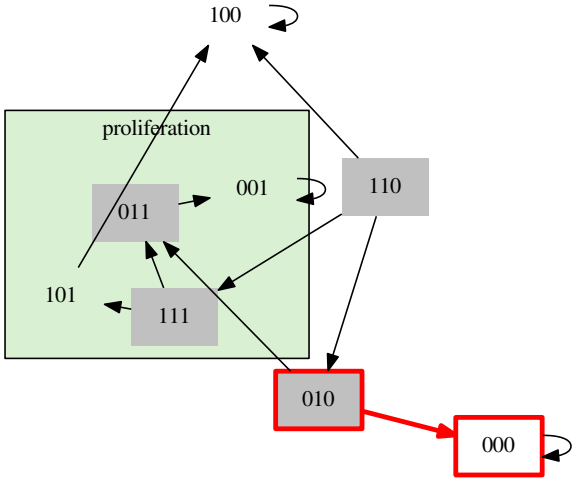
The property can, however, be formulated in CTL using the existential quantifier:

```
>>> 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:



Example 21: Counterexample

Fig. 20: The state transition graph *"example21_stg.pdf"* of the Proliferation network with a counterexample highlighted by path.

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

3.5.5 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]
```

3.5.6 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 = FileExchange.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:

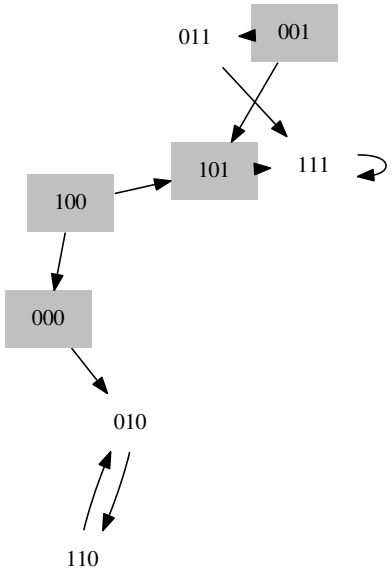
```
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> for x in stg.nodes():
...     if x[1]=="0":
...         stg.node[x]["style"] = "filled"
...         stg.node[x]["fillcolor"] = "gray"
>>> stg.graph["label"] = "Example 22: Existential queries"
>>> STGs.stg2image("example22_stg.pdf")
```

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.

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
```



Example 22: Existential queries

Fig. 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.

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*.

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> pyboolnet.state_space.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> pyboolnet.state_space.state2str(state)
100
```

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

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> pyboolnet.state_space.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> pyboolnet.state_space.state2str(state)
100
```

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

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> STGs.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> pyboolnet.state_space.state2str(state)
100
```

import pyboolnet.state_space >>> notQ2, counterex = MC.check_primes_with_counterexample(primes, update, init, specQ2)

```
>>> state = counterex[0]
>>> pyboolnet.state_space.state2str(state)
100
```

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

3.5.7 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 = FileExchange.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)
```

(continues on next page)

(continued from previous page)

```
>>> 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* 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 enumerate_states to enumerate all states that are referenced by a propositional formula:

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

3.6 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)*.

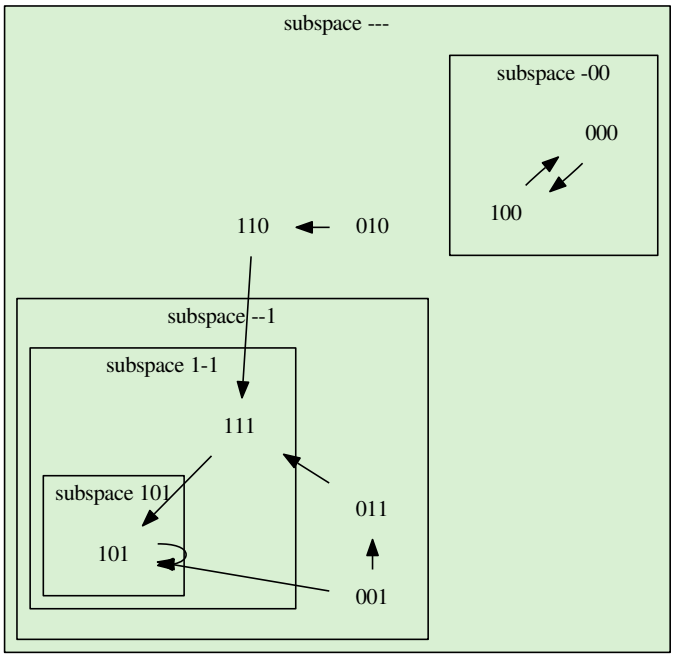
pyboolnet uses the module AspSolver and the function trap_spaces to compute trap spaces. As an example, consider the following network which has five trap spaces:

```
>>> bnet = ["x, !x | y | z",
...         "y, !x&z | y&!z",
...         "z, x&y | z"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.bnet2primes(bnet)
>>> tspaces = AspSolver.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")
```



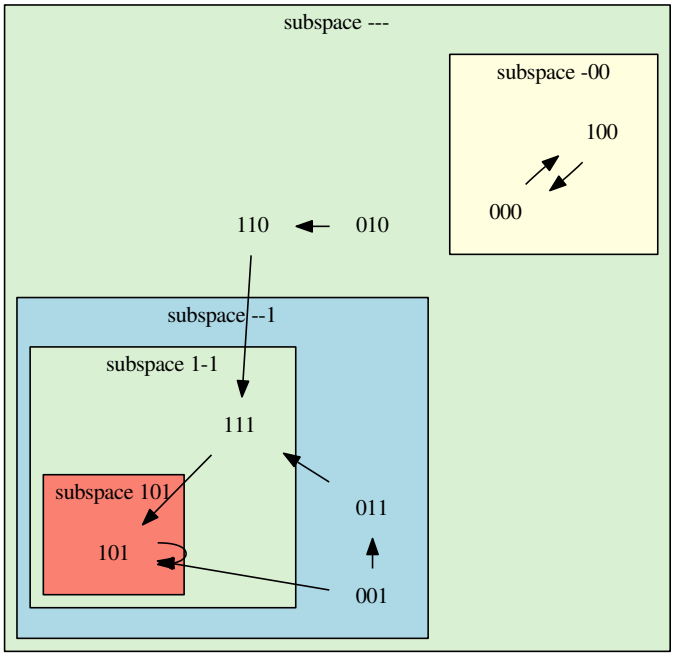
Example 23: All trap spaces

Fig. 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 = AspSolver.trap_spaces(primes, "min")
>>> for x in mintspaces:
...     sub = (x,{"fillcolor":"salmon"})
...     STGs.add_style_subspaces(primes, stg, [sub])
>>> maxtspaces = AspSolver.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

Fig. 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.

3.7 Attractors

3.7.1 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 Attractors as AD
>>> bnet = ["v1, !v1 | v3",
...         "v2, !v1 | v2&!v3",
...         "v3, !v2"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.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 = Attractors.compute_attractors_tarjan(stg)
>>> len(attractors)
2
>>> for A in attractors:
...     print([STGs.state2str(x) for x in A])
['101']
['010', '110']
```

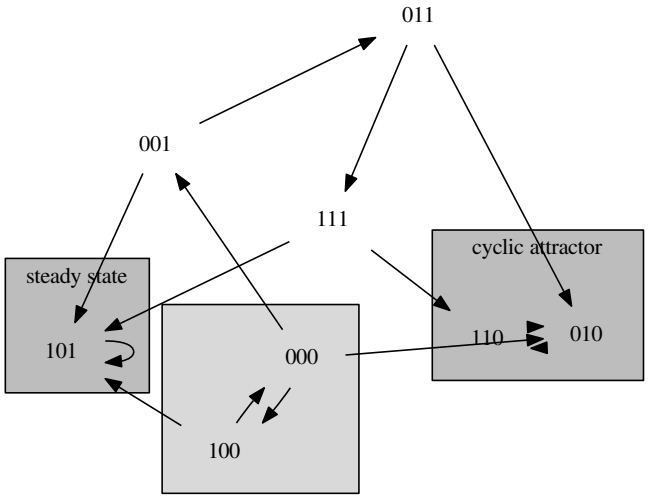
Due to the state space explosion problem, the approach of computing the terminal SCCs by explicitly constructing the underlying STG as a digraph is limited to networks with up to 15 to 20 components.

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:



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

Fig. 24: The asynchronous STG *"example25_stg.pdf"* of a network with a steady state and a cyclic attractor.

```
>>> state = Attractors.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 = Attractors.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")
```

3.7.2 basins of attraction

The module Basins 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 commitment_diagram checks for each possible combination of attractors whether the set of corresponding commitment states is empty or not. If there are states a basin node is created. An edge between commitment nodes indicates the existence of a transition between two states of the respective sets of states. The nodes of a commitment 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 commitment diagram have the following attributes:

- `"EX_formula"` (str), an expression that represents the states that can make the transition
- `"EX_size"` (int), the number of such states

and, if the parameter `AdditionalEdgeData` of commitment_diagram was set to true, there are additionally the attributes:

- `"EF_formula"` (str), an expression that represents the states that can reach a state that can make the transition
- `"EF_size"` (int), the number of such states

Commitment diagrams can be visualized with the function commitment_diagram2image.

Consider the following example:

```
>>> primes = Repository.get_primes("arellano_rootstem")
>>> diagram = Commitment.compute_diagram(primes, "asynchronous", FnameImage="commitment.
↪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 commitment_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 univocality.

3.7.3 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

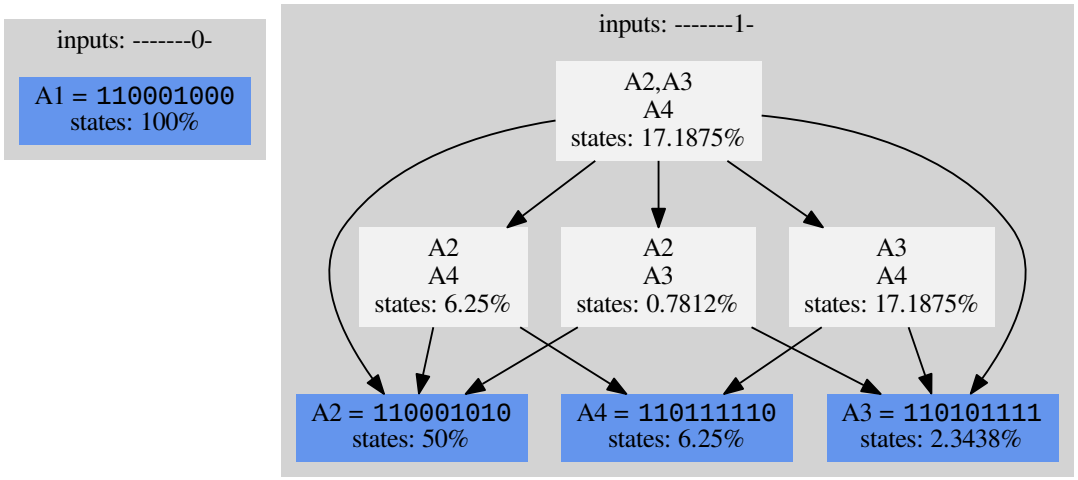


Fig. 25: The basin diagram of the network *arellano_rootstem* from the repository.

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, univocality and faithfulness 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 univocality and faithfulness each require the primes, update strategy and a trap space:

```
>>> update = "asynchronous"
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> for x in mintspaces:
...     answer_univocal = Attractors.univocal(primes, update, x)
...     answer_faithful = Attractors.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.

```
>>> Attractors.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 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.

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous")

>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>>

mintspaces = AspSolver.compute_trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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",
```

(continues on next page)

(continued from previous page)

```
...          "v3, !v1&!v2&v3 | !v1&v2&!v3 | v1&!v2&!v3 | v1&v2&v3"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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",
```

(continues on next page)

(continued from previous page)

```
...         "v3, !v1&!v2&v3 | !v1&v2&!v3 | v1&!v2&!v3 | v1&v2&v3"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.compute_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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”)

>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.compute_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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, “min”) >>> stg = STGs.primes2stg(primes, “asynchronous”) >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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",
```

(continues on next page)

(continued from previous page)

```
...          "v3, !v1&!v2&v3 | !v1&v2&!v3 | v1&!v2&!v3 | v1&v2&v3"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.compute_trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.compute_trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.compute_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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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",
```

(continues on next page)

(continued from previous page)

```
...         "v3, !v1&!v2&v3 | !v1&v2&!v3 | v1&!v2&!v3 | v1&v2&v3"]
>>> bnet = "\n".join(bnet)
>>> primes = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.compute_trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.compute_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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.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)
```

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

import pyboolnet.state_space >>> 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 = FileExchange.bnet2primes(bnet) >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> stg = STGs.primes2stg(primes, "asynchronous") >>> mintspaces = AspSolver.trap_spaces(primes, "min") >>> print [subspace2str(primes, x) for x in mintspaces]

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

```
>>> 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 = FileExchange.bnet2primes(bnet)
>>> mintspaces = AspSolver.trap_spaces(primes, "min")
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> mintspaces = AspSolver.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)
```

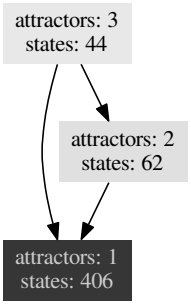


Fig. 26: WRONG FIGURE! The state transition graph *"example25_stg.pdf"* in which the minimal trap space "—" is not univocal.

import pyboolnet.state_space >>> mintspaces = AspSolver.trap_spaces(primes, "min")

```
>>> print [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"
```

(continues on next page)

(continued from previous page)

```
>>> 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

FOUR

REFERENCE

4.1 attractors

completeness(*primes: dict*, *update: str*, *max_output: int = 1000*) → bool

The ASP and CTL model checking based algorithm for deciding whether the minimal trap spaces of a network are complete. The algorithm is discussed in *Klarner2015(a)*. It splits the problem of deciding completeness into smaller subproblems by searching for so-called autonomous sets in the interaction graph of the network. If the minimal trap spaces of the corresponding restricted networks are complete then each of them defines a network reduction that is in turn subjected to a search for autonomous sets. The efficiency of the algorithm depends on the existence of small autonomous sets in the interaction graph, i.e., the existence of "hierarchies" rather than a single connected component.

Note: Completeness depends on the update strategy, i.e., the minimal trap spaces may be complete in the synchronous STG but not complete in the asynchronous STG or vice versa.

Note: The algorithm returns a counterexample, i.e., a state from which there is no path to one of the minimal trap spaces, if the minimal trap spaces are not complete.

Note: Each line that corresponds to a line of the pseudo code of Figure 3 in *Klarner2015(a)* is marked by a comment.

---

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*

returns:

- *answer*: whether *subspaces* is complete in the STG defined by *primes* and *update*,

example:

```
>>> completeness(primes, "asynchronous")
False
```

completeness_naive(*primes*, *update*, *trap_spaces*)

The naive approach to deciding whether *Trapspaces* is complete, i.e., whether there is no attractor outside of *Trapspaces*. The approach is described and discussed in *Klarner2015(a)*. It is decided by a single CTL query of the EF_oneof_subspaces. The state explosion problem limits this function to networks with around 40 variables. For networks with more variables (or a faster answer) use completeness_iterative.

Note: Completeness depends on the update strategy, i.e., a set of subspaces may be complete in the synchronous STG but not complete in the asynchronous STG or vice versa.

Note: A typical use case is to decide whether the minimal trap spaces of a network are complete.

Note: The subspaces of *Trapspaces* are in in fact not required to be a trap sets, i.e., it may contain arbitrary subspaces. If there are arbitrary subspaces then the semantics of the query is such that it checks whether each attractor *intersects* one of the subspaces.

---

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *trap_spaces*: list of subspaces in string or dict representation

returns:

- *answer*: whether *subspaces* is complete in the STG defined by *primes* and *update*,

example:

```
>>> min_trap_spaces = trap_spaces(primes, "min")
>>> answer, counterexample = completeness_naive(primes, "asynchronous", min_trap_
↪spaces)
>>> answer
True
```

completeness_naive_with_counterexample(*primes: dict*, *update: str*, *trap_spaces: List[Union[dict, str]]*)

The naive approach to deciding whether *Trapspaces* is complete, i.e., whether there is no attractor outside of *Trapspaces*. The approach is described and discussed in *Klarner2015(a)*. It is decided by a single CTL query of the EF_oneof_subspaces. The state explosion problem limits this function to networks with around 40 variables. For networks with more variables (or a faster answer) use completeness_iterative.

Note: Completeness depends on the update strategy, i.e., a set of subspaces may be complete in the synchronous STG but not complete in the asynchronous STG or vice versa.

Note: A typical use case is to decide whether the minimal trap spaces of a network are complete.

Note: The subspaces of *Trapspaces* are in in fact not required to be a trap sets, i.e., it may contain arbitrary subspaces. If there are arbitrary subspaces then the semantics of the query is such that it checks whether each

attractor *intersects* one of the subspaces.

