PyBoolNet Documentation

Release 1.0

Hannes Klarner

March 02, 2016

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://sourceforge.net/projects/boolnetfixpoints.

Note: For compatibility reasons the sourceforge project is still called *boolnetfixpoints* instead of *pyboolnet*.

Note: *PyBoolNet* does not yet have any user friendly error messages. Please post questions in the discussion section of the sourceforge project page

- https://sourceforge.net/p/boolnetfixpoints/discussion

and report bugs or suggest features at

- https://sourceforge.net/p/boolnetfixpoints/tickets

CHAPTER

ONE

INSTALLATION

1.1 Linux

Download `PyBoolNet-1.0_linux.tar.gz` from https://sourceforge.net/projects/boolnetfixpoints. *PyBoolNet* is written in Python 2.7 and packaged using *Distutils*. We recommend to install the package using *pip*:

```
$ sudo pip install PyBoolNet-1.0_linux.tar.gz
```

which should place the package here:

```
/home/<user>/.local/lib/python2.7.x/dist-packages/PyBoolNet
```

where `<user>` is the name you are logged in with (to find out, type `whoami`). Use the option `--user` (this time literally, do not replace it with you actual user name) if you do not have sudo rights:

```
$ pip install PyBoolNet-1.0.tar.gz --user
```

The package is likely going to be placed here:

```
/usr/local/lib/python2.7.x/dist-packages/PyBoolNet
```

To install *PyBoolNet* using *Distutils* unpack *PyBoolNet-1.0.tar.gz* into a temporary folder and run:

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

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

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

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

You should now be able to import *PyBoolNet*:

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

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

```
$ pip uninstall PyBoolNet
```

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

1.2 Windows

Not available yet.

1.3 Mac

Not available yet.

CHAPTER TWO

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. If you are using Linux the dependencies should work out of the box. Otherwise you will have to download the binaries that fit your OS and modify the paths to the executables. The file that records the locations is called `settings.cfg` and located in the folder `Dependencies` of *PyBoolNet*. The default location is:

```
/usr/local/lib/python2.7/dist-packages/PyBoolNet/Dependencies/settings.cfg
```

The file is a standard configuration file of `name = value` pairs. The default is:

```
[Executables]
nusmv           = ./NuSMV-2.5.4/bin/NuSMV
gringo          = ./gringo-4.4.0-x86-linux/gringo
clasp           = ./clasp-3.1.1/clasp-3.1.1-x86-linux
bnet2prime      = ./BNetToPrime/BNetToPrime
dot             = /usr/bin/dot
```

Simply replace the default paths with the paths to your own installations. Note that `./` indicates a relative path while `/` is an absolute path.

To test whether the dependencies are correctly installed, run:

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

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$
```

2.1 BNetToPrime

BNetToPrime stands for “Boolean network to prime implicants”. It is necessary to compute the prime implicants of a Boolean network. The binaries and source are available at:

https://github.com/xstreck1/BNetToPrime

2.2 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)*.

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 recommend you use them.

The binaries and source are available at:

https://sourceforge.net/projects/potassco/files/clasp/3.1.1

https://sourceforge.net/projects/potassco/files/gringo/4.4.0

2.3 NuSMV

NuSMV is a symbolic model checker that we use to decide LTL and CTL queries.

Note: *PyBoolNet* is tested with NuSMV-2.5.4.

Binaries and source available at:

http://nusmv.fbk.eu

2.4 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 graph. To install it on Linux run:

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

2.5 ImageMagick

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

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

ImageMagick is available at

http://www.imagemagick.org/

2.6 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:

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

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

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

2.7 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 and to use GINsim models you need to export them as *sbml-qual* files which can then be converted 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/

2.8 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:

https://networkx.github.io

To install it using *pip* run:

```
$ sudo pip install networkx>=1.10
```

or:

```
$ pip install networkx>=1.10 --user
```

if you do not have super user rights.

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

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*, *str2state* and *subspace2str*, *str2subspace*.

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

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

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

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

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

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

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

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

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

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

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

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

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 = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> igraph
<networkx.classes.digraph.DiGraph object at 0xb513efec>
```

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

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

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

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

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

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

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

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

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

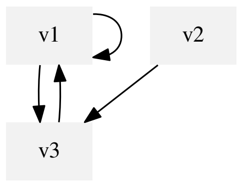


Figure 3.1: The interaction graph "*example02_igraph.pdf*" of the network defined above.

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*
- *fixedwidth*: specifies whether the width of a node is fixed, either *"true"* or *"false"*
- *width*: sets the node width, e.g. *5*, *10*, *15*

Colors can be set by names like *"red"*, *"green"* or *"blue"*, or by space-separated HSV values, e.g. *"0.1 0.2 1.0"*, or by a RGB value, e.g *"#40e0d0"*. For a list of predefined color names see for example http://www.graphviz.org/doc/info/colors.html.

The basic edge attributes are:

- *arrowhead*: sets the shape of the arrow, e.g. *"dot"*, *"tee"*, *"normal"*

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

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

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

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

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

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

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

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

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

The result is shown in *the figure below*.

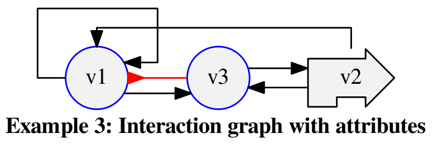


Figure 3.2: The interaction graph "*example03_igraph.pdf*".

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*.

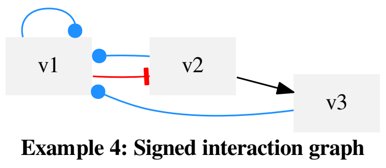


Figure 3.3: The interaction graph "*example04_igraph.pdf*" with attributes added by *add_style_interactionsings*.

3.2.3 styles for inputs, outputs and constants

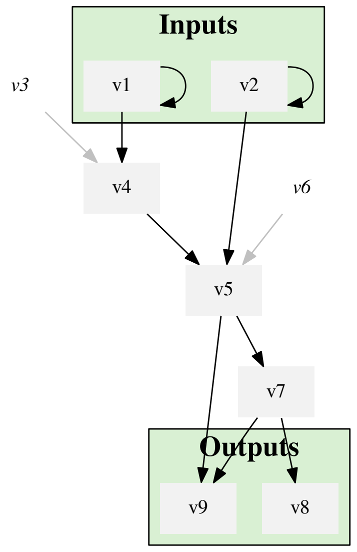
Inputs are components that are only regulated by themselves. Usually, inputs regulate themselves positively but we also consider inputs that regulate themselves negatively as inputs. *Outputs* are components that do not regulate other components and *constants* are components whose update function is 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 = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> IGs.add_style_inputs(igraph)
>>> IGs.add_style_constants(igraph)
>>> IGs.add_style_outputs(igraph)
>>> igraph.graph["label"] = "Example 5: Interaction graph with styles for"+
...                          "inputs, outputs and constants"
>>> IGs.igraph2image(igraph, "example05_igraph.pdf")
```

The result is shown in *the figure below*.



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

Figure 3.4: The interaction graph "*example05.pdf*" with styles added by *add_style_inputs*, *add_style_outputs* and *add_style_constants*.

3.2.4 the SCCs and condensation style

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

Consider the network:

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

The result is shown in *the figure below*.

In addition you may use *add_style_condensation* to add a small "map" of the SCC graph, which we call the condensation graph. Each SCC is replaced by a single node and there is an edge between two SCCs iff there is a "cascade path" between them, see *Klarner2015(b)* for a formal definition. Since styles are additive we generate example 7 by the following lines:

```
>>> IGs.add_style_condensation(igraph)
>>> igraph.graph["label"] = "Example 7: Interaction graph with SCC and condensation style"
>>> IGs.igraph2image(igraph, "example07_igraph.pdf")
```

The result is shown in *the figure below*.

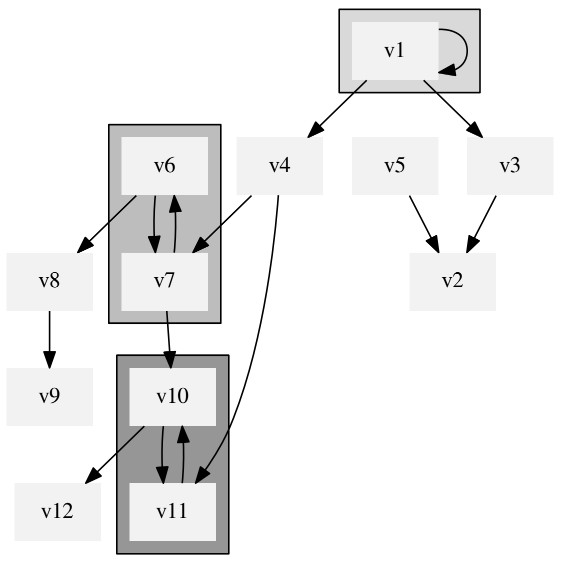


Figure 3.5: The interaction graph "*example06_igraph.pdf*" with attributes added by *add_style_sccs*.

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 either as a list of names of variables or as a tuple that consists of a list of names and a dictionary of *dot* attributes for that subgraph, e.g., a label or background color.

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

Consider the network:

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

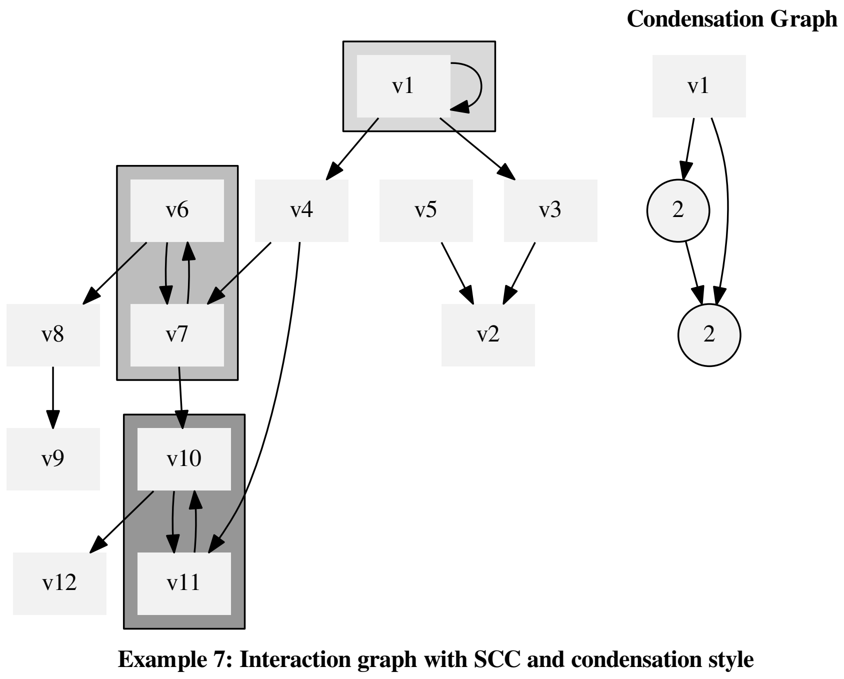

Condensation Graph
v1
v6
v4
v5
v3
v8
v7
v2
v9
v10
v12
v11
v1
2
2
Example 7: Interaction graph with SCC and condensation style

Figure 3.6: The interaction graph "*example07_igraph.pdf*" with attributes added by *add_style_sccs* and *add_style_condensation*.

```
>>> 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*.

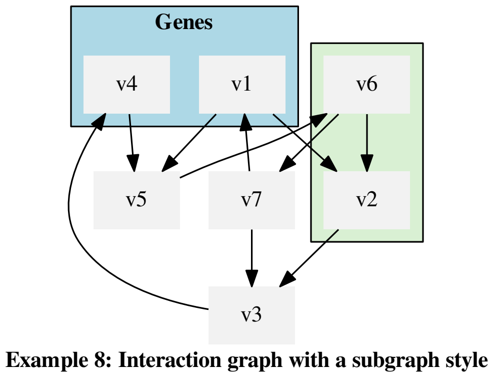


Figure 3.7: The interaction graph “*example08_igraph.pdf*” with attributes added by *add_style_subgraphs*.

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

The result is shown in *the figure below*.

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")
```

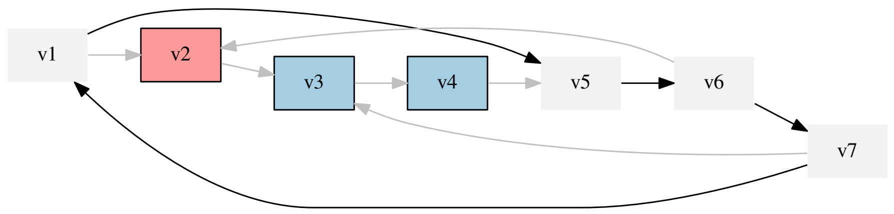


Figure 3.8: The interaction graph "*example09_igraph.pdf*" with attributes added by *add_style_activities*.

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 = FEX.bnet2primes(bnet)
>>> igraph = IGs.primes2igraph(primes)
>>> IGs.add_style_default(igraph)
>>> igraph.graph["label"] = "Example 10: Interaction graph with default style"
>>> IGs.igraph2image(igraph, "example10_igraph.pdf")
```

The result is shown in *the figure below*.

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 = FEX.bnet2primes(bnet)
>>> update = "asynchronous"
>>> stg = STGs.primes2stg(primes, update)
>>> stg
<networkx.classes.digraph.DiGraph object at 0xb3c7d64c>
```

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

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

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

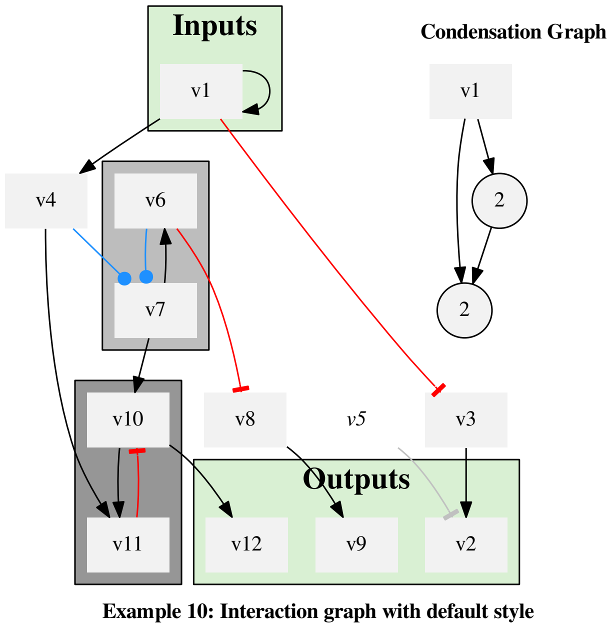


Figure 3.9: The interaction graph "*example10_igraph.pdf*" with attributes added by *add_style_default*.

```
>>> 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}
```

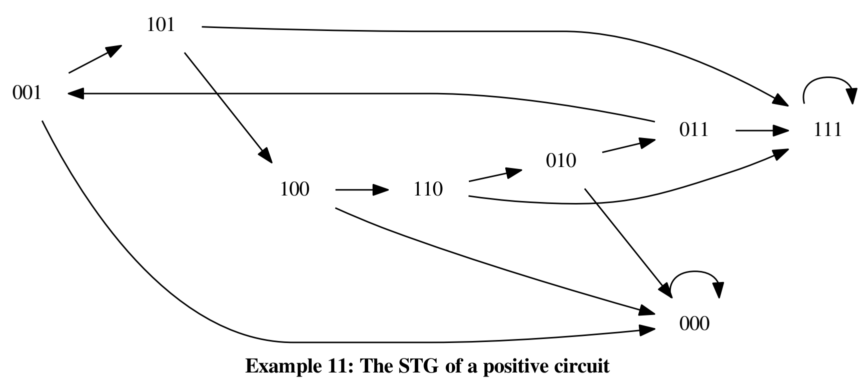


Figure 3.10: The state transition graph "*example11_stg.pdf*" of an isolated feedback circuit.

To construct the STG starting from initial states call:

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

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

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

The result is shown in *the figure below*.

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 = FEX.bnet2primes(bnet)
```

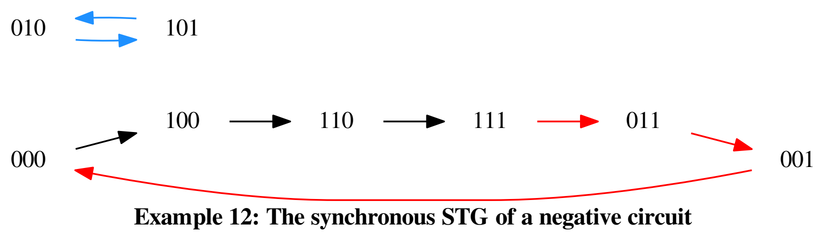


Figure 3.11: The state transition graph "*example12_stg.pdf*" with attributes added by *add_style_tendencies*.

```
>>> 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 = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> stg.graph["label"] = "The SCC style"
>>> STGs.add_style_sccs(stg)
>>> STGs.stg2image(stg, "example14_stg.pdf")
```

The result is shown in *the figure below*.