---

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *Trapspaces*: list of subspaces in string or dict representation

returns:

- *answer*: whether *subspaces* is complete in the STG defined by *primes* and *update*,
- *counter_example*: a state from which none of the *subspaces* is reachable (if *answer* is *False*)

example:

```
>>> mintspaces = trap_spaces(primes, "min")
>>> answer, counterexample = completeness_naive(primes, "asynchronous", mintspaces)
>>> answer
True
```

completeness_with_counterexample(*primes: dict, update: str, max_output: int = 1000) -> (<class 'bool'>, typing.List[dict]*)

Performs the same steps as completeness but also returns a counterexample which is *None* if it does not exist. A counterexample of a completeness test is a state that can not reach one of the minimal trap spaces of *primes*.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*

returns:

- *answer*: whether *subspaces* is complete in the STG defined by *primes* and *update*,
- *counterexample*: a state that can not reach one of the minimal trap spaces of *primes* or *None* if no counterexample exists

example:

```
>>> answer, counterexample = completeness_with_counterexample(primes, "asynchronous
↪")
>>> answer
False
>>> state2str(counterexample)
'10010111101010100001100001011011111111'
```

compute_attractors(*primes: dict, update: str, fname_json: Optional[str] = None, check_completeness: bool = True, check_faithfulness: bool = True, check_univocality: bool = True, max_output: int = 1000*) → dict

computes all attractors of *primes* including information about completeness, univocality, faithfulness

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*

- *fname_json*: json file name to save result
- *check_completeness*: enable completeness check
- *check_faithfulness*: enable faithfulness check
- *check_univocality*: enable univocality check

returns:

- *attractors*: attractor data

example::

```
>>> attractors = compute_attractors(primes, update, "attractors.json")
```

compute_attractors_tarjan(*stg: DiGraph*)

Uses networkx.strongly_connected_components , i.e., Tarjan's algorithm with Nuutila's modifications, to compute the SCCs of *stg* and networkx.has_path to decide whether a SCC is reachable from another. Returns the attractors as lists of states.

arguments:

- *stg*: state transition graph

returns:

- *steady_states*: the steady states
- *cyclic_attractors*: the cyclic attractors

example:

```
>>> bnet = ["x, !x&y | z",
...         "y, !x | !z",
...         "z, x&!y"]
>>> bnet = "\n".join(bnet)
>>> primes = bnet2primes(bnet)
>>> stg = primes2stg(primes, "asynchronous")
>>> steady_states, cyclic_attractors = compute_attractors_tarjan(stg)
>>> steady_states
["101","000"]
>>> cyclic_attractors
[{"111", "110"}, {"001", "011"}]
```

create_attractor_report(*primes: dict*, *fname_txt: Optional[str] = None*) → str

Creates an attractor report for the network defined by *primes*.

arguments:

- *primes*: prime implicants
- *fname_txt*: the name of the report file or *None*

returns:

- *txt*: attractor report as text

example::

```
>>> create_attractor_report(primes, "report.txt")
```

faithfulness(*primes: dict*, *update: str*, *trap_space: Union[dict, str]*) → bool

The model checking approach for deciding whether *trap_space* is faithful, i.e., whether all free variables oscillate in all of the attractors contained in it, in the state transition graph defined by *primes* and *update*. The approach is described and discussed in *Klarner2015(a)*. It is decided by a single CTL query of the pattern EF_all_unsteady and the random-walk-approach of the function random_walk.

Note: In the (very unlikely) case that the random walk does not end in an attractor state an exception will be raised.

Note: Faithfulness depends on the update strategy, i.e., a trapspace may be faithful in the synchronous STG but not faithful in the asynchronous STG or vice versa.

Note: A typical use case is to decide whether a minimal trap space is faithful.

Note: *trap_space* is in fact not required to be a trap set, i.e., it may be an arbitrary subspace. If it is an arbitrary subspace then the involved variables are artificially fixed to be constant.

---

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *trap_space*: a subspace

returns:

- *answer*: whether *trap_space* is faithful in the STG defined by *primes* and *update*

example:

```
>>> mintspaces = compute_trap_spaces(primes, "min")
>>> x = mintspaces[0]
>>> faithfulness(primes, x)
True
```

faithfulness_with_counterexample(*primes: dict, update: str, trap_space: dict) -> (<class 'bool'>, typing.List[dict]*)

Performs the same steps as faithfulness but also returns a counterexample which is *None* if it does not exist. A counterexample of a faithful test is a state that belongs to an attractor which has more fixed variables than there are in *trap_space*.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *trap_space*: a subspace

returns:

- *answer*: whether *trap_space* is faithful in the STG defined by *primes* and *update*

- *counter_example*: a state that belongs to an attractor that does not oscillate in all free variables or *None* if no counterexample exists

example:

```
>>> mintspaces = compute_trap_spaces(primes, "min")
>>> x = mintspaces[0]
>>> faithfulness(primes, x)
True
```

find_attractor_state_by_randomwalk_and_ctl(*primes: dict*, *update: str*, *initial_state: Union[dict, str] = {}*, *length: int = 0*, *attempts: int = 10*) → dict

Attempts to find a state inside an attractor by the "long random walk" method, see *Klarner2015(b)* Sec. 3.2 for a formal definition. The method generates a random walk of *length* states, starting from a state defined by *initial_state*. If *initial_state* is a subspace then random_state will be used to draw a random state from inside it. The function then uses CTL model checking, i.e., check_primes, to decide whether the last state of the random walk is inside an attractor. If so it is returned, otherwise the process is repeated. If no attractor state has been found after *Attempts* many trials an exception is raised.

Note: The default value for length, namely *length=0*, will be replaced by:

```
length = 10*len(primes)
```

which proved sufficiently large in our computer experiments.

---

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *initial_state*: an initial state or subspace
- *length*: length of random walk
- *attempts*: number of attempts before exception is raised

returns:

- *attractor_state*: a state that belongs to some attractor
- raises *Exception* if no attractor state is found

example::

```
>>> find_attractor_state_by_randomwalk_and_ctl(primes, "asynchronous")
{"v1": 1, "v2": 1, "v3": 1}
```

intersection(**list_of_dicts*)

each argument must be a list of subspaces (dicts):

```
>>> intersection([{"v1":1}], [{"v1":0}, {"v2":1, "v3":0}])
```

iterative_completeness_algorithm(*primes: dict*, *update: str*, *compute_counterexample: bool*, *max_output: int = 1000*) → Union[Tuple[bool, Optional[dict]], bool]

The iterative algorithm for deciding whether the minimal trap spaces are complete. The function is implemented

by line-by-line following of the pseudo code algorithm given in "Approximating attractors of Boolean networks by iterative CTL model checking", Klarner and Siebert 2015.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *compute_counterexample*: whether to compute a counterexample

returns:

- *answer*: whether *subspaces* is complete in the STG defined by *primes* and *update*,
- *counterexample*: a state that can not reach one of the minimal trap spaces of *primes* or *None* if no counterexample exists

example:

```
>>> answer, counterexample = completeness_with_counterexample(primes, "asynchronous
↪")
>>> answer
False
>>> state2str(counterexample)
1001011110101010000110000101101111111
```

`read_attractors_json`(*fname: str*) → dict

opens the attractor object

arguments:

- *fname*: file name

returns:

- *attractors*: json attractor data, see compute_attractors

example::

```
>>> attractors = read_attractors_json("attractors.json")
```

`univocality`(*primes: dict*, *update: str*, *trap_space: Union[dict, str]*) → bool

The model checking and random-walk-based method for deciding whether *trap_space* is univocal, i.e., whether there is a unique attractor contained in it, in the state transition graph defined by *primes* and *update*. The approach is described and discussed in *Klarner2015(a)*. The function performs two steps: first it searches for a state that belongs to an attractor inside of *trap_space* using the random-walk-approach and the function random_walk, then it uses CTL model checking, specifically the pattern AGEF_oneof_subspaces, to decide if the attractor is unique inside *trap_space*.

Note: In the (very unlikely) case that the random walk does not end in an attractor state an exception will be raised.

Note: Univocality depends on the update strategy, i.e., a trapspace may be univocal in the synchronous STG but not univocal in the asynchronous STG or vice versa.

Note: A typical use case is to decide whether a minimal trap space is univocal.

Note: *trap_space* is in fact not required to be a trap set, i.e., it may be an arbitrary subspace. If it is an arbitrary subspace then the involved variables are artificially fixed to be constant.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *trap_space*: a subspace

returns:

- *answer*: whether *trap_space* is univocal in the STG defined by *primes* and *update*

example:

```
>>> mintspaces = compute_trap_spaces(primes, "min")
>>> x = mintrapspaces[0]
>>> univocality(primes, "asynchronous", x)
True
```

univocality_with_counterexample(*primes: dict, update: str, trap_space: ~typing.Union[dict, str]) -> (<class 'bool'>, typing.List[dict]*)

Performs the same steps as univocality but also returns a counterexample which is *None* if it does not exist. A counterexample of a univocality test are two states that belong to different attractors.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *trap_space*: a subspace

returns:

- *answer*: whether *trap_space* is univocal in the STG defined by *primes* and *update*
- *counter_example*: two states that belong to different attractors or *None* if no counterexample exists

example:

```
>>> mintspaces = compute_trap_spaces(primes, "min")
>>> trapspace = mintrapspaces[0]
>>> answer, counterex = univocality_with_counterexample(primes, trapspace,
↪"asynchronous")
```

write_attractors_json(*attractors: dict*, *fname_json: str*)

saves the attractor object as a JSON file.

arguments:

- *attractors*: json attractor data, see attractors_compute_json
- *fname_json*: file name

example::

```
>>> save_attractor(attractors, "attractors.json")
```

4.2 basins_of_attraction

compute_basins(*attractors: dict*, *weak: bool = True*, *strong: bool = True*, *cycle_free: bool = True*, *fname_barplot: Optional[str] = None*, *fname_piechart: Optional[str] = None*, *minimize: bool = False*) → Optional[dict]

Extends *attractors* with basin of attraction. Use *fname_barplot* and *FnamePiechart* to create plots of the basins, see create_barplot and basins_create_piechart.

arguments:

- *attractors*: json attractor data, see attractors_compute_json
- *weak*: compute weak basins
- *strong*: compute strong basins
- *cycle_free*: compute cycle-free basins
- *fname_barplot*: file name of bar plot
- *fname_piechart*: file name of pie chart
- *minimize*: minimize the Boolean expressions

example:

```
>>> primes = get_primes("raf")
>>> attractors = compute_attractors(primes, update)
>>> compute_basins(attractors)
```

create_basins_of_attraction_barplot(*attractors: dict*, *fname_image: str*, *title: Optional[str] = None*, *y_unit: Optional[str] = 'perc'*, *y_max: Optional[str] = None*, *labels_map: Optional[callable] = None*)

Creates a bar plot of the basins of attraction specified in *attractors*. Requires that *attractors* has been extended with basins information by compute_basins. Requires https://matplotlib.org.

arguments:

- *attractors*: json attractor data, see attractors_compute_json
- *fname_image*: create image for bar plot
- *title*: optional title of plot
- *Yunit*: "perc" for percentage of state space and "size" for number of states
- *Ymax*: y axis limit
- *LabelsMap* (function): a map from minimal trap space dictionary of attractor to label str

example:

```
>>> attractors = compute_attractors(primes, update)
>>> compute_basins(attractors)
>>> create_basins_of_attraction_barplot(attractors, "barplot.pdf")
created barplot.pdf
```

create_basins_piechart(*attractors: dict*, *fname_image: str*, *title: Optional[str] = None*, *y_unit: Optional[str] = 'perc'*, *labels_map: Optional[callable] = None*)

Creates a pie chart of the basins of attraction specified in *attractors*. Requires that *attractors* has been extended with basins information by compute_basins. Requires https://matplotlib.org.

arguments:

- *attractors*: json attractor data, see attractors_compute_json
- *fname_image*: create image for pie chart
- *title*: optional title of plot
- *y_unit*: "perc" for percentage of state space and "size" for number of states
- *labels_map*: a map from minimal trap space dictionary of attractor to label str

example:

```
>>> attractors = compute_attractors(primes, update)
>>> compute_basins(attractors)
>>> create_basins_piechart(attractors, "piechart.pdf")
created piechart.pdf
```

cycle_free_basin(*primes: dict*, *update: str*, *subspace: Union[dict, str] = {}*, *minimize: bool = False*)

Computes the cycle-free basin of *subspace* via the CTL query AG(EF(Subspace)), for details see *Klarner2018*.

arguments: * *primes*: prime implicants * *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"* * *minimize*: minimize the Boolean expressions * *subspace*: a subspace

returns: * *basin*: with keys "size"=number of states, "formula"=state formula and "perc"=percentage of state space

example:

```
>>> cycle_free_basin(primes, update, True, "0---1")
```

{"size": 134, "formula": "Erk & !Raf | Mek", "perc": 12.89338}

strong_basin(*primes*, *update*, *subspace: Union[dict, str] = {}*, *minimize: bool = False*)

Computes the strong basin of *subspace* via the CTL query AG(EF(Subspace)), for details see *Klarner2018*.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *minimize*: minimize the Boolean expressions
- *subspace*: a subspace

returns:

- *basin*: with keys "size"=number of states, "formula"=state formula and "perc"=percentage of state space

example:

```
>>> strong_basin(primes, update, True, "0---1")
{"size":    134,
"formula":    "Erk & !Raf | Mek",
"perc":        12.89338}
```

weak_basin(*primes: dict*, *update: str*, *subspace: Union[dict, str] = {}*, *minimize: bool = False*)

Computes the weak basin of *subspace* via the CTL query AG(EF(Subspace)), for details see *Klarner2018*.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *minimize*: minimize the Boolean expressions
- *subspace*: a subspace

returns:

- *basin*: with keys "size"=number of states, "formula"=state formula and "perc"=percentage of state space

example:

```
>>> weak_basin(primes, update, "0---1", True)
{"size": 134,
"formula": "Erk & !Raf | Mek",
"perc": 12.89338}
```

4.3 boolean_logic

are_equivalent(*this: str*, *other: str*) → bool

uses minimize_espresso to compute whether *this* and *other* are equivalent.

are_mutually_exclusive(*this: str*, *other: str*) → bool

uses minimize_espresso to compute whether A and B are mutually exclusive

implies(*this: str*, *other: str*) → bool

uses minimize_espresso to compute whether *this* implies *other*.

minimize_espresso(*expression: str*, *fname_out: Optional[str] = None*, *merge: bool = False*, *equiv: bool = False*, *exact: bool = False*, *reduce: bool = False*) → str

Tries to minimize a given boolean expression utilizing the heuristic minimization algorithm espresso and eqntott for its input preparation. Resulting expression is saved in file if filename for output is specified. The argument *expression* may be either the name of the input file containing the boolean expression or the string representing the expression itself. The input expression may not contain the following words: *False*, *FALSE*, *True*, *TRUE*, *Zero*, *ZERO*, *One*, *ONE*.

arguments:

- *expression*: name of file containing the expression or string contents of file
- *fname_out*: name of the file to write the output to
- *merge*: performs distance-1 merge on input, useful if very large
- *equiv*: identifies equivalent output variables

- *exact*: performs exact minimization algorithm, guarantees minimum number of product terms and heuristically minimizes number of literals, potentially expensive
- *reduce*: eqntott tries to reduce the size of the truth table by merging minterms

returns:

- *minimized_expression*: the minimized expression

example:

```
>>> minimize_espresso("bool_function.txt", "minimized_function.txt" )
>>> minimize_espresso("var = (a & b) | a;")
>>> minimize_espresso("var = 1")
>>> minimize_espresso("(a & b) | a")
```

4.4 boolean_normal_forms

bit_count(*i*)

Count set bits of the input.

functions2mindnf(*functions: Dict[str, callable]*) → Dict[str, str]

Generates and returns a minimal *disjunctive normal form* (DNF) for the Boolean network represented by *functions*. The algorithm uses *Prekas2012*, a Python implementation of the Quine-McCluskey algorithm.

arguments:

- *functions*: keys are component names and values are Boolean functions

returns:

- *min_dnf*: keys are component names and values are minimal DNF expressions

example:

```
>>> funcs = {"v1": lambda v1,v2: v1 or not v2, "v2": lambda: 1}
>>> mindnf = functions2primes(funcs)
>>> mindnf["v1"]
((! v2) | v1)
```

functions2primes(*functions: Dict[str, callable]*) → dict

Generates and returns the prime implicants of a Boolean network represented by *functions*.

arguments:

- *functions*: keys are component names and values are Boolean functions

returns:

- *primes*: primes implicants

example:

```
>>> funcs = {"v1": lambda v1, v2: v1 or not v2,
...          "v2": lambda v1, v2: v1 + v2 == 1}
>>> primes = functions2primes(funcs)
```

get_dnf(*one_implicants: List[dict]*) → str

Returns a disjunctive normal form for *name* by converting each prime implicant into a conjunction.

arguments:

- *one_implicants*: the 1-implicants of a component

returns:

- *dnf*: the dnf of *one_implicants*

example:

```
>>> get_dnf(primes["MEK"][1])
RAF&ERK | TGF
```

is_power_of_two_or_zero(*x*)

Determine if an input is zero or a power of two. Alternative, determine if an input has at most 1 bit set.

merge(*i*, *j*)

Combine two minterms.

primes2mindnf(*primes: dict*) → Dict[str, str]

Creates a minimal *disjunctive normal form* (DNF) expression for the Boolean network represented by *primes*. The algorithm uses *Prekas2012*, a Python implementation of the Quine-McCluskey algorithm.

arguments:

- *primes*: prime implicants

returns:

- *min_dnf*: keys are names and values are minimal DNF expressions

example:

```
>>> primes["v1"][1]
[{'v1':1,'v2':0}]
>>> mindnf = primes2mindnf(primes)
>>> mindnf["v1"]
((! v2) | v1)
```

4.5 commitment_diagrams

commitment_diagram2image(*diagram: DiGraph*, *fname_image: str*, *style_inputs: bool = True*, *style_edges: bool = False*, *style_ranks: bool = True*, *first_index: int = 1*) → DiGraph

Creates the image file *fname_image* for the basin diagram given by *diagram*. The flag *StyleInputs* can be used to highlight which basins belong to which input combination. *style_edges* adds edge labels that indicate the size of the "border" (if *compute_border* was enabled in commitment_compute_diagram) and the size of the states of the source basin that can reach the target basin.

arguments:

- *diagram*: a commitment diagram
- *fname_image*: file name of image
- *style_inputs*: whether basins should be grouped by input combinations

- *style_edges*: whether edges should be size of border / reachable states
- *style_ranks*: style that places nodes with the same number of reachable attractors on the same rank (level)
- *first_index*: first index of attractor names

returns::

- *styled_diagram*: the styled commitment diagram

example:

```
>>> attractors = compute_attractors(primes, update)
>>> compute_phenotype_diagram(attractors)
>>> commitment_diagram2image(diagram, "diagram.pdf")
```

commitment_diagrams_are_equivalent(*this: DiGraph*, *other: DiGraph*) → bool

Checks whether diagrams *this* and *other* are equivalent.

compute_commitment_diagram(*attractors: dict*, *fname_image: Optional[str] = None*, *fname_json=None*, *edge_data=False*) → DiGraph

Computes the commitment diagram for the AttrJson and STG defined in *attractors*, a json object computed by AttrJson_compute_json The nodes of the diagram represent states that can reach the exact same subset of *attractors*. Edges indicate the existence of a transition between two nodes in the respective commitment sets. Edges are labeled by the number of states of the source set that can reach the target set and, if *EdgeData* is true, additionally by the size of the border.

arguments:

- *attractors*: json attractor data, see compute_attractors
- *fname_image*: generate image for diagram
- *fname_json*: save diagram as json
- *edge_data*: toggles computation of additional edge data

returns::

- *diagram*: the commitment diagram

example:

```
>>> attractors = compute_attractors(primes, update)
>>> diagram = compute_phenotype_diagram(attractors)
```

create_commitment_piechart(*diagram: DiGraph*, *fname_image: str*, *color_map: Optional[dict] = None*, *title: Optional[str] = None*)

Creates the commitment pie chart for the commitment diagram using matplotlib. The pieces of the chart represent states that can reach the exact same subset of *Attractors*.

arguments:

- *diagram*: commitment diagram, see commitment_compute_diagram
- *fname_image*: name of the output image
- *color_map*: assignment of diagram nodes to colors for custom colors
- *title*: optional title of plot

example:

```
>>> primes = get_primes("xiao_wnt5a")
>>> attractors = compute_attractors(primes, update)
>>> diagram = compute_commitment_diagram(attractors)
>>> create_commitment_piechart(diagram, "commitment_piechart.pdf")
```

open_commitment_diagram(*fname_json: str*) → DiGraph

Opens a commitment diagram json file. See also commitment_compute_diagram, commitment_save_diagram.

arguments:

- *fname_json*: a json file name

returns:

- *diagram*: the commitment diagram

example:

```
>>> diagram = open_commitment_diagram("commitment_diagram.json")
```

save_commitment_diagram(*diagram: DiGraph*, *fname_json: str*)

Saves a commitment diagram as a json file.

arguments:

- *diagram*: a commitment diagram
- *fname_json*: json file name

example:

```
>>> save_commitment_diagram(diagram, "commitment_diagram.json")
```

4.6 file_exchange

bnet2primes(*bnet: str*, *fname_primes: Optional[str] = None*) → dict

Generates and returns the prime implicants of a Boolean network in **boolnet** format. The primes are saved as a *json* file if *fname_primes* is given. The argument *bnet* may be either the name of a *bnet* file or a string containing the file contents. Use the function *read_primes* to open a previously saved *json* file.

arguments:

- *bnet*: name of *bnet* file or string contents of file
- *fname_primes*: *None* or name of *json* file to save primes

returns:

- *primes*: prime implicants

example:

```
>>> primes = bnet2primes("mapk.bnet", "mapk.primes" )
>>> primes = bnet2primes("mapk.bnet")
>>> primes = bnet2primes("Erk, !Mek \n Raf, Ras & Mek")
>>> primes = bnet2primes("Erk, !Mek \n Raf, Ras & Mek", "mapk.primes")
```

primes2bnet(*primes: dict*, *fname_bnet: Optional[str] = None*, *minimize: bool = False*, *header: bool = False*) → str

Saves *primes* as a *bnet* file, including the header *"targets, factors"* for compatibility with *BoolNet*. Without minimization, the resuting formulas are disjunctions of all prime implicants and may therefore be very long. If *Minimize=True* then a Python version of the Quine-McCluskey algorithm, namely *Prekas2012* which is implemented in QuineMcCluskey.primes2mindnf, will be used to minimize the number of clauses for each update function.

arguments:

- *primes*: prime implicants
- *fname_bnet*: name of *bnet* file or *None* for the string of the file contents
- *minimize*: minimize the Boolean expressions
- *header*: whether to include the "targets, factors" header

returns:

- *text_bnet*: str contents of bnet file

example:

```
>>> primes2bnet(primes, "mapk.bnet")
>>> primes2bnet(primes, "mapk.bnet", True)
>>> expr = primes2bnet(primes)
>>> expr = primes2bnet(primes, True)
```

primes2bns(*primes: dict*, *fname_bns: Optional[str] = None*) → str

Saves Primes as a *bns* file for the computation of all attractors of the synchronous transition system. **BNS_** is based on *Dubrova2011*. It is available at http://people.kth.se/~dubrova/bns.html.

arguments:

- *primes*: prime implicants
- *fname_bns*: name of *bns* file or *None* to return file as string

example:

```
>>> primes2bns(primes, "mapk.bns")
```

primes2eqn(*primes: dict*, *fname_eqn: Optional[str] = None*) → str

Generates a *eqn* file as specified in the manual for the model checking software **Antelope_** from *primes*. **Antelope_** was introduced in *Arellano2011*.

arguments:

- *primes*: prime implicants
- *fname_eqn*: name of *eqn* file

example:

```
>>> primes2eqn(primes, "mapk.eqn")
```

primes2genysis(*primes: dict*, *fname_genysis: str*)

Generates a **GenYsis_** file from *primes* for the computation of all attractors of the synchronous or asynchronous transition system. **GenYsis_** was introduced in *Garg2008*. It is available at http://www.vital-it.ch/software/genYsis.

arguments:

- *primes*: prime implicants
- *fname_genysis*: name of **GenYsis_** file

example:

```
>>> primes2genysis(primes, "mapk.genysis")
```

read_primes(*fname_json: str*) → dict

Reads the prime implicants of a Boolean network that were previously stored as a *json* file.

arguments:

- *fname_json*: name of *json* file

returns:

- *primes*: prime implicants

example:

```
>>> primes = read_primes("mapk.primes")
```

write_primes(*primes: dict*, *fname_json: str*)

Saves *primes* as a *json* file.

arguments:

- *primes*: prime implicants
- *fname_json*: name of *json* file

example:

```
>>> write_primes(primes, "mapk.primes")
```

4.7 interaction_graphs

activities2animation(*igraph: DiGraph*, *activities*, *fname_gif: str*, *fname_tmp: str = 'tmp*.jpg'*, *delay: int = 50*, *loop: int = 0*)

Generates an animated *gif* from the sequence of *Activities* by mapping the activities on the respective components of the interaction graph using add_style_activities. The activities may be given in *dict* or *str* format, see *states, subspaces and paths* for details. Requires the program *convert* from the *ImageMagick* software suite. The argument *FnameTMP* is the string that is used for generating the individual frames. Use "*" to indicate the position of the frame counter. The default *"tmp*.jpg"* will result in the creation of the files:

```
tmp01.jpg, tmp02.jpg, ...
```

The files will be deleted after the *gif* is generated. The *Delay* parameter sets the frame rate and *Loop* the number of repititions, both are parameters that are directly passed to *convert*.

arguments:

- *igraph*: interaction graph
- *Activities*: sequence of activities
- *Delay*: number of 1/100s between each frame

- *Loop*: number of repetitions, use 0 for infinite
- *FnameTMP*: name for temporary image files, use "*" to indicate counter
- *FnameGIF*: name of the output *gif* file

example:

```
>>> activities = ["11--1-0", "111-1-0", "11111-0", "1111100"]
>>> activities2animation(igraph, activities, "animation.gif")
```

add_style_activities(*igraph: DiGraph*, *activities: Union[str, dict]*, *color_active: str = '/paired10/5'*, *color_inactive: str = '/paired10/1'*)

Sets attributes for the color and fillcolor of nodes to indicate which variables are activated and which are inhibited in *Activities*. All activated or inhibited components get the attribute *"color"="black"*. Activated components get the attribute *"fillcolor"="red"* and inactivated components get the attribute *"fillcolor"="blue"*. Interactions involving activated or inhibited nodes get the attribute *"color"="gray"* to reflect that they are ineffective.

arguments:

- *igraph*: interaction graph
- *activities*: activated and inhibited nodes
- *color_active*: color in dot format for active components
- *color_inactive*: color in dot format for inactive components

example:

```
>>> activities = {"ERK":1, "MAPK":0}
>>> add_style_activities(igraph, activities)
```

add_style_anonymous(*igraph: DiGraph*)

Creates an anonymous interaction graph with circular nodes without labels.

arguments:

- *igraph*: interaction graph

example:

```
>>> add_style_anonymous(igraph)
```

add_style_constants(*igraph: DiGraph*)

Sets the attribute *"style"="plaintext"* with *"fillcolor"="none"* and *"fontname"="Times-Italic"* for all constants.

arguments:

- *igraph*: interaction graph

example:

```
>>> add_style_constants(igraph)
```

add_style_default(*igraph: DiGraph*)

A convenience function that adds styles for interaction signs, SCCs, inputs, outputs and constants.

arguments:

- *igraph*: interaction graph

example:

```
>>> add_style_default(igraph, path)
```

add_style_inputs(*igraph: DiGraph*)

Adds a subgraph to the *dot* representation of *igraph* that contains all inputs. Nodes that belong to the same *dot* subgraph are contained in a rectangle and treated separately during layout computations. In addition, the subgraph is labeled by a "Inputs" in bold font.

arguments:

- *igraph*: interaction graph

example:

```
>>> add_style_inputs(igraph)
```

add_style_interactionsigns(*igraph: DiGraph*)

Sets attributes for the arrow head and edge color of interactions to indicate the interaction sign. Activating interactions get the attributes *"arrowhead"="normal"* and *"color"="black"*, inhibiting interactions get the attributes *"arrowhead"="tee"* and *"color"="red"*, and ambivalent interaction get the attributes *"arrowhead"="dot"* and *"color"="blue"*.

arguments:

- *igraph*: interaction graph

example:

```
>>> add_style_interactionsigns(igraph)
```

add_style_outputs(*igraph: DiGraph*)

Adds a subgraph to the *dot* representation of *igraph* that contains all outputs. Nodes that belong to the same *dot* subgraph are contained in a rectangle and treated separately during layout computations. In addition, the subgraph is labeled by a *"Outputs"* in bold font.

arguments:

- *igraph*: interaction graph

example:

```
>>> add_style_outputs(igraph)
```

add_style_path(*igraph: DiGraph*, *path: List[str]*, *color: str*)

Sets the color of all nodes and edges involved in the given *path* to *color*.

arguments:

- *igraph*: interaction graph
- *path*: sequence of component names
- *color*: color of the path

example:

```
>>> path = ["Raf", "Ras", "Mek"]
>>> add_style_path(igraph, path, "red")
```

add_style_sccs(*igraph: DiGraph*)

Adds a subgraph for every non-trivial strongly connected component (SCC) to the *dot* representation of *igraph*. Nodes that belong to the same *dot* subgraph are contained in a rectangle and treated separately during layout computations. Each subgraph is filled by a shade of gray that gets darker with an increasing number of SCCs that are above it in the condensation graph. Shadings repeat after a depth of 9.

arguments:

- *igraph*: interaction graph

example:

```
>>> add_style_sccs(igraph)
```

add_style_subgraphs(*igraph: DiGraph*, *subgraphs*)

Adds the subgraphs given in *subgraphs* to *igraph* - or overwrites them if they already exist. Nodes that belong to the same *dot* subgraph are contained in a rectangle and treated separately during layout computations. *subgraphs* must consist of tuples of the form *NodeList*, *Attributs* where *NodeList* is a list of graph nodes and *Attributes* is a dictionary of subgraph attributes in *dot* format.

Note: *subgraphs* must satisfy the following 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.

---

arguments:

- *igraph*: interaction graph
- *subgraphs*: pairs of lists and subgraph attributes

example:

```
>>> sub1 = (["v1","v2"], {"label":"Genes"})
>>> sub2 = (["v3","v4"], {})
>>> subgraphs = [sub1,sub2]
>>> add_style_subgraphs(igraph, subgraphs)
```

copy_ig(*igraph: DiGraph*) → DiGraph

Creates a copy of *igraph* including all *dot* attributes.

arguments:

- *igraph*: interaction graph

returns:

- *new_igraph*: new interaction graph

example:

```
>>> igraph2 = copy_ig(igraph)
```

create_empty_igraph(*primes: dict*) → DiGraph

creates an empty igraph with default attributes

create_image(*primes: dict*, *fname_image: str*, *styles: List[str] = ['interactionsigns']*, *layout_engine: str = 'fdp'*)

A convenience function for drawing interaction graphs directly from the prime implicants. *styles* must be a sublist of [“interactionsigns”, “inputs”, “outputs”, “constants”, “sccs”, “anonymous”].

arguments:

- *primes*: prime implicants
- *fname_image*: name of image
- *styles* the styles to be applied
- *layout_engine*: one of “dot”, “neato”, “fdp”, “sfdp”, “circo”, “twopi”

example:

```
>>> create_image(primes, "mapk_igraph.pdf", styles=["interactionsigns", "anonymous
↪"])
```

find_minimal_autonomous_nodes(*igraph: DiGraph*, *core: Set[str]*) → List[Set[str]]

Returns the minimal autonomous node sets of *igraph*. See *Klarner2015(b)* Sec. 5.2 for a formal definition and details. Minimal autonomous sets generalize inputs, which are autonomous sets of size 1. If *Superset* is specified then all autonomous sets that are not supersets of it are ignored.

arguments:

- *igraph*: interaction graph
- *core*: all autonomous sets must be supersets of these components

returns:

- *autonomous_nodes* (list of sets): the minimal autonomous node sets of *igraph*

example:

```
>>> find_minimal_autonomous_nodes(igraph)
[set(["raf"]), set(["v1","v8","v9"])]
```

igraph2dot(*igraph: DiGraph*, *fname_dot: Optional[str] = None*) → str

Generates a *dot* file from *igraph* and saves it as *fname_dot* or returns it as a string.

arguments:

- *igraph*: interaction graph
- *fname_dot*: name of *dot* file

returns:

- *dot*: contents of dot file as text

example:

```
>>> igraph2dot(igraph, "irma.dot")
>>> dotfile = igraph2dot(igraph)
```

igraph2image(*igraph: DiGraph*, *fname_image: str*, *layout_engine='fdp'*)

Creates an image file from *igraph* using *Graphviz* and the force directed layout engine *fdp*. To find out which file formats are supported call `$ dot -T?`.

arguments:

- *igraph*: interaction graph
- *fname_image*: name of image
- *layout_engine*: one of “dot”, “neato”, “fdp”, “sfdp”, “circo”, “twopi”

example:

```
>>> igraph2image(igraph, "mapk_igraph.pdf")
>>> igraph2image(igraph, "mapk_igraph.jpg")
>>> igraph2image(igraph, "mapk_igraph.svg")
```

local_igraph_of_state(*primes: dict*, *state: Union[dict, str]*) → DiGraph

Computes the local interaction graph dF/dx of a state x.

arguments:

- *primes*: prime implicants
- *state*: a state

returns:

- *local_igraph*: the local interaction graph

example:

```
>>> primes = get_primes("remy_tumorigenesis")
>>> state = random_state(primes)
>>> local_igraph = local_igraph_of_state(primes, state)
>>> add_style_interactionsigns(local_igraph)
>>> igraph2image(local_igraph, "local_igraph.pdf")
created local_igraph.pdf
```

local_igraph_of_states(*primes: dict*, *states: List[Union[dict, str]]*)

Computes the local interaction graph of a states.

primes2igraph(*primes: dict*) → DiGraph

Creates the interaction graph from the prime implicants of a network. Interaction graphs are implemented as *NetworkX* digraph objects. Edges are given the attribute *sign* whose value is a Python set containing 1 or -1 or both, depending on whether the interaction is activating or inhibiting or both.

arguments:

- *primes*: prime implicants

returns:

- *igraph*: interaction graph

example:

```
>>> bnet = "\n".join(["v1, v1","v2, 1", "v3, v1&!v2 | !v1&v2"])
>>> primes = bnet2primes(bnet)
>>> igraph = primes2igraph(primes)
>>> igraph.nodes()
["v1", "v2", "v3"]
>>> igraph.edges()
[("v1", "v1"), ("v1", "v3"), ("v2", "v3"), ("v3", "v1")]
>>> igraph.adj["v1"]["v3"]["sign"]
set([1, -1])
```

4.8 model_checking

model_checking(*primes: dict*, *update: str*, *initial_states: str*, *specification: str*, *dynamic_reorder: bool = True*, *disable_reachable_states: bool = True*, *cone_of_influence: bool = True*, *enable_counterexample: bool = False*, *enable_accepting_states: bool = False*)

Calls *NuSMV* to check whether the *specification* is true or false in the transition system defined by *primes*, the *initial_states* and *update*. The remaining arguments are *NuSMV* options, see the manual at http://nusmv.fbk.eu for details.

See primes2smv and *Sec. 3.4* for details on model checking with *pyboolnet*.

The accepting states are a dictionary with the following keywords:

- *INIT*: a Boolean expression for the initial states, or *None*, see note below
- *INIT_SIZE*: integer number of initial states, or *None*, see note below
- *ACCEPTING*: a Boolean expression for the accepting states
- *ACCEPTING_SIZE*: integer number of accepting states
- *INITACCEPTING*: a Boolean expression for the intersection of initial and accepting states, or *None*, see note below
- *INITACCEPTING_SIZE*: integer number of states in the intersection of initial and accepting states, or *None*, see note below

---

Note: *disable_reachable_states* is enforced as the accepting states are otherwise over-approximated.