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)* and also *trap_spaces_and_attractors*:

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

The result is shown in *the figure below*.

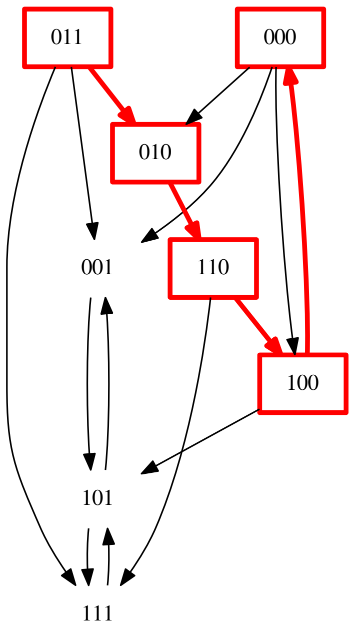


Figure 3.12: The state transition graph “*example13_stg.pdf*” with attributes added by *add_style_path*.

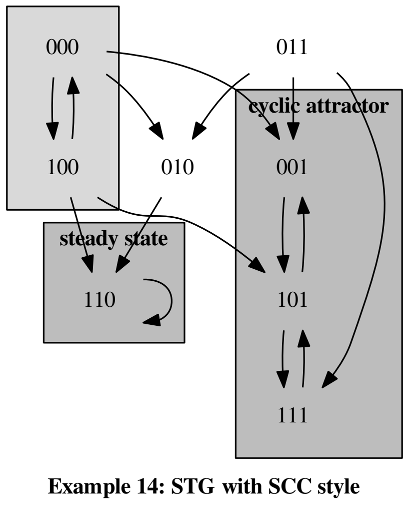


Figure 3.13: The state transition graph “*example14_stg.pdf*” with attributes added by *add_style_sccs*.

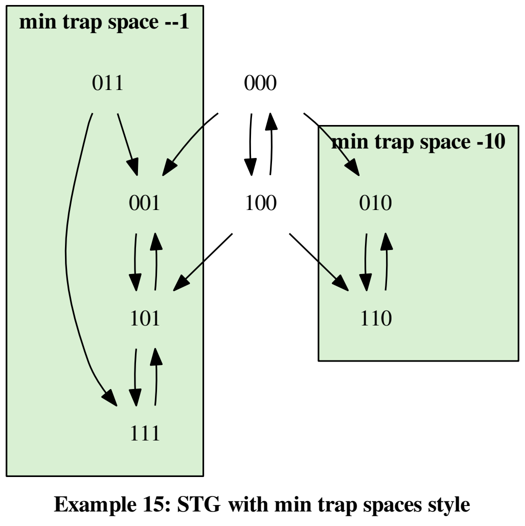


Figure 3.14: The state transition graph “*example15_stg.pdf*” with attributes added by *add_style_mintrapspaces*.

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 = FEX.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.

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

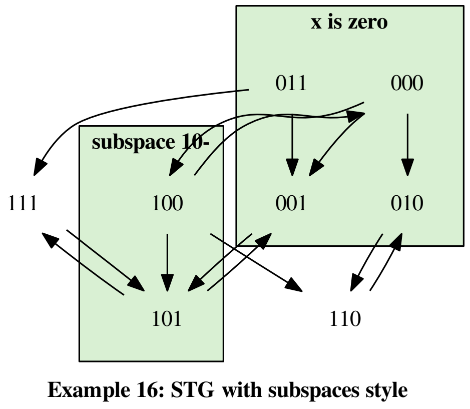


Figure 3.15: The state transition graph *"example16_stg.pdf"* with attributes added by *add_style_subspaces*.

```
>>> STGs.add_style_default(primes, stg)
>>> STGs.stg2image(stg, "example17_stg.pdf")
```

The result is shown in *the figure below*.

3.4 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*.

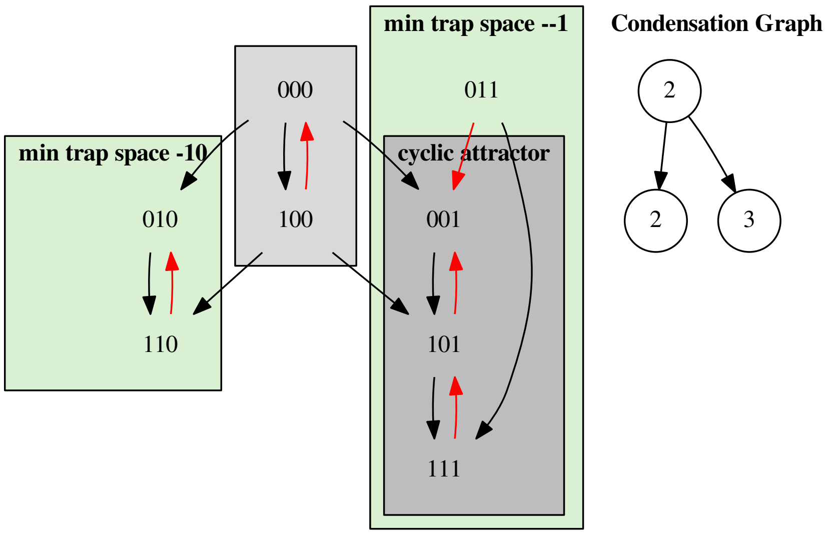
3.4.1 Transition Systems

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

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

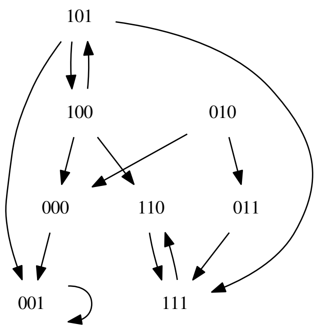
The state transition graph is shown in *the figure below*.

When model checking, *PyBoolNet* 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-"*.

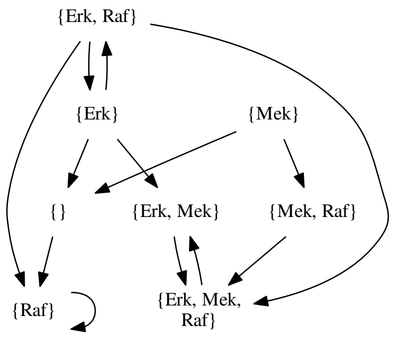


Example 17: STG with default style

Figure 3.16: The state transition graph *"example17_stg.pdf"* with attributes added by *add_style_default*.



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



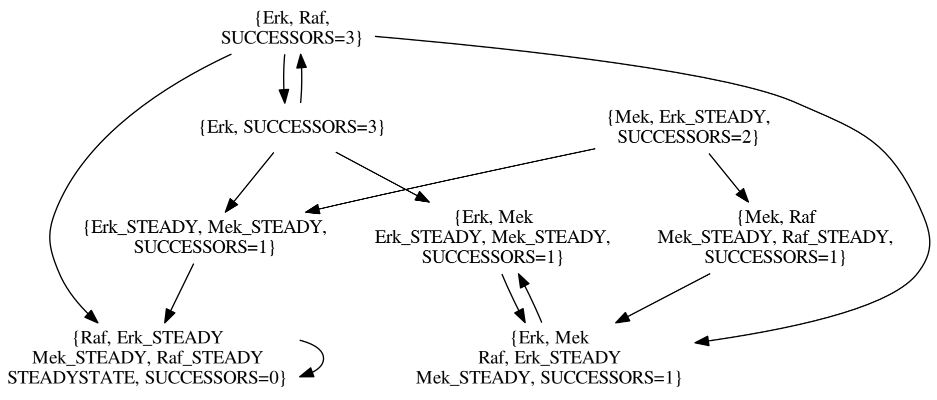
Example 19: Basic transition system of Erk-Mek-Raf

Figure 3.17: 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*.


{Erk, Raf, SUCCESSORS=3}
{Erk, SUCCESSORS=3}
{Mek, Erk_STEADY, SUCCESSORS=2}
{Erk_STEADY, Mek_STEADY, SUCCESSORS=1}
{Erk, Mek Erk_STEADY, Mek_STEADY, SUCCESSORS=1}
{Mek, Raf Mek_STEADY, Raf_STEADY, SUCCESSORS=1}
{Raf, Erk_STEADY Mek_STEADY, Raf_STEADY STEADYSTATE, SUCCESSORS=0}
{Erk, Mek Raf, Erk_STEADY Mek_STEADY, SUCCESSORS=1}

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

Figure 3.18: The extended transition system for the Erk-Mek-Raf network.

3.4.2 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)
>>> 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.

3.4.3 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 pass the argument *DisableCounterExamples=False* to the function *check_primes*. The function will then always return a tuple that consists of 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(primes, update, init, spec, False)
```

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

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

Here, *state2str* is a function of the module *StateTransitionGraphs* that 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")
```

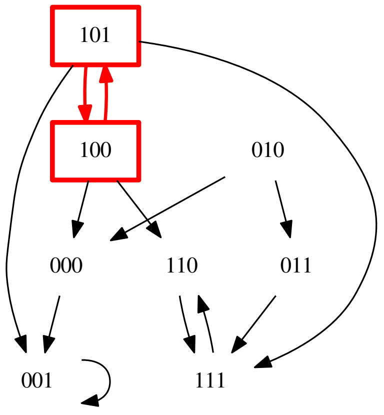


Figure 3.19: The state transition graph *"example18_stg.pdf"* of the Erk-Mek-Raf network with a path style that indicates a counterexample to the LTL query with all states being initial and the formula *F(Raf & F(STEADYSTATE))*.

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")
```

3.4.4 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 = FEX.bnet2primes(bnet)
>>> update = "asynchronous"
```

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

```
>>> stg = STGs.primes2stg(primes, update)
>>> for x in stg.nodes():
...     x_dict = STGs.str2state(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(primes, update, init, spec, False)
>>> answer
False
>>> STGs.add_style_path(stg, counterex, "red")
>>> stg.graph["label"] = "Example 21: Counterexample"
>>> STGs.stg2image(stg, "example21_stg.pdf")
```

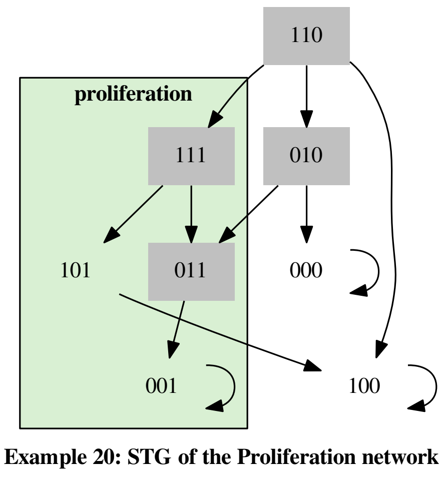


Figure 3.20: The state transition graph *"example20_stg.pdf"* of the Proliferation network with initial states highlighted by gray rectangles and proliferation states gathered in a subgraph.

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

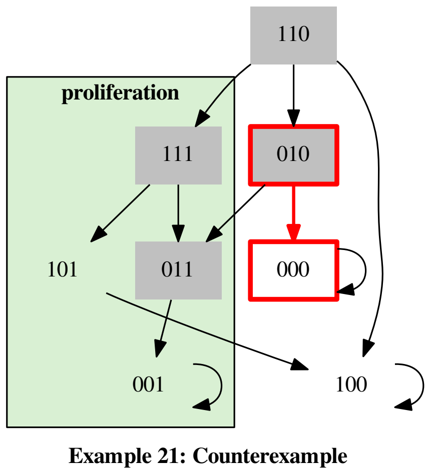


Figure 3.21: The state transition graph *"example21_stg.pdf"* of the Proliferation network with a counterexample highlighted by path.

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

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

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

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

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

3.4.5 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(primes, update, init, spec, False)
>>> counterex
({'DNAdamage': 1, 'Proliferation': 1, 'GrowthFactor': 0},)
>>> STGs.state2str(counterex[0])
101
```

The counterexample returns a paths that consists of a single state, namely 101, that does not satisfy *EX(Proliferation)*, i.e., that does not have a successor that is a proliferation state. 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(primes, update, init, spec, False)
>>> state = counterex[0]
```

3.4.6 Existential Queries

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

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

and

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

As an example consider the following network:

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

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

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

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

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

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

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

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

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

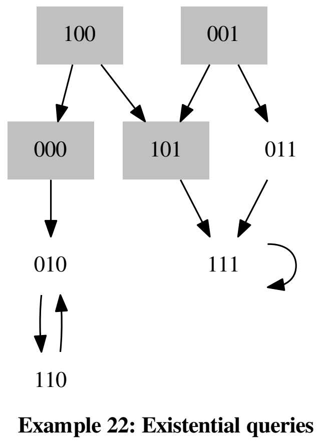


Figure 3.22: The state transition graph *"example22_stg.pdf"* with initial states highlighted by gray rectangles. The attractors are the steady state 111 and the cyclic attractor that consists of the states 010 and 110.

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

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

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

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

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

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

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

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

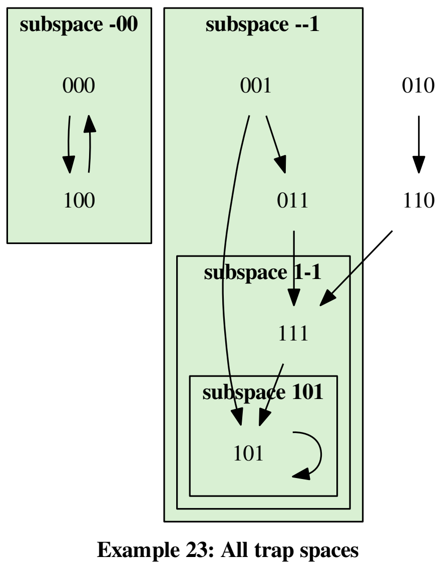


Figure 3.23: The state transition graph *"example23_stg.pdf"* with every trap space highlighted by its own subgraph.

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

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

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

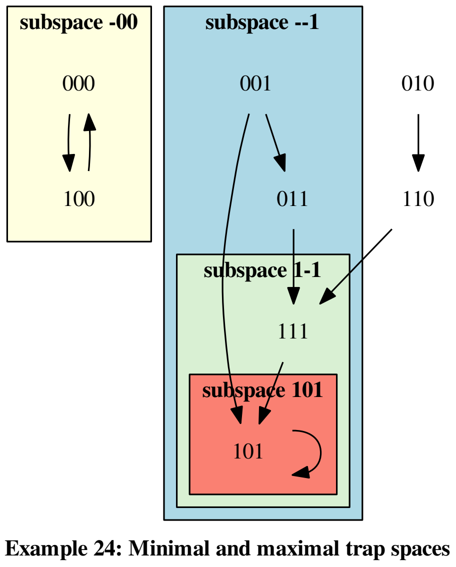


Figure 3.24: The state transition graph *"example24_stg.pdf"* with minimal trap spaces in red, maximal trap spaces in blue, trap spaces that are minimal and maximal at the same time in yellow and the remaining trap spaces in green.

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.

3.6 Attractors

3.6.1 Attractor Detection

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

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

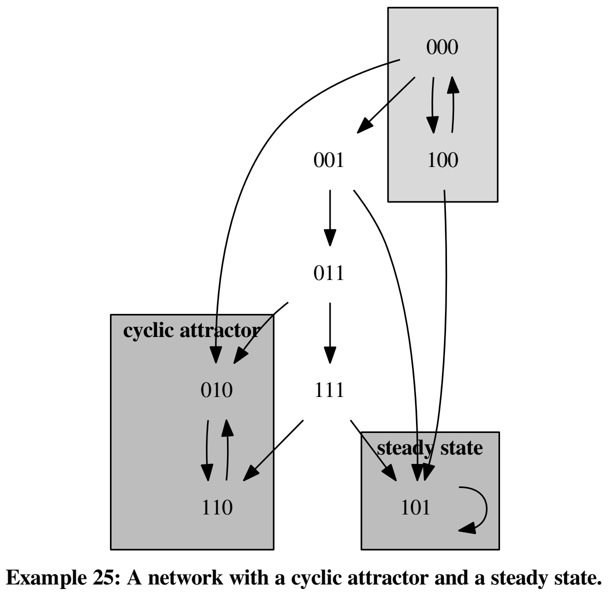

000
001
100
011
cyclic attractor
010
111
steady state
110
101
Example 25: A network with a cyclic attractor and a steady state.

Figure 3.25: The asynchronous STG *"example25_stg.pdf"* of a network with a steady state and a cyclic attractor.