Note: Model checking with accepting states is only possible for CTL model checking.

Note: If the *ctl_spec* is equivalent to either *TRUE* or *FALSE* then NuSMV will not compute the initial states, because it does not have to to find out what the *answer* to the query is. In that case the four values that involve the initial states are set to *None*.

Note: *cone_of_influence* is not available when using counterexamples as it "messes" with the output.

---

arguments:

- *primes*: prime implicants
- *update*: the update strategy, either *"synchronous"*, *"asynchronous"* or *"mixed"*
- *initial_states*: a *NuSMV* expression for the initial states, including the keyword *INIT*
- *specification*: a *NuSMV* formula, including the keyword *LTLSPEC* or *CTLSPEC*
- *dynamic_reorder*: enables dynamic reordering of variables using *-dynamic*
- *disable_reachable_states*: disables the computation of reachable states using *-df*
- *cone_of_influence*: enables cone of influence reduction using *-coi*
- *enable_counterexample*: enables cone of influence reduction using *-coi*

- *enable_accepting_states*: enables cone of influence reduction using *-coi*

returns:

- *answer*: model checking result
- *counterexample*: a countereample, if *enable_counterexample*
- *accepting_states*: the accepting states, if *enable_accepting_states*

example:

```
>>> initial_states = "INIT TRUE"
>>> update = "asynchronous"
>>> specification = "CTLSPEC AF(EG(v1&!v2))"
>>> model_checking(primes, update, initial_states, specification)
False
>>> answer, counterexample = model_checking(primes, update, initial_states,␣
↪specification, enable_counterexample=True)
>>> answer, accepting_states = model_checking(primes, update, initial_states,␣
↪specification, enable_accepting_states=True)
```

model_checking_smv_file(*fname_smv: str*, *dynamic_reorder: bool = True*, *disable_reachable_states: bool = True*, *cone_of_influence: bool = True*, *enable_counterexample: bool = False*, *enable_accepting_states: bool = False*)

Convenience function for model checking of smv files. See *model_checking* for details of parameters and returned objects.

primes2smv(*primes: dict*, *update: str*, *initial_states*, *specification: str*, *fname_smv: Optional[str] = None*) → str

Creates a **NuSMV**_ file from Primes and additional parameters that specify the update strategy, the initial states and the temporal logic specification.

The initial states must be defined in *NuSMV* syntax, i.e., starting with the keyword *INIT*. *NuSMV* uses | for disjunction, *&* for conjunction and *!* for negation. To set the initial states to the whole state space use *"INIT TRUE"*. CTL formulas must start with the keyword *CTLSPEC* and LTL formulas with the keyword *LTLSPEC*.

Note: The *NuSMV* language is case-sensitive and does not allow single character names for variables.

Note: This function detects Boolean variables that represent multi-valued components (van Ham encoding), see StateTransitionGraphs.find_vanham_variables for details. For each multi-valued variable it adds an INIT constraint that restricts the initial states to the admissible states of the van Ham encoding.

In addition to variables that encode the activities of the components, auxillary variables are defined and available for use in CTL or LTL formulas, see *Sec. 3.4* for details:

They are the Boolean *name_IMAGE* which is the value of the update function of the variable *name* in a state, the Boolean *name_STEADY* which is the value for whether the variable *name* is steady in a state, the integer *SUCCESSORS* which is the number of successors excluding itself (i.e., the number of variables that are changing in a state), and the Boolean *STEADYSTATE* which is the value for whether the current state is a steady state (which is equivalent to *SUCCESSORS=0*).

arguments:

- *primes*: prime implicants
- *update*: the update strategy, either *"synchronous"*, *"asynchronous"* or *"mixed"*

- *initial_states*: a *NuSMV* expression for the initial states, including the keyword *INIT*
- *specification*: a *NuSMV* formula, including the keyword *LTLSPEC* or *CTLSPEC*
- *fname_smv*: name for *smv* file or *None*

returns:

- *fname_smv*: file as string or *None* if *FnameSMV is None*

example:

```
>>> ctlspec = "CTLSPEC EF(AG(!ERK) | AG(ERK))"
>>> ltlspec = "LTLSPEC F(G(ERK))"
>>> primes2smv(primes, "asynchronous", "INIT TRUE",  ctlspec, "mapk.smv")
>>> primes2smv(primes, "synchronous",  "INIT ERK=1", ltlspec, "mapk.smv")
>>> lines = primes2smv(primes, "synchronous",  "INIT ERK=1", ltlspec)
```

print_warning_accepting_states_bug(*primes: dict*, *ctl_spec: str*)

This bug occurs when the Specification is equivalent to TRUE or FALSE.

4.9 network_generators

balanced_tree(*height: int*, *branching_factor: int = 2*, *edge_sign: int = 1*, *loop_sign: int = 1*) → dict

Creates a balanced tree network of given *height* and *branching_factor*. All edges are either positive or negative, depending on *edge_sign*. The self-loop of the root component is either positive or negative, depending on *loop_sign*.

arguments:

- *height*: height of tree
- *branching_factor*: branching factor of tree
- *edge_sign*: either *1* or *-1* for positive or negative edges
- *loop_sign*: either *1* or *-1* for positive or negative self-loop of first component

returns:

- *primes*: primes implicants

example:

```
>>> primes = balanced_tree(height=3)
```

cycle_graph(*n: int*, *edge_sign: int = 1*) → dict

Creates a cycle network with *n* components. All edges are either positive or negative, depending on *edge_sign*.

arguments:

- *n*: number of components
- *edge_sign*: either *1* or *-1* for positive or negative edges

returns:

- *primes*: primes implicants

example:

```
>>> primes = cycle_graph(n=3)
```

path_graph(*n: int*, *edge_sign: int = 1*, *loop_sign: int = 1*) → dict

Creates a path graph network with *n* nodes. All edges are either positive or negative, depending on *edge_sign*. The self-loop of the root component is either positive or negative, depending on *loop_sign*.

arguments:

- *n*: number of components
- *edge_sign*: either *1* or *-1* for positive or negative edges
- *loop_sign*: either *1* or *-1* for positive or negative self-loop of root component

returns:

- *primes*: primes implicants

example:

```
>>> primes = path_graph(n=3, edge_sign=-1, loop_sign=1)
```

random_boolean_expression(*names: List[str]*, *k: int*, *seed: int = 0*) → str

Creates a random Boolean expression by sampling *k* variables from *names* and filling a truth table randomly. The expression is guaranteed to be non-constant but it may effectivly be dependent by less than *k* variables.

arguments:

- *names* (List[str]): components to sample from
- *k*: inputs to the expression
- *seed*: the seed of the randomizer

returns:

- *expression*: random expression in bnet format

example:

```
>>> primes = random_boolean_expression(names=list("abcdefgh"), k=4)
```

random_regular_network(*n: int*, *k: int*, *connector: str = 'and'*, *edge_sign: int = 1*, *seed: int = 0*) → dict

Creates a random netowrk with *n* components each with a random truthtable of *k* inputs. The *seed* is used for the randomizer. Use *0* to generate a random seed.

arguments:

- *n*: number of components
- *k*: inputs of each truthtable
- *connector*: either *"and"* or *"or"*
- *edge_sign*: either *1* or *-1* for positive or negative edges
- *seed*: the seed of the randomizer

returns:

- *primes*: primes implicants

example:

```
>>> primes = random_regular_network(n=3, k=2, connector="or")
```

random_truth_table_network(*n: int*, *k: int*, *seed: int = 0*) → dict

Creates a random netowrk with *n* components each with a random truthtable of *k* inputs. The *seed* is used for the randomizer. Use *0* to generate a random seed.

arguments:

- *n*: number of components
- *k*: inputs of each truthtable
- *seed*: the seed of the randomizer

returns:

- *primes*: primes implicants

example:

```
>>> primes = random_truth_table_network(n=3, k=2)
```

4.10 phenotypes

compute_phenotype_diagram(*phenotypes: dict*, *fname_json: Optional[str] = None*, *fname_image: Optional[str] = None*)

Computes the phenotype diagram from the phenotypes object obtained from phenotypes_compute_json. save the diagram as json data with *fname_json*. useful for e.g. manually renaming nodes.

arguments:

- *phenotypes*: result of compute_json(..)
- *fname_json*: save diagram as json
- *fname_image*: generate image for diagram

returns::

- *diagram*: the phenotype diagram

example:

```
>>> phenos = compute_phenotypes(attrobj, markers)
>>> compute_phenotype_diagram(phenos, fname_image="phenos.pdf")
created phenos.pdf
```

compute_phenotypes(*attractors: dict*, *markers: List[str]*, *fname_json: Optional[str] = None*) → dict

Computes the phenotypes object for given *markers*.

structure:

primes: dict update: str markers: tuple phenotypes: (tuple)

name: str pattern: str activated_markers: list of markers inhibited_markers: list of markers state-formula: disjunction over all state props

states: (tuple)

str: state string dict: state dict prop: state proposition is_steady: bool is_cyclic: bool

mintrapspace:
str: subspace string dict: subspace dict prop: subspace proposition

arguments:

- *attractors*: json attractor data, see attractors_compute_json
- *markers*: list of names
- *fname_json*: save phenotypes as json

returns::

- *phenotypes*: the phenotypes data

example:

```
>>> attractors = compute_attractors(primes, update, "attractors.json")
>>> markers = ["raf", "mek"]
>>> compute_phenotypes(attractors, markers, "phenotypes.json")
```

create_phenotypes_piechart(*diagram: DiGraph*, *fname_image: str*, *title: Optional[str] = None*, *color_map: Optional[dict] = None*)

creates the abstract commitment pie.

arguments:

- *diagram*: a phenotype diagram
- *fname_image*: name of the output image
- *title*: optional title of plot
- *color_map*: assignment of diagram nodes to colors for custom colors

example:

```
>>> attractors = compute_attractors(primes, update)
>>> phenos = compute_phenotypes(attractors, markers)
>>> diagram = compute_phenotype_diagram(phenos)
>>> create_phenotypes_piechart(diagram, "piechart.pdf")
```

open_phenotype_diagram(*fname_json: str*) → DiGraph

Opens a phenotype diagram.

arguments:

- *fname_json*: json file name

returns:

- *phenotype_diagram*: the phenotype diagram

example:

```
>>> open_phenotype_diagram("diagram.json")
```

phenotype_diagram2image(*diagram: DiGraph*, *fname_image: Optional[str] = None*)

Creates an image of the abstract commitment diagram.

arguments:

- *diagram*: a phenotype diagram

- *fname_image*: name of the diagram image

returns::

- *styled_diagram*: the styled abstract commitment diagram

example:

```
>>> phenotype_diagram2image(primes, diagram, "diagram.pdf")
```

save_phenotype_diagram(*diagram: DiGraph*, *fname_json: str*)

Save the diagram as a json file.

arguments:

- *diagram*: prime implicants
- *fname_json*: json file name

example:

```
>>> save_phenotype_diagram(diagram, "diagram.json")
```

4.11 prime_implicants

active_primes(*primes: dict*, *subspace: Dict[str, int]*) → Dict[str, List[List[dict]]]

returns all primes that are active in, i.e., consistent with *subspace*.

copy_primes(*primes: dict*) → dict

Creates a copy of *primes*.

arguments:

- *primes*: prime implicants

returns:

- *new_primes*: a copy of *primes*

example:

```
>>> new_primes = copy_primes(primes)
```

create_blinkers(*primes: dict*, *names: Iterable[str]*, *copy: bool = False*) → Optional[dict]

Creates a blinker for every member of *names*. Variables that alrerady exist in *primes* are overwritten. A blinker is a variable with in-degree one and negative auto-regulation. Blinkers can therefore change their activity in every state of the transition system.

Note: Suppose that a given network has a lot of inputs, e.g.:

```
>>> len(find_inputs(primes))
20
```

Since input combinations define trap spaces, see e.g. *Klarner2015(b)* Sec. 5.1, all of which contain at least one minimal trap space, it follows that the network has at least 2^20 minimal trap spaces - too many to enumerate. To find out how non-input variables stabilize in minimal trap spaces turn the inputs into blinkers:

```
>>> inputs = find_inputs(primes)
>>> create_blinkers(primes, inputs)
>>> tspaces = trap_spaces(primes, "min")
```

---

arguments:

- *primes*: prime implicants
- *names*: variables to become blinkers
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> names = ["p21", "p53"]
>>> create_blinkers(primes, names)
```

create_constants(*primes: dict*, *constants: Dict[str, int]*, *copy: bool = False*) → Optional[dict]

Creates a constant in *primes* for every name, value pair in *constants*. Names that already exist in *primes* are overwritten.

arguments:

- *primes*: prime implicants
- *constants*: names and values
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> create_constants(primes, {"p53":1, "p21":0})
```

create_disjoint_union(*primes1: dict*, *primes2: dict*) → dict

Creates a new primes dictionary that is the disjoint union of the networks represented by *primes1* and *primes2*. Here, "disjoint" means that the names of *primes1* and *primes2* must not intersect.

arguments:

- *primes1*: prime implicants
- *primes2*: prime implicants

returns:

- *new_primes*: the disjoint union of *primes1* and *primes2*

example:

```
>>> primes1 = bnet2primes("A, B \n B, A")
>>> primes2 = bnet2primes("C, D \n D, E")
>>> primes = create_disjoint_union(primes1, primes2)
```

(continues on next page)

(continued from previous page)

```
>>> primes2bnet(primes)
A, B
B, A
C, D
D, E
```

create_inputs(*primes: dict*, *names: Iterable[str]*, *copy: bool = False*) → Optional[dict]

Creates an input for every member of *names*. Variables that already exist in *primes* are overwritten.

Note: Suppose that a given network has constants, e.g.:

```
>>> len(find_constants(primes))>0
True
```

Too analyze how the network behaves under all possible values for these constants, turn them into inputs:

```
>>> constants = find_constants(primes)
>>> create_inputs(primes, constants)
```

---

arguments:

- *primes*: prime implicants
- *names*: variables to become constants
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> names = ["p21", "p53"]
>>> create_inputs(primes, names)
```

create_variables(*primes: dict*, *update_functions: Dict[str, Union[callable, str]]*, *copy: bool = False*) → Optional[dict]

Creates the variables defined in *update_functions* and add them to *primes*. Variables that already exist in *primes* are overwritten. Raises an exception if the resulting primes contain undefined variables. The *update_functions* are given as a dictionary of names and functions that are either a bnet string or a Python callable.

arguments:

- *primes*: prime implicants
- *update_functions*: a dictionary of names and update functions
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> primes = bnet2primes("A, A")
>>> create_variables(primes, {"B": "A"})
>>> create_variables(primes, {"C": lambda A, B: A and not B})
>>> primes2bnet(primes)
A, A
B, A
C, A&!B
```

find_constants(*primes: dict*) → Dict[str, int]

Finds all constants in the network defined by *primes*.

arguments:

- *primes*: prime implicants

returns:

- *constants*: the names and activities of constants

example:

```
>>> find_constants(primes)
{"CGC":1,"IFNAR1":1,"IFNAR2":0,"IL4RA":1}
```

find_inputs(*primes: dict*) → List[str]

Finds all inputs in the network defined by *primes*.

arguments:

- *primes*: prime implicants

returns:

- *names*: the names of the inputs

example:

```
>>> find_inputs(primes)
["DNA_damage","EGFR","FGFR3"]
```

find_outputs(*primes: dict*) → List[str]

Finds all outputs in the network defined by *primes*.

arguments:

- *primes*: prime implicants

returns:

- *names*: the names of the outputs

example:

```
>>> find_inputs(primes)
["Proliferation","Apoptosis","GrowthArrest"]
```

find_predecessors(*primes: dict*, *targets: Iterable[str]*) → List[str]

Finds all predecessors of *targets* in *primes*.

arguments:

- *primes*: prime implicants

- *targets*: compute predecessors of these nodes

returns:

- *predecessors*: the predecessors

example:

```
>>> find_predecessors(primes)
["ERK", "MEK", "RAF"]
```

find_successors(*primes: dict*, *sources: Iterable[str]*) → List[str]

Finds all successors of *sources* in *primes*.

arguments:

- *primes*: prime implicants
- *sources*: compute successors of these nodes

returns:

- *successors*: the successors

example:

```
>>> find_successors(primes)
["ERK", "MEK", "RAF"]
```

get_constant_primes(*value: int*) → List[List[dict]]

Gets the prime implicants for a constant of give *value*.

arguments:

- *value*: either 0 or 1

returns:

- *implicants*: the implicants

example:

```
>>> primes[name] = get_constant_primes(value=1)
```

is_constant(*primes: dict*, *name: str*) → bool

Decides whether *name* is a constant in *primes*.

arguments:

- *primes*: prime implicants
- *name*: the component

returns:

- *constant*: whether *name* is constant

example:

```
>>> is_constant(primes, "MEK")
False
```

list_input_combinations(*primes: dict*, *format: str = 'dict'*) → Union[List[str], List[dict]]

A generator for all possible input combinations of *primes*. Returns the empty dictionary if there are no inputs.

arguments:

- *primes*: prime implicants
- *format*: format of returned subspaces, "str" or "dict"

returns:

- *subspaces*: input combination in desired format

example:

```
>>> for x in list_input_combinations(primes, "str"): print(x)
0--0--
0--1--
1--0--
1--1--
```

percolate(*primes: dict*, *add_constants: Optional[Dict[str, int]] = None*, *remove_constants: bool = False*, *max_iterations: Optional[int] = None*, *copy: bool = False*) → Optional[dict]

Detects constants in *primes* and percolates their values.

arguments:

- *primes*: prime implicants
- *max_iterations*: max number of components to fix during percolation
- *add_constants*: create new constants before percolation
- *remove_constants*: remove constants after percolation
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> new_primes = percolate(primes, copy=True)
```

primes_are_equal(*primes1: dict*, *primes2: dict*) → bool

Tests whether *primes1* and *primes2* are equal. The dictionary comparison *primes1* == *primes2* does in general not work because the clauses of each may not be in the same order.

arguments:

- *primes1*, *primes2*: prime implicants

returns:

- *answer*: whether *primes1* == *primes2*

example:

```
>>> primes_are_equal(primes1, primes2)
True
```

remove_all_constants(*primes: dict*, *copy: bool = False*) → Optional[dict]

Removes all constants from *primes*.

arguments:

- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> remove_all_constants(primes)
```

remove_all_variables_except(*primes: dict*, *names: Iterable[str]*, *copy: bool = False*) → Optional[dict]

Removes all variables except those in *names* from *primes*. Members of *names* that are not in *primes* are ignored. Note that *names* must be closed under the predecessors relation, i.e., it must be a set of variables that contains all its predecessors.

arguments:

- *primes*: prime implicants
- *names*: the names of variables to keep
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> names = ["PKC", "GADD45", "ELK1", "FOS"]
>>> remove_all_variables_except(primes, names)
```

remove_variables(*primes: dict*, *names: Iterable[str]*, *copy: bool = False*) → Optional[dict]

Removes all variables contained in *names* from *primes*. Members of *names* that are not in *primes* are ignored. Note that *names* must be closed under the successors relation, i.e., it must be a set of variables that contains all its successors.

arguments:

- *primes*: prime implicants
- *names*: the names of variables to remove
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> names = ["PKC", "GADD45", "ELK1", "FOS"]
>>> remove_variables(primes, names)
```

rename_variable(*primes: dict*, *old_name: str*, *new_name: str*, *copy: bool = False*) → Optional[dict]

Renames a single component, i.e., replace every occurence of *old_name* with *new_name*. Throws an exception if *new_name* is already contained in *primes*.

arguments:

- *primes*: prime implicants
- *old_name*: the old name of the component
- *new_name*: the new name of the component
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> names = ["PKC", "GADD45", "ELK1", "FOS"]
>>> remove_all_variables_except(primes, names)
```

update_primes(*primes: dict*, *name: str*, *constants: dict*, *copy: bool = False*) → Optional[dict]

Updates the primes of the component *name* given the *constants*.

arguments:

- *primes*: prime implicants
- *name*: name of the component to update
- *constants*: the assumed constants
- *copy*: change *primes* in place or copy and return

returns:

- *new_primes* if *copy* == *True*

example:

```
>>> primes["ERK"] = update_primes(primes, name="ERK", subspace={"MEK": 1, "RAF": 0})
```

4.12 state_space

bounding_box(*primes: dict*, *subspaces: List[Union[dict, str]]*) → Dict[str, int]

Returns the smallest subspace that contains the union of *subspaces*.

arguments:

- *primes*: prime implicants

returns:

- *box*: the smallest subspace that contains the union of *subspaces*

example:

```
>>> subspaces = [{"v1": 0, "v2": 0}, {"v1": 1, "v2": 0})]
>>> bounding_box(primes, subspaces)
{"v2": 0}
```

enumerate_states(*primes: dict*, *proposition: str*)

Generates all states that are referenced by *proposition* in the context of the variables given by *primes*. The syntax of *proposition* should be as in bnet files and TRUE and FALSE in will be treated as 1 and 0.

Note: This function uses bnet2primes and list_states_in_subspace to enumerate the states referenced by an expression. The efficiency of this approach can decreases a lot starting from around 15 variables that appear in *proposition*.

---

arguments:

- *primes*: prime implicants
- *proposition*: a propositional formula

returns:

- *states*: the referenced states in str format

example:

```
>>> prop = "!Erk | (Raf & Mek)"
>>> enumerate_states(primes,prop)[0]
"010"
```

find_vanham_variables(*primes: dict*) → Dict[int, List[str]]

Detects variables that represent multi-valued variables using the Van Ham encoding, see Didier2011 for more details. E.g. three-valued variables x are encoded via two Boolean variables x_medium and x_high. This function is used for example by ModelChecking.primes2smv to add INIT constraints to the smv file that forbid all states that are not admissible, e.g. in which "!x_medium & x_high" is true.

arguments:

- *primes*: prime implicants

returns:

- *names*: activity levels and names

example:

```
>>> find_vanham_variables(primes)
{2: ['E2F1', 'ATM'],
 3: ['ERK'],
 4: [],
 5: []}
```

hamming_distance(*this: dict*, *other: dict*) → int

Returns the Hamming distance between to subspaces. Variables that are free in either subspace are ignored.

arguments:

- *this, other*: subspaces in dictionary representation

returns:

- *distance*: the distance between *Subspace1* and *Subspace2*

example:

```
>>> hamming_distance({"v1":0,"v2":0}, {"v1":1,"v2":1})
2
>>> hamming_distance({"v1":1}, {"v2":0})
0
```

is_subspace(*primes: dict, this: ~typing.Union[dict, str], other: [<class 'dict'>, <class 'str'>]*) → bool

Checks whether *this* is a subspace of *other*.

arguments:

- *primes*: prime implicants
- *this*: a subspace
- *other*: a subspace

returns:

- *answer*: whether *this* is subspace of *other*

example:

```
>>> is_subspace(primes, this=sub1, other=sub2)
True
```

list_states_in_subspace(*primes: dict*, *subspace*)

Generates all states contained in *subspace*.

arguments:

- *primes*: prime implicants or a list of names
- *subspace*: a subspace

returns:

- *states*: the states contained in *subspace*

example:

```
>>> subspace = "1-1"
>>> list_states_in_subspace(primes,subspace)
['101','111']
```

random_state(*primes: dict*, *subspace: Union[dict, str] = {}*) → dict

Generates a random state of the transition system defined by *primes*. If *subspace* is given then the state will be drawn from that subspace.

arguments:

- *primes*: prime implicants
- *subspace*: a subspace

returns:

- *state*: random state inside *subspace*

example:

```
>>> random_state(primes)
{'v1':1, 'v2':1, 'v3':1}
>>> random_state(primes, {"v1":0})
{'v1':0, 'v2':1, 'v3':0}
>>> random_state(primes, "0--")
{'v1':0, 'v2':0, 'v3':1}
```

size_state_space(*primes: dict*, *van_ham: bool = True*, *fixed_inputs: bool = False*)

This function computes the number of states in states space of *primes*. Detects if there are variables that encode multi-valued variables via the Van Ham encoding. Variables that have the same name module the Van Ham extension (see example below) are identified. E.g. the state space of x_medium, x_high is 3 instead of 4 since "!x_medium & x_high" is not an admissible state, see Didier2011 for more details.

arguments:

- *primes*: prime implicants
- *van_ham*: exclude states that are not admissible in Van Ham encoding
- *fixed_inputs*: return number of states for single input combination

returns:

- *size*: number of states

example:

```
>>> StateTransitionGraphs.VAN_HAM_EXTENSIONS
["_medium", "_high", "_level1", "_level2", "_level3", "_level4", "_level5"]
>>> size_state_space(primes, van_ham=False)
256
>>> size_state_space(primes)
192
>>> size_state_space(primes, fixed_inputs=True)
96
```

state2dict(*primes: dict*, *state*) → dict

Converts the string representation of a state into the dictionary representation of a state. If *state* is already of type *dict* it is simply returned.

arguments

- *primes*: prime implicants or a list of names
- *state*: string representation of state

returns

- *state*: dictionary representation of state

example:

```
>>> state = "101"
>>> state2dict(primes, state)
{'v2':0, 'v1':1, 'v3':1}
```

state2str(*state: Union[dict, str]*) → str

Converts the dictionary representation of a state into the string representation of a state. If *state* is already of type string it is simply returned.

arguments

- *state*: dictionary representation of state

returns

- *state*: string representation of state

example:

```
>>> state = {"v2":0, "v1":1, "v3":1}
>>> state2str(primes, state)
'101'
```

state_is_in_subspace(*primes: dict*, *state: Union[str, dict]*, *subspace: Union[str, dict]*) → bool

Checks whether *state* is a state in *subspace*.

arguments:

- *primes*: prime implicants
- *state*: a state
- *subspace*: a subspace

returns:

- *answer*: whether *state* is a state in *subspace*

example:

```
>>> state_is_in_subspace(primes, state, subspace)
False
```

states2dict(*primes: dict*, *states: List[str]*) → List[dict]

Converts the string representation of a list of states into the dictionary representations. If *state* is already of type *dict* it is simply returned.

arguments

- *primes*: prime implicants or a list of names
- *states*: string representation of states

returns

- *states*: dictionary representation of states

example:

```
>>> states = ["101", "100"]
>>> states2dict(primes, state)
[{'v2':0, 'v1':1, 'v3':1},{'v2':0, 'v1':1, 'v3':0}]
```

states2str(*primes: dict*, *states: List[dict]*) → List[str]

Converts the string representation of a list of states into the dictionary representations. If *state* is already of type *dict* it is simply returned.

arguments

- *primes*: prime implicants or a list of names
- *states*: dictionary representation of states

returns

- *states*: string representation of states

example:

```
>>> states = [{'v2':0, 'v1':1, 'v3':1},{'v2':0, 'v1':1, 'v3':0}]
>>> states2str(primes, state)
["101", "100"]
```

subspace2dict(*primes: dict*, *subspace*)

Converts the string representation of a subspace into the dictionary representation of a subspace. Use "-" to indicate free variables. If *subspace* is already of type *dict* it is simply returned.

arguments

- *primes*: prime implicants or a list of names
- *subspace*: a subspace

returns

- *subspace*: the dictionary representation of subspace

example:

```
>>> sub = "-01"
>>> subspace2dict(primes, sub)
{'v2':0, 'v3':1}
```

subspace2str(*primes: dict*, *subspace*)

Converts the dictionary representation of a subspace into the string representation of a subspace. Uses "-" to indicate free variables. If *subspace* is already of type *str* it is simply returned.

arguments

- *primes*: prime implicants or a list of names
- *subspace*: a subspace

returns

- *subspace*: the string representation of *subspace*

example:

```
>>> sub = {"v2":0, "v3":1}
>>> subspace2str(primes, sub)
'-01'
```

4.13 state_transition_graphs

add_style_anonymous(*stg: DiGraph*)

Removes the labels and draws each state as a filled circle.

arguments:

- *stg*: state transition graph

example:

```
>>> add_style_anonymous(stg)
```

add_style_default(*primes: dict*, *stg: DiGraph*)

A convenience function that adds styles for tendencies, SCCs and minimal trap spaces.

arguments:

- *primes*: primes implicants
- *stg*: state transition graph

example:

```
>>> add_style_default(stg)
```

add_style_mintrapspaces(*primes: dict*, *stg: DiGraph*, *max_output: int = 100*)

A convenience function that combines add_style_subspaces and trap_spaces. It adds a *dot* subgraphs for every minimal trap space to *stg* - subgraphs that already exist are overwritten.

arguments:

- *primes*: prime implicants
- *stg*: state transition graph
- *MaxOutput*: maximal number of minimal trap spaces, see trap_spaces

example:

```
>>> add_style_mintrapspaces(primes, stg)
```

add_style_path(*stg: DiGraph*, *path: Union[List[str], List[dict]]*, *color: str*, *pen_width: int = 3*)

Sets the color of all nodes and edges involved in the given *path* to *color*.

arguments:

- *stg*: state transition graph
- *path*: state dictionaries or state strings
- *color*: color of the path
- *pen_width*: width of nodes and edges involved in *path* in pt

example:

```
>>> path = ["001", "011", "101"]
>>> add_style_path(stg, path, "red")
```

add_style_sccs(*stg: DiGraph*)

Adds a subgraph for every non-trivial strongly connected component (SCC) to the *dot* representation of *stg*. Nodes that belong to the same *dot* subgraph are contained in a rectangle and treated separately during layout computations. Each subgraph is filled by a shade of gray that gets darker with an increasing number of SCCs that are above it in the condensation graph. Shadings repeat after a depth of 9.

arguments:

- *stg*: state transition graph

example:

```
>>> add_style_sccs(stg)
```

add_style_subgraphs(*stg: DiGraph*, *subgraphs*)

Adds the subgraphs given in *subgraphs* to *stg* - or overwrites them if they already exist. Nodes that belong to the same *dot* subgraph are contained in a rectangle and treated separately during layout computations. *subgraphs* must consist of tuples of the form *NodeList*, *Attributs* where *NodeList* is a list of graph nodes and *Attributes* is a dictionary of subgraph attributes in *dot* format.

Note: *subgraphs* must satisfy the following 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.

---

arguments:

- *stg*: state transition graph
- *subgraphs*: pairs of lists and subgraph attributes

example:

```
>>> sub1 = (["001","010"], {"label":"critical states"})
>>> sub2 = (["111","011"], {})
>>> subgraphs = [sub1,sub2]
>>> add_style_subgraphs(stg, subgraphs)
```

add_style_subspaces(*primes: dict*, *stg: DiGraph*, *subspaces*)

Adds a *dot* subgraph for every subspace in *subspace* to *stg* - or overwrites them if they already exist. Nodes that belong to the same *dot* subgraph are contained in a rectangle and treated separately during layout computations. To add custom labels or fillcolors to a subgraph supply a tuple consisting of the subspace and a dictionary of subgraph attributes.

Note: *subgraphs* must satisfy the following 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.

---

arguments:

- *primes*: prime implicants
- *stg*: state transition graph
- *subspaces*: list of subspaces in string or dict representation

example:

```
>>> subspaces = [{"v1":0},{"v1":0,"v3":1},{"v1":1,"v2":1}]
>>> add_style_subspaces(primes, stg, subspaces)
>>> subspaces = ["0--","0-1","11-"]
>>> add_style_subspaces(primes, stg, subspaces)
```

add_style_tendencies(*stg: DiGraph*)

Sets or overwrites the edge colors to reflect whether a transition increases values (*black*), decreases values (*red*), or both (*blue*) which is only possible for non-asynchronous transitions.

arguments:

- *stg*: state transition graph

example:

```
>>> add_style_tendencies(stg)
```

best_first_reachability(*primes: dict*, *initial_space: Union[str, dict]*, *goal_space: Union[str, dict]*, *memory: int = 1000*)

Performs a best-first search in the asynchronous transition system defined by *primes* to answer the question whether there is a path from a random state in *InitalSpace* to a state in *GoalSpace*. *Memory* specifies the maximal number of states that can be kept in memory as "already explored" before the algorithm terminates. The search is guided by minimizing the Hamming distance between the current state of an incomplete path and the *GoalSpace* where variables that are free in *GoalSpace* are ignored.

Note: If the number of variables is less than 40 you should use LTL or CTL model checking to answer questions of reachability. best_first_reachability is meant for systems with more than 40 variables. If best_first_reachability returns *None* then that does not prove that there is no path between *InitialSpace* and *GoalSpace*.

---

arguments:

- *primes*: prime implicants
- *initial_space*: initial subspace
- *goal_space*: goal subspace
- *memory*: maximal number of states memorized before search is stopped

returns:

- *path*: a path from *InitalSpace* to *GoalSpace* if it was found, or *None* otherwise.

example:

```
>>> initspace = "1--0"
>>> goalspace = "0--1"
>>> path = best_first_reachability(primes, initialstate, goalspace)
>>> if path: print(len(path))
4
```

condensationgraph2dot(*cgraph: DiGraph*, *fname_dot: Optional[str] = None*)

Creates a *dot* file from a condensation graph.

arguments:

- *cgraph*: state transition graph
- *fname_dot*: name of *dot* file or *None*

returns:

- *text_dot*: file as string if not *FnameDOT is None*, otherwise it returns *None*

example:

```
>>> condensationgraph2dot(cgraph, "mapk_cgraph.dot")
```

condensationgraph2image(*cgraph: DiGraph*, *fname_image: str*, *layout_engine: str = 'dot'*)

Creates an image file from the condensation graph.

arguments:

- *cgraph*: condensation graph
- *fname_image*: name of output file
- *layout_engine*: one of "dot", "neato", "fdp", "sfdp", "circo", "twopi"

example:

```
>>> condensationgraph2image(cgraph, "dot", "mapk_cgraph.pdf")
```

copy_stg(*stg: DiGraph*) → DiGraph

Creates a copy of *stg* including all *dot* attributes.

arguments:

- *stg*: state transition graph

returns:

- *stg_copy*: new state transition graph

example:

```
>>> stg_copy = copy_stg(stg)
```

create_stg_image(*primes: dict*, *update: str*, *fname_image: str*, *initial_states=<function <lambda>>*, *styles: ~typing.List[str] = []*, *layout_engine: str = 'dot'*)

A convenience function for drawing state transition graphs directly from the prime implicants. *styles* must be a sublist of ["tendencies", "sccs", "mintrapspaces", "anonymous"].

arguments:

- *primes*: prime implicants
- *fname_image*: name of image
- *initial_states*: a function, a subspace, a state or a list of states
- *styles*: the styles to be applied
- *layout_engine*: one of "dot", "neato", "fdp", "sfdp", "circo", "twopi"

example:

```
>>> create_stg_image(primes, "asynchronous", "mapk_stg.pdf", styles=[
↪"interactionsigns", "anonymous"])
```

energy(*primes: dict*, *state*) → int

This integer valued energy function E for Boolean networks is decreasing with transitions. That is, E(x) >= E(y) holds for any transition x -> y. It is based on the number of free variables in the smallest trapspace that contains *state*. The energy is therefore n >= E(x) >= 0 and E(x)=0 for steady states holds.

arguments:

- *primes*: prime implicants
- *state*: a state

returns:

- *energy*: number of free variables in smallest trapspace containing *state*

example:

```
>>> primes = Repository.get_primes("raf")
>>> StateTransitionGraphs.energy(primes, "000")
1
>>> StateTransitionGraphs.energy(primes, "010")
3
>>> StateTransitionGraphs.energy(primes, "001")
0
```

htg2dot(*htg: DiGraph*, *fname_dot: Optional[str] = None*) → str

Creates a *dot* file of the *HTG*.

arguments:

- *HTG*: HTG
- *fname_dot*: name of *dot* file or *None*

returns:

- *text_dot*: text content of dot file

example:

```
>>> htg2dot(cgraph, "mapk_htg.dot")
```

htg2image(*htg: DiGraph*, *fname_image: str*, *layout_engine: str = 'dot'*)

Creates an image file from the *HTG*.

arguments:

- *HTG*: HTG
- *fname_image*: name of output file
- *layout_engine*: one of "dot", "neato", "fdp", "sfdp", "circo", "twopi"

example:

```
>>> htg2image(cgraph, "mapk_htg.pdf")
```

list_reachable_states(*primes: dict*, *update: str*, *initial_states: List[str]*, *memory: int*)

Performs a depth-first search in the transition system defined by *primes* and *update* to list all states that are reachable from the *inital states*. *Memory* specifies the maximum number of states that can be kept in memory as "already explored" before the algorithm terminates.

arguments:

- *primes*: prime implicants
- *update*: update strategy (either asynchronous or snchronous)
- *initial_states*: a list of initial states
- *memory*: maximal number of states memorized before search is stopped

returns:

- *reachable_states*: a list of all states explored

example:

```
>>> initial_states = ["1000", "1001"]
>>> update = "asynchronous"
>>> memory = 1000
>>> states = list_reachable_states(primes, update, initial_states, memory)
>>> print(len(states))
287
```

primes2stg(*primes: dict*, *update: str*, *initial_states=<function <lambda>>*) → DiGraph

Creates the state transition graph (STG) of a network defined by *primes* and *update*. The *initial_states* are either a list of states (in *dict* or *str* representation), a function that flags states that belong to the initial states, or a subspace (in *dict* or *str* representation). If *initial_states* is a function then it must take a single parameter *state* in dict representation and return a Boolean value that indicates whether it belongs to the initial states or not.

The STG is constructed by a depth first search (DFS) starting from the given initial states. The default for *initial_states* is `lambda x: True`, i.e., every state is initial. For a single initial state, say *"100"* use *initial_states="100"*, for a set of initial states use *initial_states=["100", "101"]* and for a initial subspace use *initial_states="1–"* or the *dict* representation of subspaces.

arguments:

- *primes*: prime implicants
- *update*: either *"asynchronous"* or *"synchronous"*
- *initial_states*: a function, a subspace, a state or a list of states

returns:

- *stg*: state transition graph

example:

```
>>> primes = FEX.read_primes("mapk.primes")
>>> update = "asynchronous"
>>> init = lambda x: x["ERK"]+x["RAF"]+x["RAS"]>=2
>>> stg = primes2stg(primes, update, init)

>>> stg.order()
32

>>> stg.edges()[0]
('01000','11000')

>>> init = ["00100", "00001"]
>>> stg = primes2stg(primes, update, init)

>>> init = {"ERK":0, "RAF":0, "RAS":0, "MEK":0, "p38":1}
>>> stg = primes2stg(primes, update, init)
```

random_successor_asynchronous(*primes: dict*, *state: Union[dict, str]*) → dict

Returns a random asynchronous successor of *state* in the transition system defined by *primes*.

arguments:

- *primes*: prime implicants
- *state*: a state

returns:

- *successor*: a random asynchronous successor of *state*

random_successor_mixed(*primes: dict*, *state: Union[dict, str]*) → dict

Returns a random successor of *state* in the mixed transition system defined by *primes*. The mixed update contains the synchronous and asynchronous STGs but it also allows transitions in which an arbitrary number of unstable components (but at least one) are updated.

Note: The reason why this function returns a random mixed transition rather than all mixed successors is that there are up to 2^n mixed successors for a state (n is the number of variables).

---

arguments:

- *primes*: prime implicants
- *state*: a state

returns:

- *successor*: a random successor of *state* using the mixed update

example:

```
>>> state = "100"
>>> random_successor_mixed(primes, state)
{'v1':1, 'v2':1, 'v3':1}
```

random_walk(*primes: dict*, *update: str*, *initial_state: Union[dict, str]*, *length: int*)

Returns a random walk of *length* many states in the transition system defined by *primes* and *update* starting from a state defined by *initial_state*. If *initial_state* is a subspace then random_state will be used to draw a random state from inside it.

arguments:

- *primes*: prime implicants
- *update*: the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *initial_state*: an initial state or subspace
- *length*: length of the random walk

returns:

- *path*: sequence of states

example:

```
>>> path = random_walk(primes, "asynchronous", "11---0", 4)
```

sccgraph2dot(*scc_graph: DiGraph*, *fname_dot: Optional[str] = None*)

Creates a *dot* file from a SCC graph.

arguments:

- *scc_graph*: state transition graph
- *fname_dot*: name of *dot* file or *None*

returns:

- *text_dot*: file as string if not *FnameDOT is None*, otherwise it returns *None*

example:

```
>>> sccgraph2dot(sccg, "sccgraph.dot")
```

sccgraph2image(*scc_graph: DiGraph*, *fname_image: str*, *layout_engine: str = 'dot'*)

Creates an image file from a SCC graph.

arguments:

- *scc_graph*: SCC graph
- *fname_image*: name of output file
- *layout_engine*: one of "dot", "neato", "fdp", "sfdp", "circo", "twopi"

example:

```
>>> sccgraph2image(sccgraph, "mapk_sccgraph.pdf")
```

stg2condensationgraph(*stg: DiGraph*) → DiGraph

Converts the *stg* into the condensation graph, for a definition see *Klarner2015(b)*.

arguments:

- *stg*: state transition graph

returns:

- *cgraph*: the condensation graph of *stg*

example:

```
>>> cgraph = stg2condensationgraph(stg)
```

stg2dot(*stg: DiGraph*, *fname_dot: Optional[str] = None*) → str

Creates a *dot* file from a state transition graph. Graph, node and edge attributes are passed to the *dot* file by adding the respective key and value pairs to the graph, node or edge data. Node and edge defaults are set by the specials graph keys *"node"* and *"edge"* and must have attribute dictionaries as values. For a list of attributes see http://www.graphviz.org/doc/info/attrs.html.

arguments:

- *stg*: state transition graph
- *fname_dot*: name of *dot* file or *None*

returns:

- *text_dot*: content of dot file

example:

```
>>> stg = primes2stg(primes, update, init)
>>> stg.graph["label"] = "IRMA Network - State Transition Graph"
>>> stg.graph["node"] = {"style":"filled", "color":"red"}
>>> stg.graph["edge"] = {"arrowsize": 2.0}
>>> stg.nodes["001000"]["fontsize"] = 20
>>> stg.adj["001110"]["001010"]["style"] = "dotted"
>>> stg2image(stg, "irma_stg.pdf")
```

stg2htg(*stg: DiGraph*)

Computes the HTG of the *stg*. For a definition see *Berenguier2013*.

arguments:

- *stg*: state transition graph

returns:

- *htg*: the HTG of *stg*

example:

```
>>> htg = stg2htg(stg)
```

stg2image(*stg: DiGraph*, *fname_image: str*, *layout_engine: str = 'fdp'*)

Creates an image file from a state transition graph using *Graphviz* and the *layout_engine*. Use `dot -T?` to find out which output formats are supported on your installation.

arguments:

- *stg*: state transition graph
- *fname_image*: name of output file
- *layout_engine*: one of "dot", "neato", "fdp", "sfdp", "circo", "twopi"

example:

```
>>> stg2image(stg, "mapk_stg.pdf")
>>> stg2image(stg, "mapk_stg.jpg", "neato")
>>> stg2image(stg, "mapk_stg.svg", "dot")
```

stg2sccgraph(*stg: DiGraph*) → DiGraph

Computes the SCC graph of the *stg*. For a definition see Sec. 3.1 of *Tournier2009*.

arguments:

- *stg*: state transition graph

returns:

- *scc_graph*: the SCC graph of *stg*

example:

```
>>> sccgraph = stg2sccgraph(stg)
```

successor_synchronous(*primes: dict*, *state: Union[dict, str]*) → dict

Returns the successor of *state* in the fully synchronous transition system defined by *primes*. See *Klarner2015(b)* Sec. 2.2 for a formal definition.

arguments:

- *primes*: prime implicants
- *state*: a state

returns:

- *successor*: the synchronous successor of *state*

example::

```
>>> primes = {
'v1': [[{'v2': 0}], [{'v2': 1}]],
'v2': [[{'v3': 0}, {'v1': 0}], [{'v1': 1, 'v3': 1}]],
'v3': [[{'v1': 1, 'v2': 0}], [{'v2': 1}, {'v1': 0}]]
}
>>> state = "100"
>>> successor_synchronous(primes, state)
{'v1': 0, 'v2': 0, 'v3': 0}
```

successors_asynchronous(*primes: dict*, *state: Union[dict, str]*) → List[dict]

Returns the successors of *state* in the fully asynchronous transition system defined by *primes*. See *Klarner2015(b)* Sec. 2.2 for a formal definition.

arguments:

- *primes*: prime implicants
- *state*: a state

returns:

- *successors*: the asynchronous successors of *state*

example::

```
>>> primes = {
'v1': [[{'v2': 0}], [{'v2': 1}]],
'v2': [[{'v3': 0}, {'v1': 0}], [{'v1': 1, 'v3': 1}]],
'v3': [[{'v1': 1, 'v2': 0}], [{'v2': 1}, {'v1': 0}]]}
>>> state = "101"
>>> successors_asynchronous(primes, state)
[{'v1': 0, 'v2': 0, 'v3': 1}, {'v1': 1, 'v2': 1, 'v3': 1}, {'v1': 1, 'v2': 0,
↪'v3': 0}]
```

successors_mixed(*primes: dict*, *state: Union[dict, str]*) → List[dict]

Returns the successors of *state* in the mixed transition system defined by *primes*. The mixed update contains the synchronous and asynchronous STGs but it also allows transitions in which an arbitrary number of unstable components (but at least one) are updated.

Note: There are up to 2^n mixed successors for a state (n is the number of variables).

---

arguments:

- *primes*: prime implicants
- *state*: a state

returns:

- *successors*: the mixed successors of *state*

example::

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

(continues on next page)

(continued from previous page)

```
'v3': [[{'v1': 1, 'v2': 0}], [{'v2': 1}, {'v1': 0}]]
}
>>> state = "010"
>>> successors_mixed(primes, state)
[{'v1': 1, 'v2': 1, 'v3': 0},
 {'v1': 0, 'v2': 0, 'v3': 0},
 {'v1': 0, 'v2': 1, 'v3': 1},
 {'v1': 1, 'v2': 0, 'v3': 0},
 {'v1': 1, 'v2': 1, 'v3': 1},
 {'v1': 0, 'v2': 0, 'v3': 1},
 {'v1': 1, 'v2': 0, 'v3': 1}]
```

4.14 temporal_logic

all_globally_exists_finally_one_of_sub_spaces(*primes: dict*, *sub_spaces: List[dict]*) → str

Constructs a CTL formula that queries whether there it is alsways possible to reach one of the given *subspaces*.

Note: This query is equivalent to asking whether every attractor is inside one of the *subspaces*.

Note: Typically this query is used to decide whether a known set of attractors A1, A2, … An is complete, i.e., whether there are any more attractors. To find out pick arbitrary representative states x1, x2, … xn for each attractor and call the function *AGEF_oneof_subspaces* with the argument *Subspaces = [x1, x2, …, xn]*.

---

arguments:

- *subspaces*: a list of subspace

returns:

- *formula*: the CTL formula

example:

```
>>> subspaces = [{"v1":0,"v2":0},{"v2":1}]
>>> all_globally_exists_finally_one_of_sub_spaces(subspaces)
"AG(EF(!v1&!v2 | v2))"
```

exists_finally_nested_reachability(*primes: dict*, *subspaces: List[Union[dict, str]]*) → str

Constructs a CTL formula that queries whether there is a path that visits the given *subspaces* in the order given.

arguments:

- *subspaces*: a list of subspaces

returns:

- *ctl_formula*: the CTL formula

example:

```
>>> subspaces = ["1--", "-01"]
>>> exists_finally_nested_reachability(subspaces)
"EF(v1&EF(!v2&v3))"
```

exists_finally_one_of_subspaces(*primes: dict*, *subspaces: List[Union[dict, str]]*) → str

Constructs a CTL formula that queries whether there is a path that leads to one of the Subspaces.

arguments:

- *subspaces*: a list of subspaces

returns:

- *formula*: the CTL formula

example:

```
>>> subspaces = [{"v1":0,"v2":0}, "1-1--"]
>>> exists_finally_one_of_subspaces(primes, subspaces)
"EF(!v1&!v2 | v1&v3)"
```

exists_finally_unsteady_components(*names: List[str]*) → str

Constructs a CTL formula that queries whether for every variables v specified in *names* there is a path to a state x in which v is unsteady.

Note: Typically this query is used to find out if the variables given in *names* are oscillating in a given attractor.

---

arguments:

- *names*: a list of names of variables

returns:

- *ctl_formula*: the CTL formula

example:

```
>>> names = ["v1","v2"]
>>> exists_finally_unsteady_components(names)
"EF(v1_steady!=0) & EF(v2_steady!=0))"
```

subspace2proposition(*primes: dict*, *subspace: Union[dict, str]*) → str

Constructs a CTL formula that is true in a state x if and only if x belongs to the given Subspace.

Note: Typically this query is used to define INIT constraints from a given subspace.

---

arguments:

- *subspace*: a subspace in string or dictionary representation

returns:

- *proposition*: the proposition

example:

```
>>> subspace = {"v1":0,"v2":1}
>>> init = f"INIT {subspace2proposition(subspace)}"
>>> init
"INIT v1&!v2"
```

4.15 trap_spaces

compute_circuits(*primes: dict*, *max_output: int = 1000*, *fname_asp: Optional[str] = None*, *representation: str = 'dict'*)

Computes minimal trap spaces but also distinguishes between nodes that are fixed due to being part of a circuit and nodes that are fix due to percolation effects.

arguments:

- *primes*: prime implicants
- *max_output*: maximum number of returned solutions
- *fname_asp*: file name or *None*
- *representation*: either "str" or "dict", the representation of the trap spaces

returns:

- *circuits*: of tuples consisting of circuit nodes and percolation nodes

example:

```
>>> compute_circuits(primes)
[({'Mek': 0, 'Erk': 0},{'Raf': 1}),..]
```

compute_smallest_trapspace(*primes: dict*, *state: dict*, *representation: str = 'dict'*)

Returns the (unique) smallest trap space that contains *state*. Calls trapspaces_that_contain_state

arguments:

- *primes*: prime implicants
- *state*: a state in dict format
- *representation*: either "str" or "dict", the representation of the trap spaces

returns:

- *trap_space*: the unique minimal trap space that contains *state*

example:

```
>>> compute_smallest_trapspace(primes, {"v1":1,"v2":0,"v3":0})
```

compute_steady_states(*primes: dict*, *max_output: int = 1000*, *fname_asp: Optional[str] = None*, *representation: str = 'dict'*) → Union[List[dict], List[str]]

Returns steady states.

arguments:

- *primes*: prime implicants
- *max_output*: maximal number of trap spaces to return

- *fname_asp*: file name or *None*
- *representation*: either "str" or "dict", the representation of the trap spaces

returns:

- *states*: the steady states

example:

```
>>> steady = compute_steady_states(primes)
>>> len(steady)
2
```

compute_steady_states_projected(*primes: dict*, *project: List[str] = []*, *max_output: int = 1000*, *fname_asp: Optional[str] = None*)

Returns a list of projected steady states using the **Potassco_** ASP solver *[Gebser2011]*.

arguments:

- *primes*: prime implicants
- *project*: list of names
- *max_output*: maximal number of trap spaces to return
- *fname_asp*: file name or *None*

returns:

- *Activities*: projected steady states

example:

```
>>> steady_states = compute_steady_states_projected(primes, ["v1","v2"])
>>> len(steady_states)
2
>>> compute_steady_states
[{"v1":1,"v2":0},{"v1":0,"v2":0}]
```

compute_trap_spaces(*primes: dict*, *type_: str = 'min'*, *max_output: int = 10000*, *fname_asp: Optional[str] = None*, *representation: str = 'dict'*) → Union[List[dict], List[str]]

Returns a list of trap spaces using the *Potassco* ASP solver, see *Gebser2011*. For a formal introcution to trap spaces and the ASP encoding that is used for their computation see *Klarner2015(a)*.

The parameter *type_* must be one of *"max"*, *"min"*, *"all"* or *"percolated"* and specifies whether subset minimal, subset maximal, all trap spaces or all percolated trap spaces should be returned.

Warning: The number of trap spaces is easily exponential in the number of components. Use the safety parameter *max_output* to control the number of returned solutions.

To create the *asp* file for inspection or manual editing, pass a file name to *fname_asp*.

arguments:

- *primes*: prime implicants
- *type_*: either *"max"*, *"min"*, *"all"* or *"percolated"*
- *max_output*: maximal number of trap spaces to return

- *fname_asp*: name of *asp* file to create, or *None*
- *representation*: either "str" or "dict", the representation of the trap spaces

returns:

- *subspaces*: the trap spaces

example:

```
>>> bnet = ["x, !x | y | z",
...         "y, !x&z | y&!z",
...         "z, x&y | z"]
>>> bnet = "\n".join(bnet)
>>> primes = bnet2primes(bnet)
>>> tspaces = compute_trap_spaces(primes, "all", representation="str")
---, --1, 1-1, -00, 101
```

compute_trap_spaces_bounded(*primes: dict*, *type_: str*, *bounds: tuple*, *max_output: int = 1000*, *fname_asp: Optional[str] = None*)

Returns a list of bounded trap spaces using the **Potassco_** ASP solver *[Gebser2011]*. See trap_spaces for details of the parameters *type_*, *max_output* and *fname_asp*. The parameter *bounds* is used to restrict the set of trap spaces from which maximal, minimal or all solutions are drawn to those whose number of fixed variables are within the given range. Example: `bounds=(5,8)` instructs **Potassco_** to consider only trap spaces with 5 to 8 fixed variables as feasible. *type_* selects minimal, maximal or all trap spaces from the restricted set. .. warning:

```
The *Bound* constraint is applied *before* selecting minimal or maximal trap spaces.
A trap space may therefore be minimal w.r.t. to certain bounds but not minimal in␣
↪the unbounded sense.
```

Use `"n"` as a shortcut for "all variables", i.e., instead of `len(primes)`. Example: Use `bounds=("n","n")` to compute steady states. Note that the parameter *type_* becomes irrelevant for `bounds=(x,y)` with `x=y`.

arguments:

- *primes*: prime implicants
- *type_* in `["max","min","all"]`: subset minimal, subset maximal or all solutions
- *bounds*: the upper and lower bound for the number of fixed variables
- *max_output*: maximal number of trap spaces to return
- *fname_asp*: file name or *None*

returns:

- list of trap spaces

example::

```
>>> tspaces = compute_trap_spaces_bounded(primes, "min", (2,4))
>>> len(tspaces)
12
>>> tspaces[0]
{'TGFR':0,'FGFR':0}
```

compute_trapspaces_that_contain_state(*primes: dict*, *state: dict*, *type_: str*, *fname_asp: Optional[str] = None*, *representation: str = 'dict'*, *max_output: int = 1000*)

Computes trap spaces that contain *state*.

arguments:

- *primes*: prime implicants
- *state*: a state in dict format
- *type_*: either "min", "max", "all" or "percolated"
- *fname_asp*: file name or *None*
- *representation*: either "str" or "dict", the representation of the trap spaces
- *max_output*: maximum number of returned solutions

returns:

- *trap_spaces*: the trap spaces that contain *state*

example:

```
>>> compute_trapspaces_that_contain_state(primes, {"v1":1,"v2":0,"v3":0})
```

compute_trapspaces_that_intersect_subspace(*primes: dict*, *subspace: dict*, *type_: str*, *fname_asp: Optional[str] = None*, *representation: str = 'dict'*, *max_output: int = 1000*) → List[dict]

Computes trap spaces that have non-empty intersection with *subspace*

arguments:

- *primes*: prime implicants
- *subspace*: a subspace in dict format
- *type_*: either "min", "max", "all" or "percolated"
- *fname_asp*: file name or *None*
- *representation*: either "str" or "dict", the representation of the trap spaces
- *max_output*: maximum number of returned solutions

returns:

- *trap_spaces*: the trap spaces that have non-empty intersection with *subspace*

example:

```
>>> compute_trapspaces_that_intersect_subspace(primes, {"v1":1,"v2":0,"v3":0})
```

compute_trapspaces_within_subspace(*primes: dict*, *subspace: dict*, *type_: str*, *fname_asp: Optional[str] = None*, *representation: str = 'dict'*, *max_output: int = 1000*) → Union[List[dict], List[str]]

Computes trap spaces contained within *subspace*

arguments:

- *primes*: prime implicants
- *subspace*: a subspace in dict format
- *type_*: either "min", "max", "all" or "percolated"
- *fname_asp*: file name or *None*
- *representation*: either "str" or "dict", the representation of the trap spaces
- *max_output*: maximum number of returned solutions

returns:

- *trap_spaces*: the trap spaces contained within *subspace*

example:

```
>>> trapspaces_in_subspace(primes, {"v1":1,"v2":0,"v3":0})
```

is_trap_space(*primes: dict*, *subspace: Dict[str, int]*) → bool

Tests whether *subspace* is a trap space in *primes*.

arguments:

- *primes*: prime implicants
- *subspace*: a subspace

returns:

- *result: whether *subspace* is a trap space

example:

```
>>> is_trap_space(primes=primes, subspace={'RAF':1, 'ERK':0, 'MEK':1})
True
```

CHAPTER

FIVE

REPOSITORY

The repository contains bnet files for various example networks including citations for publications where appropriate. Each network has a name by which its bnet string and primes dictionary can be retrieved. These are get_bnet and get_primes, respectively. In addition the function get_all_names can be used for iterating over all existing names. The following is a list of all networks available in *pyboolnet*.

The motivation for the repository is to make it easier to share networks by referring to a fixed file, to gather analysis results and provide a benchmark set for analysis tools. Further information about the repository can be found at:

https://github.com/hklarner/pyboolnet/tree/master/repository

5.1 arellano_rootstem

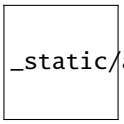


Fig. 1: Interaction graph of arellano_rootstem.

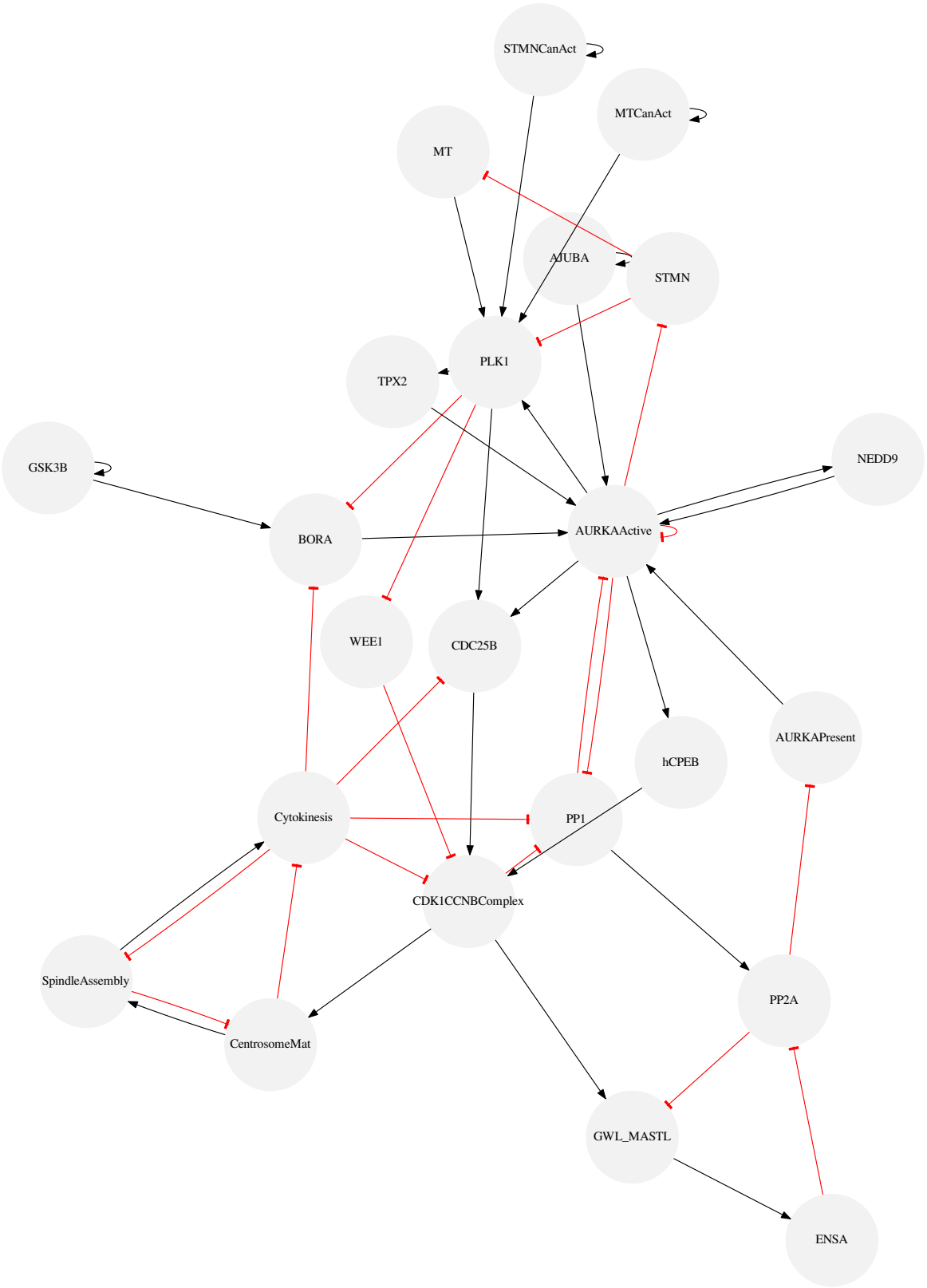


Fig. 2: Interaction graph of dahlhaus_neuroplastoma.

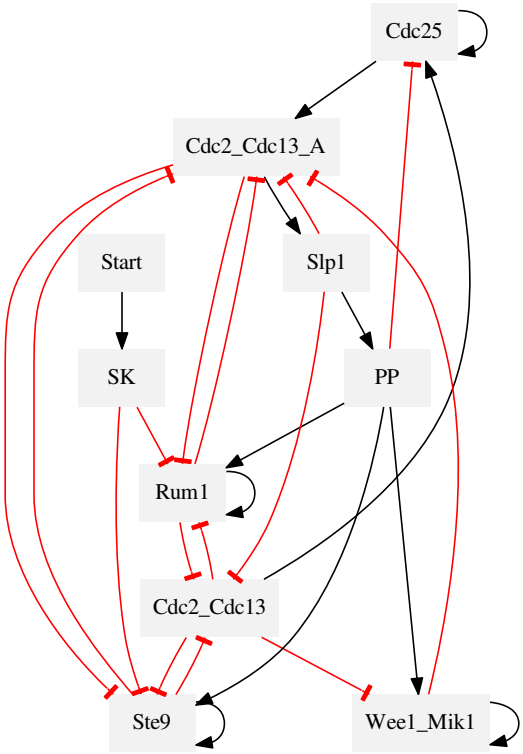


Fig. 3: Interaction graph of davidich_yeast.

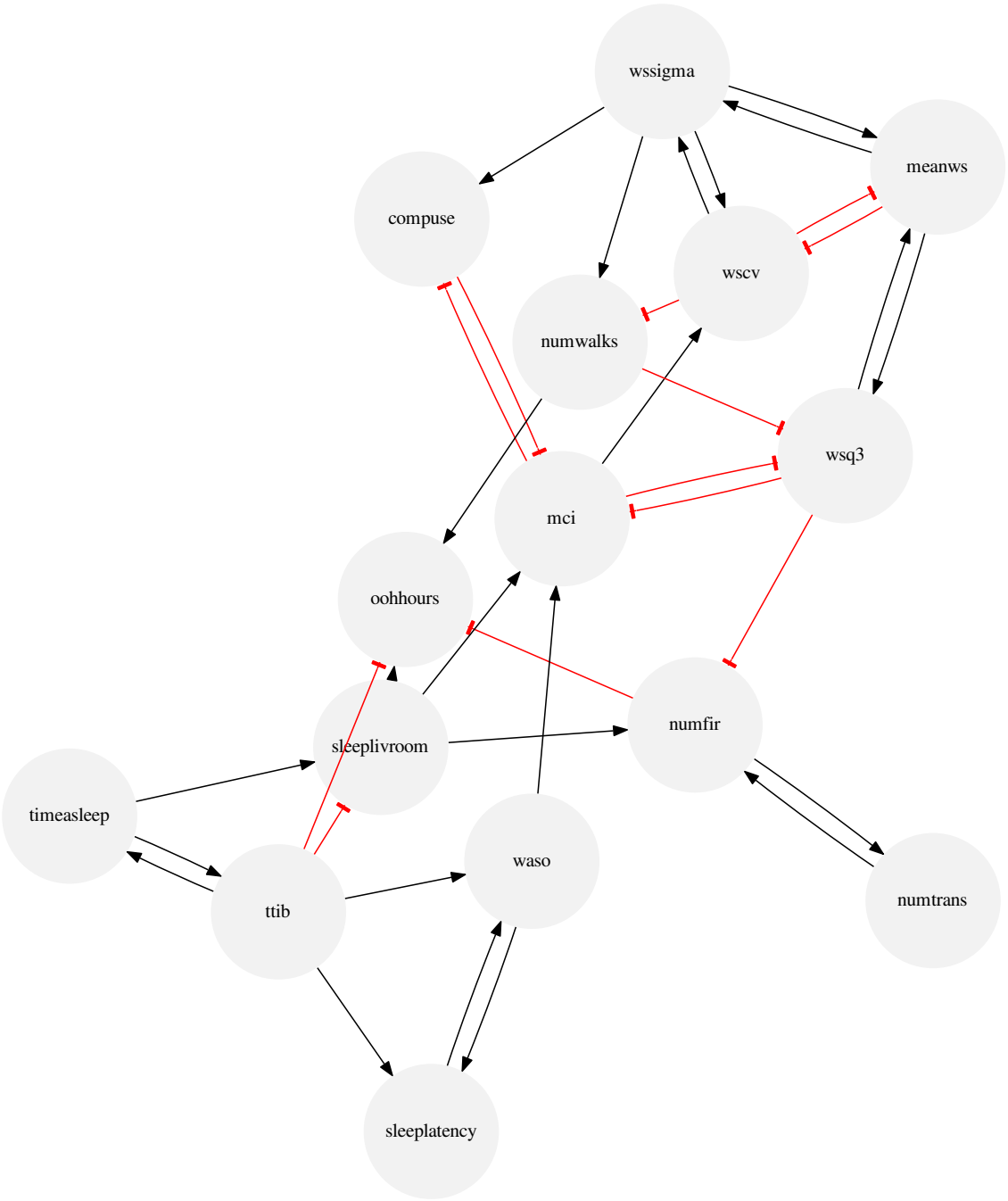


Fig. 4: Interaction graph of dinwoodie_life.

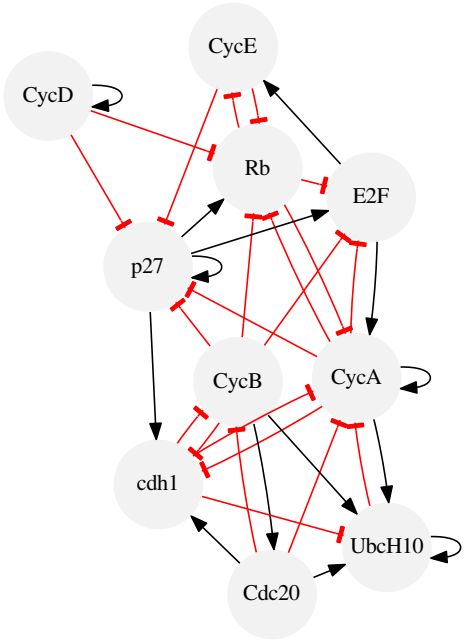


Fig. 5: Interaction graph of faure_cellcycle.

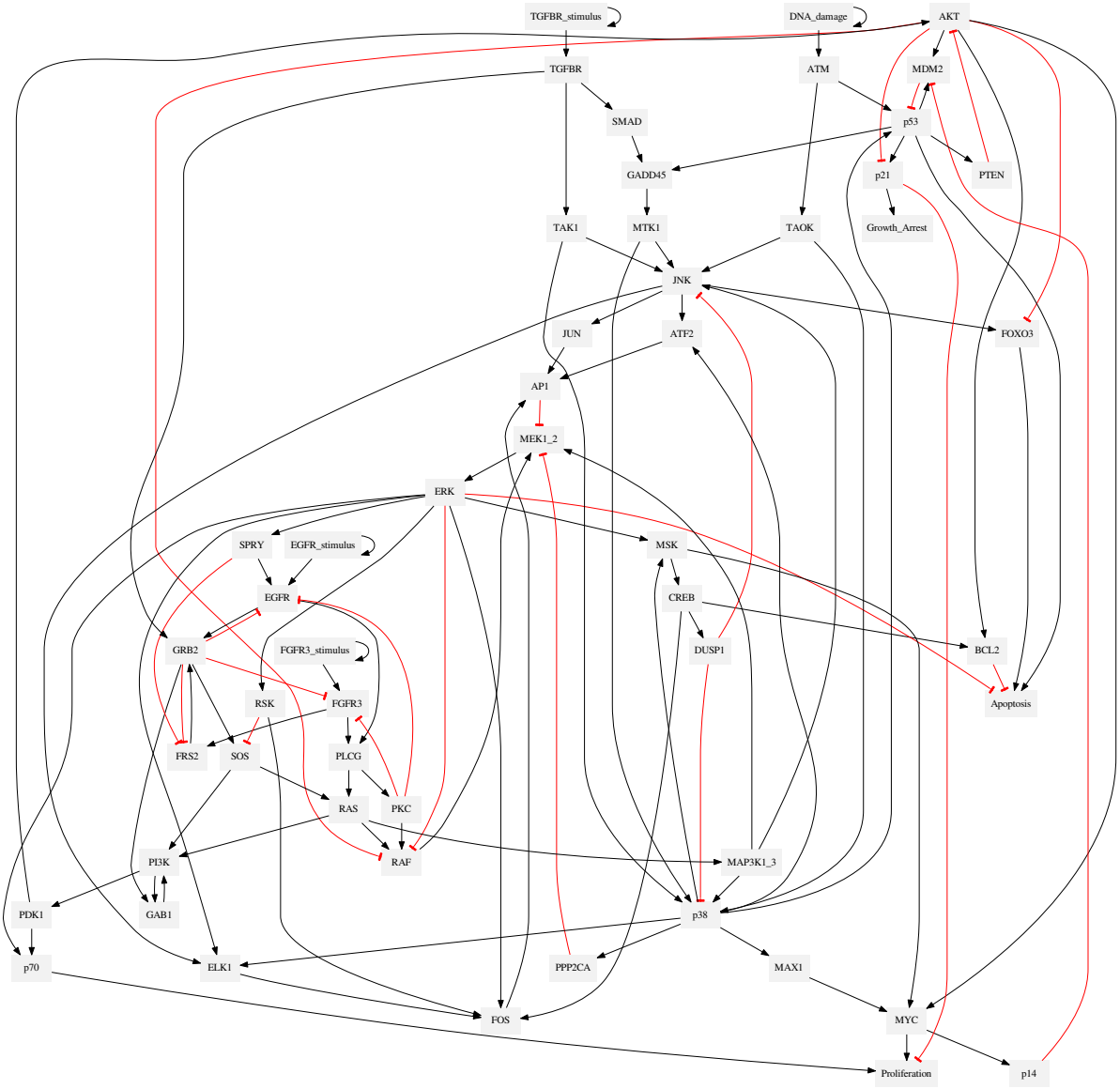


Fig. 6: Interaction graph of grieco_mapk.

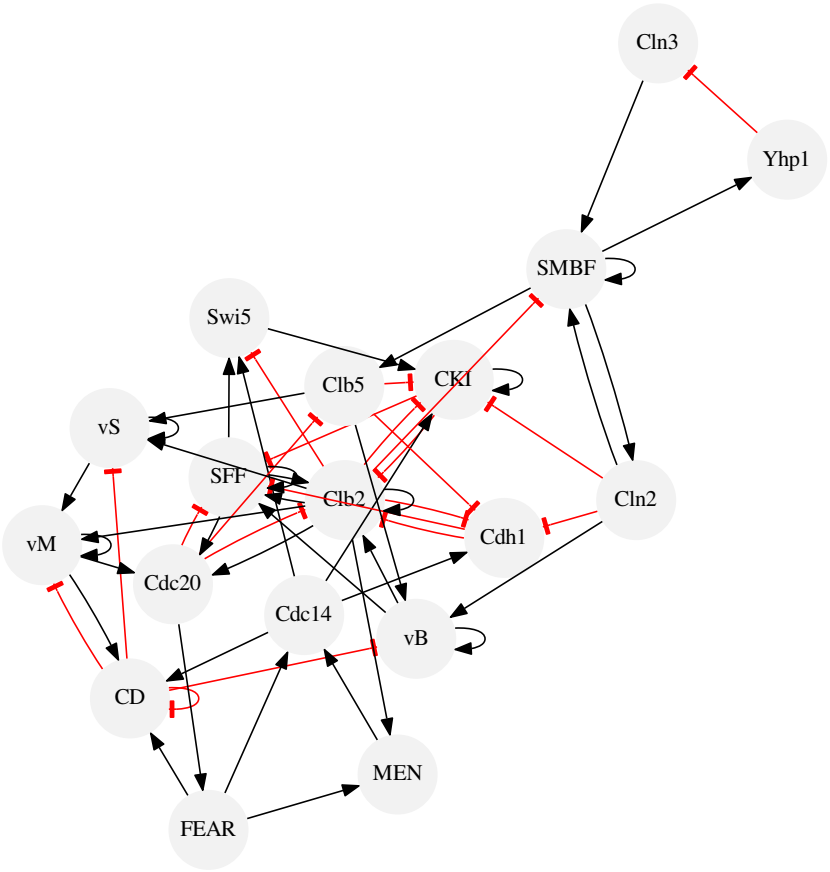


Fig. 7: Interaction graph of irons_yeast.

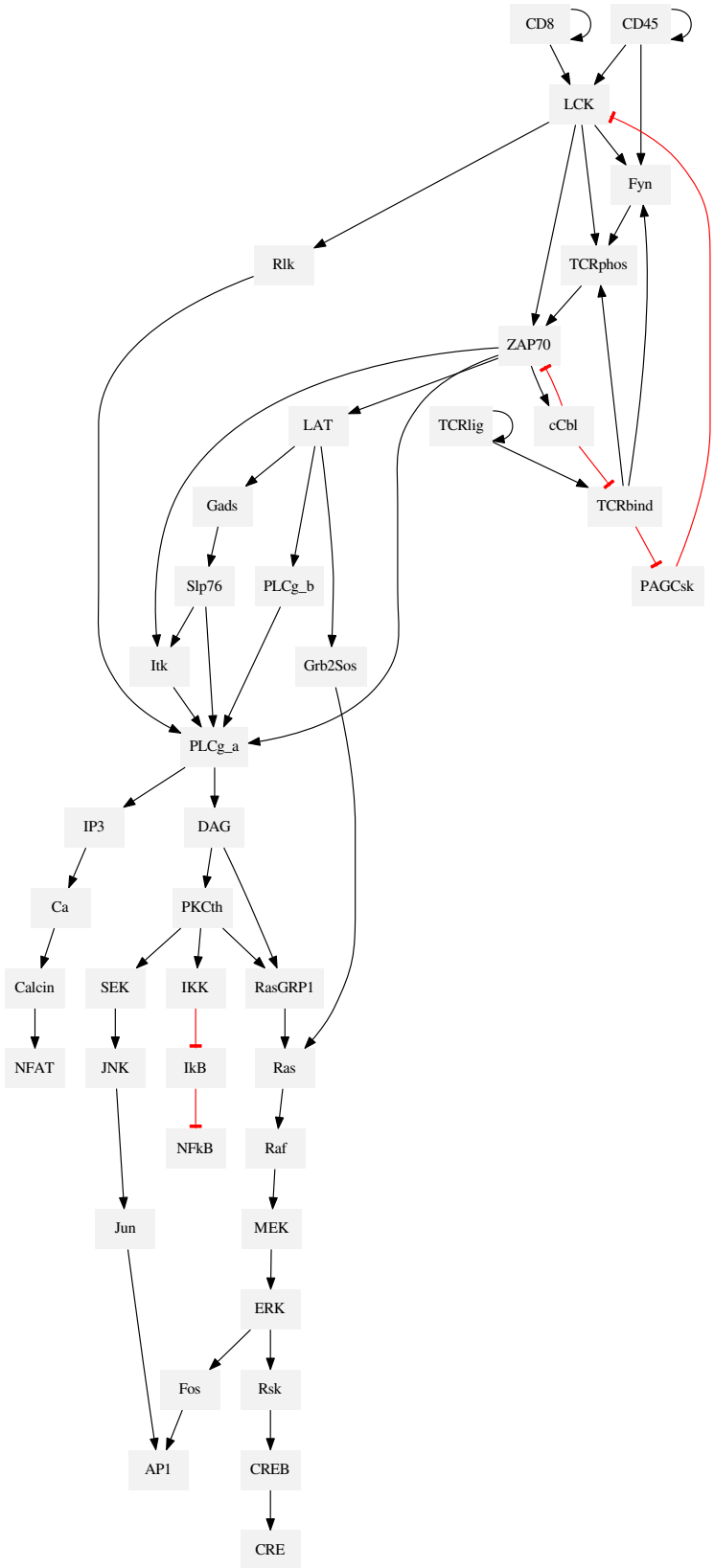


Fig. 8: Interaction graph of klamt_tcr.

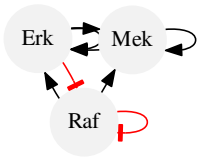


Fig. 9: Interaction graph of raf.

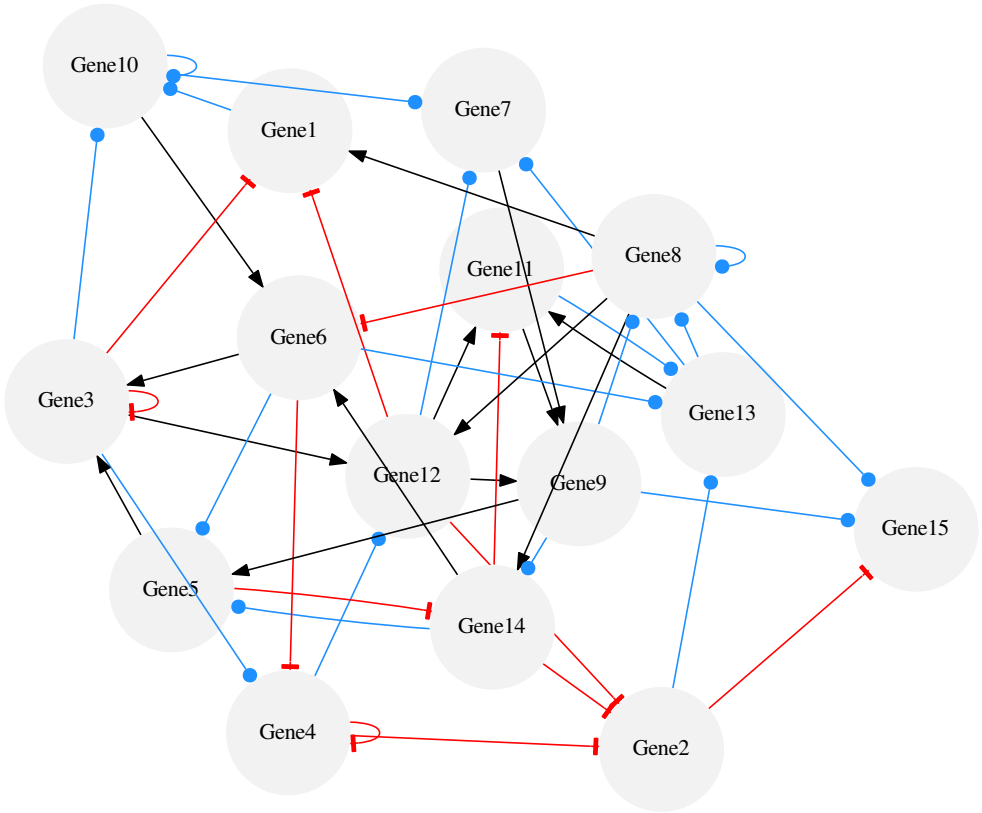


Fig. 10: Interaction graph of randomnet_n15k3.

5.2 dahlhaus_neuroplastoma

5.3 davidich_yeast

5.4 dinwoodie_life

5.5 faure_cellcycle

5.6 grieco_mapk

5.7 irons_yeast

5.8 klamt_tcr

5.9 raf

5.10 randomnet_n15k3

5.11 randomnet_n7k3

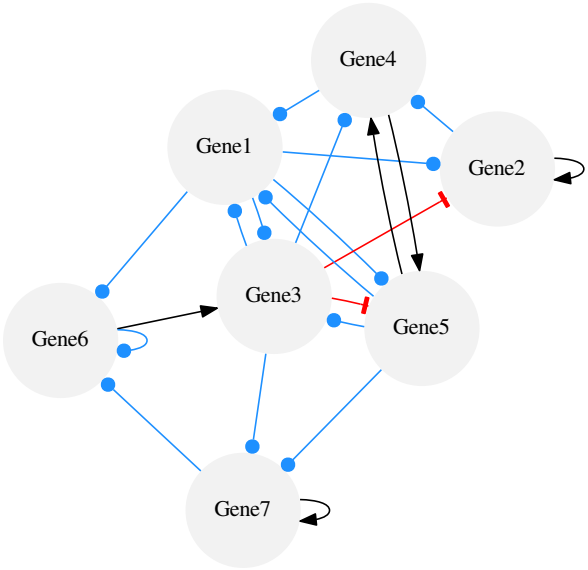


Fig. 11: Interaction graph of randomnet_n7k3.

5.12 remy_tumorigenesis

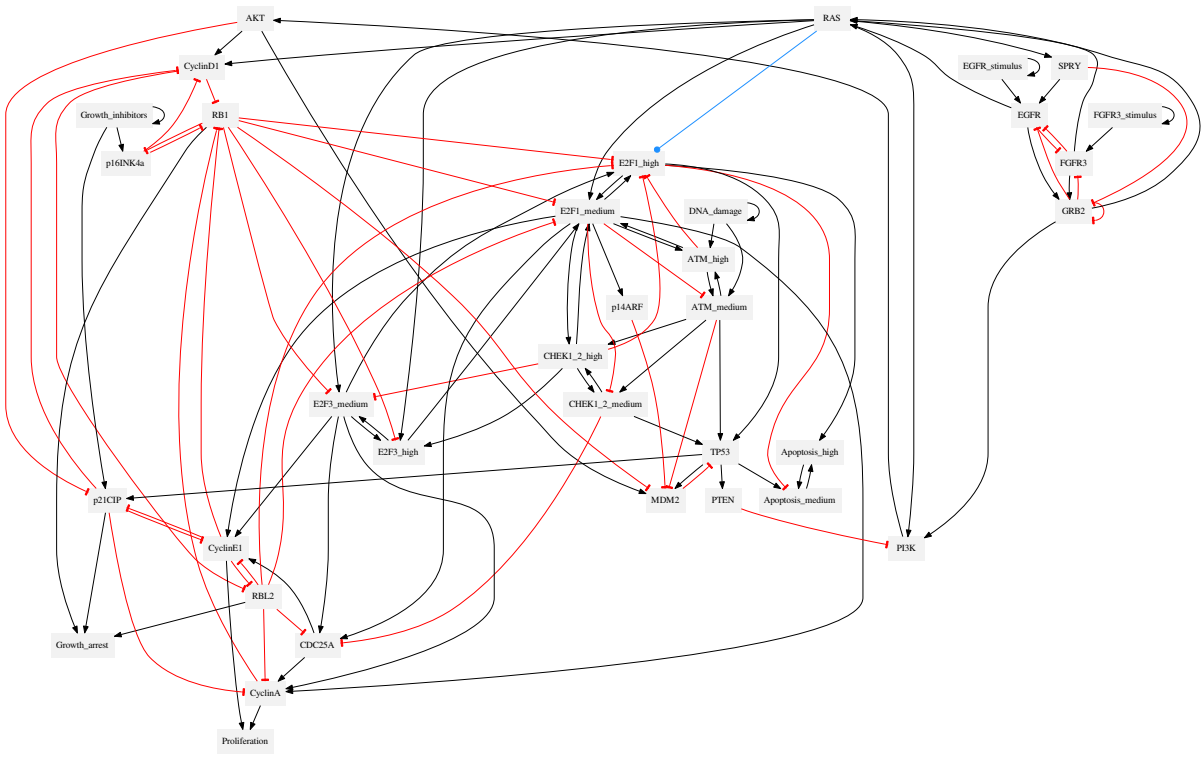


Fig. 12: Interaction graph of remy_tumorigenesis.

5.13 saadatpour_guardcell

5.14 tournier_apoptosis

5.15 xiao_wnt5a

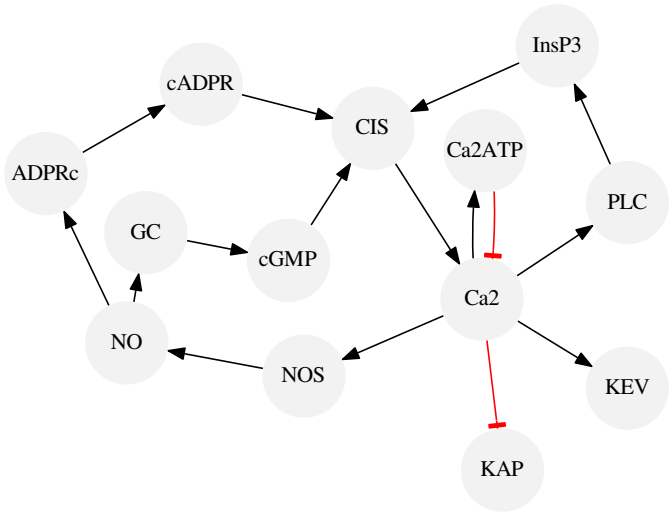


Fig. 13: Interaction graph of saadatpour_guardcell.

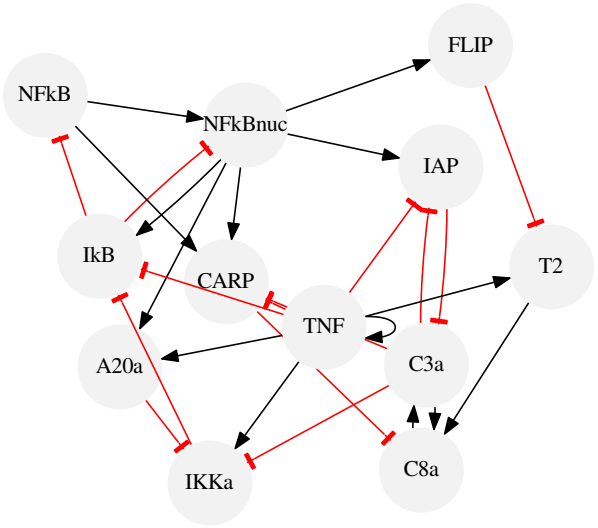


Fig. 14: Interaction graph of tournier_apoptosis.

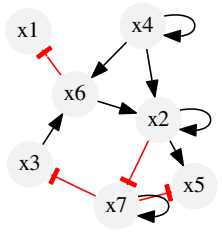


Fig. 15: Interaction graph of xiao_wnt5a.

CHAPTER

SIX

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.

Didier2011: Mapping multivalued onto Boolean dynamics. G. Didier, E. Remy, and C. Chaouiya. Journal of Theoretical Biology, 2011.

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.

Klarner2018: Basins, Commitment Sets and Phenotypes of Boolean networks. H. Klarner, F. Heinitz, S. Nee, H. Sienert. In preparation, 2018.

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: 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

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

R

S

U

W