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

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

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

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

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

```
>>> state = AD.find_attractor_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_by_randomwalk_and_ctl* should always be encapsulated in a *Try-Except* block:

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

3.6.2 Approximations

This section is based on the results published in *Klarner2015(a)*. The overall goal is to efficiently compute all attractors of an asynchronous STG. In particular we are interested in the cyclic attractors of an asynchronous STG because its steady states are identical to the steady states of synchronous STGs which can already be computed efficiently using SAT solvers, e.g., the approach suggested in *Dubrova2011*.

The idea is based on the observation that the efficiency of computing minimal trap spaces is about as efficient as computing steady states and using similar techniques, i.e., solvers for 0-1 problems. Also, trap spaces always contain at least some attractors. It is therefore try to quanity what "some" means. This section asks whether

1. the minimal trap spaces are **complete**, i.e., whether every attractor of the given STG is contained in one of its minimal trap spaces
2. each minimal trap space is **univocal**, i.e., whether each minimal trap spaces contains exactly one attractor
3. each minimal trap space is **faithful**, i.e., the number of its free variables is equal to the number of oscillating variables in each of its attractors

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

As an example of a network whose minimal trap spaces are complete, univocal and faithful consider again the network defined in *the figure above*. The functions *univocal* and *faithful* each require the primes, update strategy and a trap space. Since *univocal* 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, the function returns a triplet consisting of the answer, an attractor state and a counterexample (if the trap space is not univocal), see *univocal* for details. The function *faithful* returns a tuple that consists of the answer and a counterexample (if it exists), again see *faithful* for details. For now we are only interested in printing the answer of each property for each trap space:

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

The naive function for deciding whether a set of trap spaces if complete requires also three arguments, the primes, the update strategy and a list of trap spaces. It returns a tuple consisting of the answer and a counterexample, if it exists. See *completeness_naive* for details.

```
>>> answer_complete, _ = AD.completeness_naive( primes, update, mintspaces )
>>> "min trap spaces are complete:", answer_complete
min trap spaces are complete: True
```

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

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

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

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

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

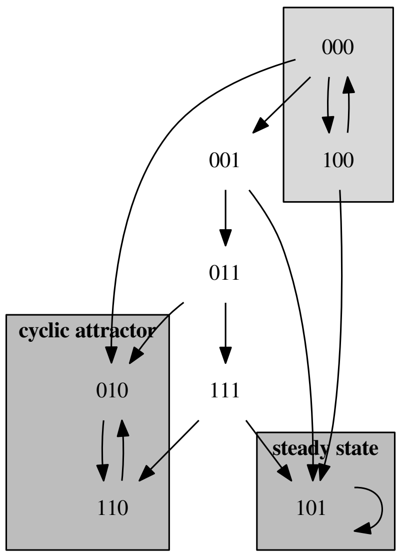

000
001
100
011
cyclic attractor
010
111
110
steady state
101
Example 25: A network with a cyclic attractor and a steady state.

Figure 3.26: The state transition graph *"example25_stg.pdf"* in which the minimal trap space "—" is not univocal.

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

CHAPTER

FOUR

REFERENCE

4.1 FileExchange

4.1.1 bnet2primes

bnet2primes (*BNET*, *FnamePRIMES=None*)

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

Note: Requires the program *BNetToPrime*.

arguments:

- *BNET*: name of *bnet* file or string contents of file
- *FnamePRIMES*: *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")
```

4.1.2 primes2bnet

primes2bnet (*Primes*, *FnameBNET*, *Minimize=False*)

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
- *FnameBNET* (str): name of *bnet* file or *None* for the string of the file contents

- *Minimize* (bool): whether the expressions should be minimized

returns:

- *BNET* (str) if *FnameBNET=None* or *None* otherwise

example:

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

4.1.3 write_primes

write_primes(*Primes*, *FnamePRIMES*)

Saves *Primes* as a *json* file.

arguments:

- *Primes*: prime implicants
- *FnamePRIMES* (str): name of *json* file

example:

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

4.1.4 read_primes

read_primes(*FnamePRIMES*)

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

arguments:

- *FnamePRIMES* (str): name of *json* file

returns:

- *Primes*: prime implicants

example:

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

4.1.5 primes2genysis

primes2genysis(*Primes*, *FnameGENYSIS*)

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
- *FnameGENYSIS* (str): name of GenYsis file

example:

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

4.1.6 primes2bns

primes2bns(*Primes*, *FnameBNS*)

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
- *FnameBNS* (str): name of *bns* file

example:

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

4.2 PrimeImplicants

4.2.1 are_equal

are_equal(*Primes1*, *Primes2*)

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* (bool): whether *Primes1=Primes2*

example:

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

4.2.2 find_inputs

find_inputs(*Primes*)

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

arguments:

- *Primes*: prime implicants

returns:

- *Names* (list): the names of the inputs

example:

```
>>> inputs = find_inputs(primes)
>>> print(inputs)
['DNA_damage','EGFR','FGFR3']
```

4.2.3 find_outputs

find_outputs(*Primes*)
Finds all outputs in the network defined by *Primes*.

arguments:

- *Primes*: prime implicants

returns:

- *Names* (list): the names of the outputs

example:

```
>>> inputs = find_inputs(primes)
>>> print(inputs)
['Proliferation','Apoptosis','GrowthArrest']
```

4.2.4 find_constants

find_constants(*Primes*)
Finds all constants in the network defined by *Primes*.

arguments:

- *Primes*: prime implicants

returns:

- *Activities* (dict): the names and activities of constants

example:

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

4.2.5 copy

copy(*Primes*)
Creates a copy of *Primes*.

arguments:

- *Primes*: prime implicants

returns:

- *PrimesNew* (dict): a copy of *Primes*

example:

```
>>> primes_new = copy(primes)
```

4.2.6 create_constants

create_constants(*Primes*, *Activities*)
Overwrites *Primes* for every name, value pair in *Activities* such that name is constant at value. For safety it is required that the names of *Activities* exist in *Primes*.

arguments:

- *Primes*: prime implicants
- *Activities* (dict): a subspace

example:

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

4.2.7 create_blinkers

`create_blinkers`(*Primes*, *Names*)

Overwrite *Primes* such that every name in *Names* becomes a so-called blinker, i.e., a variable that is only regulated by itself, like an input, but the self-regulation is negative. Blinkers can therefore change their activity in every state of the transition system. For safety it is required that the names of *Activities* exist in *Primes*.

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 - usually too many to list. 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 = TS.trap_spaces(primes, "min")
```

arguments:

- *Primes*: prime implicants
- *Names* (list): variables to become blinkers

example:

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

4.2.8 create_inputs

`create_inputs`(*Primes*, *Names*)

Overwrite *Primes* such that each name in *Names* becomes an input. For safety it is required that the names of *Activities* exist in *Primes*.

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* (list): variables to become constants

example:

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

4.2.9 remove_variables

remove_variables(*Primes*, *Names*)

Removes all variables contained in *Names* from *Primes*. For safety it is required that the names of *Activities* exist in *Primes*.

arguments:

- *Primes*: prime implicants
- *Names* (list): the names of variables to remove

example:

```
>>> names = ["PKC","GADD45","ELK1","FOS"]
>>> remove_variables(primes, names)
```

4.2.10 percolate_constants

percolate_constants(*Primes*, *RemoveConstants=False*)

Percolates the values of constants, see *Klarner2015(b)* Sec. 3.1 for a formal definition. Use *RemoveConstants* to determine whether constants should be kept in the remaining network or whether you want to remove them.

arguments:

- *Primes*: prime implicants
- *RemoveConstants* (bool): whether constants should be kept

returns:

- *Names* (list): names of variables that are constant due to the percolation

example:

```
>>> percolate_constants(primes)
>>> constants = percolate_constants(primes, True)
>>> constants
['Erk', 'Mapk', 'Raf']
```

4.3 InteractionGraphs

4.3.1 primes2igraph

primes2igraph (*Primes*)

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* (networkx.DiGraph): 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.edge["v1"]["v3"]["sign"]
set([1, -1])
```

4.3.2 copy

copy (*IGraph*)

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

arguments:

- *IGraph*: interaction graph

returns:

- *IGraph2*: new interaction graph

example:

```
>>> igraph2 = copy(igraph)
```

4.3.3 igraph2dot

igraph2dot (*IGraph*, *FnameDOT=None*)

Generates a *dot* file from *IGraph* and saves it as *FnameDOT* or returns it as a string.

arguments:

- *IGraph*: interaction graph
- *FnameDOT* (str): name of *dot* file or *None*

returns:

- *FileDOT* (str): file as string if not *FnameDOT==None*, otherwise it returns *None*

example:

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

4.3.4 igraph2image

igraph2image(*IGraph*, *FnameIMAGE*, *Silent=False*)

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

arguments:

- *IGraph*: interaction graph
- *FnameIMAGE* (str): name of image
- *Silent* (bool): disables print statements

example:

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

4.3.5 add_style_interactionsigns

add_style_interactionsigns(*IGraph*)

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)
```

4.3.6 add_style_activities

add_style_activities(*IGraph*, *Activities*)

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* (dict): activated and inhibited nodes

example:

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

4.3.7 add_style_inputs

add_style_inputs(*IGraph*)

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)
```

4.3.8 add_style_outputs

add_style_outputs(*IGraph*)

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)
```

4.3.9 add_style_constants

add_style_constants(*IGraph*)

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

arguments:

- *IGraph*: interaction graph

example:

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

4.3.10 add_style_sccs

add_style_sccs(*IGraph*)

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)
```

4.3.11 add_style_condensation

add_style_condensation(*IGraph*)

Adds a separate graph to *IGraph* that depicts the *condensation graph*, a map of how the SCCs regulate each other. A node in the condensation graph indicates how many variables are contained in the respective SCC. If the SCC contains a single variable then its name is displayed.

arguments:

- *IGraph*: interaction graph

example:

```
>>> add_style_condensation(igraph)
```

4.3.12 add_style_path

add_style_path(*IGraph*, *Path*, *Color*)

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

arguments:

- *IGraph*: interaction graph
- *Path* (list): sequence of component names
- *Color* (str): color of the path

example:

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

4.3.13 add_style_subgraphs

add_style_subgraphs(*IGraph*, *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. To add custom labels or fillcolors to a subgraph supply a tuple consisting of the list of nodes and a dictionary of subgraph attributes.

Note: *Subgraphs* must satisfy the 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* (list): lists of nodes *or* pairs of lists and subgraph attributes

example:

```
>>> sub1 = ["v1","v2"]
>>> sub2 = ["v3","v4"]
>>> subgraphs = [sub1,sub2]
>>> add_style_subgraphs(igraph, subgraphs)

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

4.3.14 add_style_default

add_style_default(*IGraph*)

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

arguments:

- *IGraph*: interaction graph

example:

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

4.3.15 activities2animation

activities2animation(*IGraph*, *Activities*, *FnameGIF*, *FnameTMP='tmp*.jpg'*, *Delay=50*, *Loop=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* (list): sequence of activities
- *Delay* (int): number of 1/100s between each frame
- *Loop* (int): number of repetitions, use 0 for infinite
- *FnameTMP* (str): name for temporary image files, use "*" to indicate counter
- *FnameGIF* (str): name of the output *gif* file

example:

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

4.4 StateTransitionGraphs

4.4.1 copy

copy(*STG*)

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

arguments:

- *STG*: state transition graph

returns:

- *STG2*: new state transition graph

example:

```
>>> stg2 = copy(stg)
```

4.4.2 primes2stg

primes2stg(*Primes*, *Update*, *InitialStates=<function <lambda> at 0x4331672c>*)

Creates the state transition graph (STG) of a network defined by *Primes* and *Update*. The *InitialStates* 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).

The STG is constructed by a depth first search (DFS) starting from the given initial states. The default for *InitialStates* is `lambda x: True`, i.e., every state is initial. For a single initial state, say *"100"* use *InitialStates="100"*, for a set of initial states use *InitialStates=["100", "101"]* and for a initial subspace use *InitialStates="1–"* or the *dict* representation of subspaces.

arguments:

- *Primes*: prime implicants
- *Update* (str): either *"asynchronous"* or *"synchronous"*
- *InitialStates* (func / str / dict / list): a function, a subspace, a state or a list of states

returns:

- *STG* (networkx.DiGraph): 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)
```

4.4.3 stg2dot

stg2dot (*STG*, *FnameDOT=None*)

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
- *FnameDOT* (str): name of *dot* file or *None*

returns:

- *FileDOT* (str): file as string if not *FnameDOT==None*, otherwise it returns *None*

example:

```
>>> stg = primes2stg(primes, update, init)
>>> igraph.graph["label"] = "IRMA Network - State Transition Graph"
>>> igraph.graph["node"] = {"style":"filled", "color":"red"}
>>> igraph.graph["edge"] = {"arrowsize": 2.0}
>>> igraph.node["001000"]["fontsize"] = 20
>>> igraph.edge["001110"]["001010"]["style"] = "dotted"
>>> stg2image( igraph, "irma_stg.pdf")
```

4.4.4 stg2image

stg2image (*STG*, *FnameIMAGE*, *Silent=False*)

Creates an image file from a state transition graph using *Graphviz* and the layout engine *dot*. Use `dot -T?` to find out which output formats are supported on your installation.

arguments:

- *STG*: state transition graph
- *FnameIMAGE* (str): name of output file
- *Silent* (bool): disables print statements

example:

```
>>> stg2image(stg, "mapk_stg.pdf")
>>> stg2image(stg, "mapk_stg.jpg")
>>> stg2image(stg, "mapk_svg.pdf")
```

4.4.5 add_style_tendencies

add_style_tendencies (*STG*)

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)
```

4.4.6 add_style_sccs

add_style_sccs(*STG*)

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)
```

4.4.7 add_style_subspaces

add_style_subspaces(*Primes*, *STG*, *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 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): 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)
```

4.4.8 add_style_mintrapspaces

add_style_mintrapspaces(*Primes*, *STG*, *MaxOutput=100*)

A convenience function that combines *add_style_subspaces* and *TrapSpaces.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* (int): maximal number of minimal trap spaces, see *trap_spaces*

example:

```
>>> add_style_mintrapspaces(primes, stg)
```

4.4.9 add_style_condensation

add_style_condensation(*STG*)

Adds a separate graph to *STG* that depicts the *condensation graph*, a map of how the SCCs regulate each other. A node in the condensation graph indicates how many states are contained in the respective SCC. If the SCC contains a single state then its name is displayed.

arguments:

- *STG*: state transition graph

example:

```
>>> add_style_condensation(stg)
```

4.4.10 add_style_path

add_style_path(*STG*, *Path*, *Color*, *Penwidth=3*)

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

arguments:

- *STG*: state transition graph
- *Path* (list): state dictionaries or state strings
- *Color* (str): color of the path
- *Penwidth* (int): width of nodes and edges involved in *Path* in pt

example:

```
>>> path = ["001", "011", "101"]
>>> add_style_path(stg, path, "red")
```

4.4.11 add_style_default

add_style_default(*Primes*, *STG*)

A convenience function that adds styles for tendencies, SCCs, minimal trap spaces and the condensation graph.

arguments:

- *Primes*: primes implicants
- *STG*: state transition graph

example:

```
>>> add_style_default(stg)
```

4.4.12 successor_synchronous

successor_synchronous(*Primes*, *State*)

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* (str / dict): a state

returns:

- *Successor* (dict): the synchronous successor of *State*

example:

```
>>> state = "100"
>>> successor_synchronous(primes, state)
{'v1':0, 'v2':1, 'v3':1}
```

4.4.13 successors_asynchronous

successors_asynchronous(*Primes*, *State*)

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* (str / dict): a state

returns:

- *Successors* (list): the asynchronous successors of *State*

example:

```
>>> state = "100"
>>> successors_asynchronous(primes, state)
[{'v1':1, 'v2':1, 'v3':1},{'v1':0, 'v2':0, 'v3':1},{'v1':0, 'v2':1, 'v3':0}]
```

4.4.14 random_successor_mixed

random_successor_mixed(*Primes*, *State*)

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* (str / dict): a state

returns:

- *Successor* (dict): a random successor of *State* using the mixed update

example:

```
>>> state = "100"
>>> random_successor_mixed(primes, state)
{'v1':1, 'v2':1, 'v3':1}
```

4.4.15 random_state

random_state(*Primes*, *Subspace={}*)

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* (str / dict): a subspace

returns:

- *State* (dict): 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}
```

4.4.16 random_walk

random_walk(*Primes*, *Update*, *InitialState*, *Length*)

Returns a random walk of *Length* many states in the transition system defined by *Primes* and *Update* starting from a state defined by *InitialState*. If *InitialState* is a subspace then *random_state* will be used to draw a random state from inside it.

arguments:

- *Primes*: prime implicants
- *Update* (str): the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *InitialState* (str / dict): an initial state or subspace
- *Length* (int): length of the random walk

returns:

- *Path* (list): sequence of states

example:

```
>>> path = random_walk(primes, "asynchronous", "11---0", 4)
```

4.4.17 best_first_reachability

best_first_reachability(*Primes*, *InitialSpace*, *GoalSpace*, *Memory=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
- *InitialSpace* (str / dict): initial subspace
- *GoalSpace* (str / dict): goal subspace
- *Memory* (int): maximal number of states memorized before search is stopped

returns:

- *Path* (list): 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
```

4.4.18 state2str

state2str(*State*)

Converts the dictionary representation of a state into the string representation of a state.

arguments

- *State* (dict): dictionary representation of state

returns

- *State* (str): string representation of state

example:

```
>>> state = {"v2":0, "v1":1, "v3":1}
>>> state2str(primes, state)
'101'
```

4.4.19 str2state

str2state (*Primes*, *State*)

Converts the string representation of a state into the dictionary representation of a state.

arguments

- *Primes*: prime implicants or a list of names
- *State* (str): string representation of state

returns

- *State* (dict): dictionary representation of state

example:

```
>>> state = "101"
>>> str2state(primes, state)
{'v2':0, 'v1':1, 'v3':1}
```

4.4.20 subspace2str

subspace2str (*Primes*, *Subspace*)

Converts the dictionary representation of a subspace into the string representation of a subspace. Uses "-" to indicate free variables.

arguments

- *Primes*: prime implicants or a list of names
- *Subspace* (dict): a subspace

returns

- *Subspace* (str): the string representation of *Subspace*

example:

```
>>> sub = {"v2":0, "v3":1}
>>> subspace2str(primes, sub)
'-01'
```

4.4.21 str2subspace

str2subspace (*Primes*, *Str*)

Converts the string representation of a subspace into the dictionary representation of a subspace. Use "-" to indicate free variables.

arguments

- *Primes*: prime implicants or a list of names
- *Subspace* (str): a subspace

returns

- *Subspace* (dict): the dictionary representation of subspace

example:

```
>>> sub = "-01"
>>> str2subspace(primes, sub)
{'v2':0, 'v3':1}
```

4.4.22 subspace2states

subspace2states(*Primes*, *Subspace*)

Generates all states contained in *Subspace*.

arguments:

- *Primes*: prime implicants or a list of names
- *Subspace* (str or dict): a subspace

returns:

- *States* (list): the states contained in *Subspace*

example:

```
>>> subspace = "1-1"
>>> subspace2states(primes,subspace)
[{'v1':1,'v2':0,'v3':1},{'v1':1,'v2':1,'v3':1}]
```

4.4.23 hamming_distance

hamming_distance(*Subspace1*, *Subspace2*)

Returns the Hamming distance between to subspaces. Variables that are free in either subspace are ignored.

arguments:

- *Subspace1, Subspace2* (dict): subspaces in dictionary representation

returns:

- *Distance* (int): 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
```

4.5 ModelChecking

4.5.1 check_primes

check_primes(*Primes*, *Update*, *InitialStates*, *Specification*, *DisableCounterExamples=True*, *DynamicReorder=True*, *DisableReachableStates=False*, *ConeOfInfluence=True*)

Calls *NuSMV* to check whether the *Specification* is true or false in the transition system defined by *Primes*, the *InitialStates* 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*.

Note: If *DisableCounterExamples* is *False* then *ConeOfInfluence* is forced to *False* because otherwise the counterexample output is incomplete.

arguments:

- *Primes*: prime implicants
- *Update* (str): the update strategy, either *"synchronous"*, *"asynchronous"* or *"mixed"*
- *InitialStates* (str): a *NuSMV* expression for the initial states, including the keyword *INIT*
- *Specification* (str): a *NuSMV* formula, including the keyword *LTLSPEC* or *CTLSPEC*
- *DisableCounterexamples* (bool): disables the computation of counterexamples using *-dcx*
- *DynamicReorderBDDs* (bool): enables dynamic reordering of variables using *-dynamic*
- *DisableReachableStates* (bool): disables the computation of reachable states using *-df*
- *ConeOfInfluence* (bool): enables cone of influence reduction using *-coi*

returns:

- *Answer* (bool): result of query, if *DisableCounterexamples==True*
- *(Answer, Counterexample)* (bool, tuple/None): result of query and counterexample, if *DisableCounterexamples==False*. If *Answer==True* then *CounterExample* will be assigned *None*.

example:

```
>>> init = "INIT TRUE"
>>> update = "asynchronous"
>>> spec = "CTLSPEC AF(EG(v1&!v2))"
>>> answer = check_primes(primes, update, init, spec)
>>> answer
False
>>> answer, counterex = check_primes(primes, update, init, spec, False)
>>> counterex
 ({'v1':0,'v2':0},{'v1':1,'v2':0},{'v1':1,'v2':1})
```

4.5.2 check_smv

check_smv(*FnameSMV*, *DisableCounterExamples=False*, *DynamicReorder=True*, *DisableReachableStates=False*, *ConeOfInfluence=True*)

Calls *NuSMV* with the query defined in the *smv* file *FnameSMV*. 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*.

Note: It is currently required that *FnameSMV* contains a single LTL or CTL formula. For future versions it is planned that *check_smv* returns a dictionary of answers.

Note: If *DisableCounterExamples* is *False* then *ConeOfInfluence* is forced to *False* because otherwise the counterexample output is incomplete.

arguments:

- *FnameSMV*: name of smv file

- *DisableCounterexamples* (bool): disables computation of counterexamples (*-dcx*)
- *DynamicReorderBDDs* (bool): enables dynamic reordering of variables (*-dynamic*)
- *DisableReachableStates* (bool): disables the computation of reachable states (*-df*)
- *ConeOfInfluence* (bool): enables cone of influence reduction (*-coi*)

returns:

- *Answer* (bool): result of query, if *DisableCounterexamples==True*
- *(Answer, Counterexample)* (bool, tuple/None): result of query and counterexample, if *DisableCounterexamples==False*. If *Answer==True* then *CounterExample* will be assigned *None*.

example:

```
>>> check_smv("mapk.smv")
False
>>> answer, counterex = check_smv("mapk.smv", False)
>>> counterex
({'Erk':0,'Mek':0},{'Erk':1,'Mek':0},{'Erk':1,'Mek':1})
```

4.5.3 primes2smv

primes2smv(*Primes*, *Update*, *InitialStates*, *Specification*, *FnameSMV=None*)

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.

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* (str): the update strategy, either *"synchronous"*, *"asynchronous"* or *"mixed"*
- *InitialStates* (str): a *NuSMV* expression for the initial states, including the keyword *INIT*
- *Specification* (str): a *NuSMV* formula, including the keyword *LTLSPEC* or *CTLSPEC*
- *FnameSMV* (str): name for *smv* file or *None*

returns:

- *FileSMV* (str): file as string or *None* if *FnameSMV==None*
- raises *Exception* if *Primes* is the empty dictionary

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)
```

4.6 TrapSpaces

4.6.1 trap_spaces

trap_spaces(*Primes*, *Type*, *MaxOutput=100*, *FnameASP=None*)

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"* or *"all"* and specifies whether subset minimal, subset maximal or all trap spaces should be returned.

Warning: The number of trap spaces is easily exponential in the number of components. Use the safety parameter *MaxOutput* to control the number of returned solutions.

To create the *asp* file for inspection or manual editing, pass a file name to *FnameASP*.

arguments:

- *Primes*: prime implicants
- *Type* (str): either *"max"*, *"min"* or *"all"*
- *MaxOutput* (int): maximal number of trap spaces to return
- *FnameASP* (str): name of *asp* file to create, or *None*

returns:

- *Subspaces* (list): the trap spaces

example:

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

4.6.2 steady_states

steady_states(*Primes*, *MaxOutput=100*, *FnameASP=None*)

Wrapper function for *trap_spaces_bounded* that sets the bounds to *n,n* to return steady states.

arguments:

- *Primes*: prime implicants

- *MaxOutput* (int): maximal number of trap spaces to return
- *FnameASP*: file name or *None*

returns:

- *States* (list): the steady states

example:

```
steady = steady_states(primes)
print len(steady)
>>> 2
```

4.6.3 steady_states_projected

steady_states_projected(*Primes*, *Project*, *Aggregate=False*, *MaxOutput=100*, *FnameASP=None*)

Returns a list of projected steady states using the Potassco ASP solver *[Gebser2011]*. This function works like *trap_spaces_projected* but enforces that the returned elements are steady states.

arguments:

- *Primes*: prime implicants
- *Project*: list of names
- *Aggregate*: count number of steady states per projection
- *MaxOutput* (int): maximal number of trap spaces to return
- *FnameASP*: file name or *None*

returns:

- *Activities* (list): projected steady states

example:

```
psteady = steady_states_projected(primes, ["v1","v2"])
print len(psteady)
>>> 2
print psteady
>>> [{"v1":1,"v2":0},{"v1":0,"v2":0}]
```

4.6.4 primes2asp

primes2asp(*Primes*, *FnameASP*, *Bounds*, *Project*, *InsideOf*, *OutsideOf*)

Saves Primes as an *asp* file in the Potassco format intended for computing minimal and maximal trap spaces. The homepage of the Potassco solving collection is http://potassco.sourceforge.net. The *asp* file consists of data, the hyperarcs of the prime implicant graph, and a problem description that includes the consistency, stability and non-emptiness conditions.

There are four additional parameters that modify the problem:

Bounds must be either a tuple of integers *(a,b)* or *None*. A tuple *(a,b)* uses Potassco's cardinality constraints to enforce that the number of fixed variables *x* of a trap space satisfies *a<=x<=b*. *None* results in no bounds.

Project must be either a list of names or *None*. A list of names projects the solutions onto these variables using the meta command "#show" and the clasp parameter "–project". Variables of *Project* that do not appear in *Primes* are ignored. *None* results in no projection.

InsideOf must be a subspace (dict) that specifies that only trap spaces that are contained in it are wanted.

OutsideOf must be a subspace (dict) that specifies that only trap spaces that contain it are wanted.

arguments:

- *Primes*: prime implicants
- *FnameASP*: name of *ASP* file or None
- *Bounds* (tuple): cardinality constraint for the number of fixed variables
- *Project* (list): names to project to or *None* for no projection
- *InsideOf* (dict): a subspace or *None*
- *OutsideOf* (dict): a subspace or *None*

returns:

- *FileASP* (str): file as string if not *FnameASP==None* and *None* otherwise

example:

```
>>> primes2asp(primes, "mapk.asp", False, False)
>>> primes2asp(primes, "mapk_bounded.asp", (20,30), False)
>>> primes2asp(primes, "mapk_projected.asp", False, ['AKT','GADD45','FOS','SMAD'])
```

4.7 AttractorDetection

4.7.1 compute_attractors_tarjan

compute_attractors_tarjan(*Primes*, *STG*)

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 whther a SCC is reachable from another. Returns the attractors, i.e., those SCCs from which no other SCC is reachable.

arguments:

- *Primes*: prime implicants
- *STG*: state transition graph

returns:

- *Attractors* (list of tuples of states): the attractors

example:

```
>>> bnet = ["x, !x&y | z",
...         "y, !x | !z",
...         "z, x&!y"]
>>> bnet = "\n".join(bnet)
>>> primes = FEX.bnet2primes(bnet)
>>> stg = STGs.primes2stg(primes, "asynchronous")
>>> attractors = STGs.compute_attractors_tarjan(stg)
>>> for A in attractors:
...     print A
[{'v1': 1, 'v2': 0, 'v3': 1}]
[{'v1': 0, 'v2': 1, 'v3': 0}, {'v1': 1, 'v2': 1, 'v3': 0}]
```

4.7.2 find_attractor_by_randomwalk_and_ctl

`find_attractor_by_randomwalk_and_ctl`(*Primes*, *Update*, *InitialState={}*, *Length=0*, *Attempts=10*)

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* many states, starting from a state defined by *InitialState*. If *InitialState* 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., *ModelChecking.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 converted:

```
Length = 10*len(Primes)
```

which proved sufficient in out computer experiments.

arguments:

- *Primes*: prime implicants
- *Update* (str): the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *InitialState* (str / dict): an initial state or subspace
- *Length* (int): length of random walk
- *Attempts* (int): number of attempts before exception is raised

returns:

- *State* (dict): a state that belongs to some attractor
- raises *Exception* if no attractor state is found

example:

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

4.7.3 completeness_naive

`completeness_naive`(*Primes*, *Update*, *Trapspaces*)

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 the query checks whether each attractor intersects one of the subspaces.

---

arguments:

- *Primes*: prime implicants
- *Update* (str): the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *Trapspaces* (list): list of subspaces in string or dict representation

returns:

- *Answer* (bool): whether *Subspaces* is complete in the STG defined by *Primes* and *Update*,
- *CounterExample* (dict): a state from which none of the *Subspaces* is reachable (if *Answer==False*)

example:

```
>>> mintspaces = TrapSpaces.trap_spaces(primes, "min")
>>> answer, counterex = completeness_naive(primes, "asynchronous", mintspaces)
>>> answer
True
```

4.7.4 completeness_iterative

completeness_iterative(*Primes*, *Update*)

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* (str): the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*

returns:

- *Answer* (bool): whether *Subspaces* is complete in the STG defined by *Primes* and *Update*,
- *Counterexample* (dict): a state from which no subspace is reachable, if *Answer==False*

example:

```
>>> answer, counterex = completeness_iterative( primes, "asynchronous" )
>>> answer
False
>>> STGs.state2str(counterex)
10010111101010100001100001011011111111
```

4.7.5 univocal

univocal (*Primes*, *Update*, *Trapspace*)

The model checking and random-walk-based method for deciding whether *Trapspace* 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 *Trapspace* 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 *Trapspace*.

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: *Trapspace* 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* (str): the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *Trapspace* (str / dict): a subspace

returns:

- *Answer* (bool): whether *Trapspace* is univocal in the STG defined by *Primes* and *Update*
- *AttractorState* (dict): a state that belongs to an attractor
- *CounterExample* (dict): a state that belongs to another attractor (if *Answer=False*)

example:

```
>>> mintspaces = TrapSpaces.trap_spaces(primes, 'min', None, None, 1000, None)
>>> trapspace = mintrapspaces[0]
>>> answer, state, counterex = univocal(primes, trapspace, "asynchronous")
>>> answer
True
```

4.7.6 faithful

faithful(*Primes*, *Update*, *Trapspace*)

The model checking approach for deciding whether *Trapspace* 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*.

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: *Trapspace* 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* (str): the update strategy, one of *"asynchronous"*, *"synchronous"*, *"mixed"*
- *Trapspace* (str / dict): a subspace

returns:

- *Answer* (bool): whether *Trapspace* is faithful in the STG defined by *Primes* and *Update*
- *CounterExample* (dict): a state that belongs to an attractor that does not oscillate in all free variables (if *Answer=False*)

example:

```
>>> mintspaces = TrapSpaces.trap_spaces(primes, "min")
>>> answer, counterex = faithful(primes, mintspaces[0])
>>> answer
True
```

4.8 TemporalQueries

4.8.1 AGEF_oneof_subspaces

AGEF_oneof_subspaces(*Primes*, *Subspaces*)

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* (str): the CTL formula

example:

```
>>> subspaces = [{"v1":0,"v2":0},{"v2":1}]
>>> AGEF_oscillation(subspaces)
'AG(EF(!v1&!v2 | v2))'
```

4.8.2 EF_oneof_subspaces

EF_oneof_subspaces(*Primes*, *Subspaces*)

Constructs a CTL formula that queries whether there is a path that leads to one of the Subspaces.

arguments:

- *Subspaces* (list): a list of subspaces

returns:

- *Formula* (str): the CTL formula

example:

```
>>> subspaces = [{"v1":0,"v2":0}, "1-1--"]
>>> EF_oneof_subspaces(subspaces)
'EF(!v1&!v2 | v1&v3)'
```

4.8.3 EF_unsteady_states

EF_unsteady_states(*Names*)

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* (list): a list of names of variables

returns:

•*Formula* (str): the CTL formula

example:

```
>>> names = ["v1","v2"]
>>> EF_unsteady_states(names)
'EF(v1_steady!=0) & EF(v2_steady!=0))'
```

4.8.4 subspace2proposition

subspace2proposition(*Primes*, *Subspace*)

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* (str / dict): a subspace in string or dictionary representation

returns:

- *Proposition* (str): the proposition

example:

```
>>> subspace = {"v1":0,"v2":1}
>>> init = "INIT " + subspace2proposition(subspace)
>>> init
'INIT v1&!v2'
```

4.9 QuineMcCluskey

4.9.1 functions2mindnf

functions2mindnf(*BooleanFunctions*)

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

arguments:

- *BooleanFunctions* (dict): keys are component names and values are Boolean functions

returns:

- *MinDNF* (dict): 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)
```

4.9.2 functions2primes

functions2primes(*BooleanFunctions*)

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

arguments:

- *BooleanFunctions* (dict): 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)
```

4.9.3 primes2mindnf

primes2mindnf(*Primes*)

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:

- *MinDNF* (dict): keys are names and values are minimal DNF expressions

example:

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

CHAPTER

FIVE

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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.

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.