libsbgn-python Documentation

Release 0.2.2

Matthias König

Nov 01, 2020

CONTENTS

Source code and support are available at https://github.com/matthiaskoenig/libsbgn-python.

CHAPTER

ONE

INTRODUCTION

1.1 Overview

Python library to work with SBGN. This library is based on the SBGN XML schema and supports reading, writing and validation of SBGN files. Python 3 is supported. The initial library was generated using generateDS. Additional utility functions for reading, writing and rendering SBGN documents are provided.

To cite libsbgnpy use the following BibTex or equivalent:

```
@software{libsbgnpy,
    author = {Matthias König},
    title = {{matthiaskoenig/libsbgn-python: libsbgn-python-v0.2.0}},
    month = mar,
    year = 2020,
    publisher = {Zenodo},
    version = {v0.2.0},
    doi = {10.5281/zenodo.3712285},
    url = {https://doi.org/10.5281/zenodo.3712285}
}
```

Source code is available from https://github.com/matthiaskoenig/libsbgn-python.

To report bugs, request features or asking questions please file an issue.

1.2 Installation

The libsbgn-python package is available from pypi and can be installed via:

```
pip install libsbgnpy
```

CHAPTER

TWO

EXAMPLES

In this section example use cases of `libsbgnpy` are provided.

2.1 Create SBGN

The example demonstrates how to create a `SBGN` document from scratch. This includes creation of the following objects.

- `map`
- `bbox`: bounding boxes
- `glyph`
- `arc`

The full map consists of a simple reaction catalysed via `ADH1` which converts `Ethanol + NAD+ --> Ethanal + NADH + H+`.

[1]:
```
from IPython.display import Image
import tempfile

import IPython
from IPython.core.display import HTML
from pygments import highlight
from pygments.lexers import PythonLexer
from pygments.formatters import HtmlFormatter

def pprint_xml(xml_str):
    """ Helper function for highlighted xml. """
    IPython.display.display(HTML('<style type="text/css">{}</style>{}'.format(
        HtmlFormatter().get_style_defs('.highlight'),
        highlight(xml_str, PythonLexer(), HtmlFormatter()))))
```

[2]:
```
# import libsbgn and important SBGN types
import libsbgnpy.libsbgn as libsbgn
from libsbgnpy.libsbgnTypes import Language, GlyphClass, ArcClass, Orientation

# create empty sbgn
sbgn = libsbgn.sbgn()

# create map, set language and set in sbgn
map = libsbgn.map()
map.set_language(Language.PD)
```

(continues on next page)

(continued from previous page)

```
sbgn.set_map(map)

# create a bounding box for the map
box = libsbgn.bbox(x=0, y=0, w=363, h=253)
map.set_bbox(box)

# create some glyphs
# class attribute is named 'class_' ! in glyphs and arcs
'''
    <glyph class="simple chemical" id="glyph1">
            <label text="Ethanol"/> <!-- fontsize="" etc -->
            <!-- Line breaks are allowed in the text attribute -->
            <bbox x="40" y="120" w="60" h="60"/>
    </glyph>
'''
# glyphs with labels
g = libsbgn.glyph(class_=GlyphClass.SIMPLE_CHEMICAL, id='glyph1')
g.set_label(libsbgn.label(text='Ethanol'))
g.set_bbox(libsbgn.bbox(x=40, y=120, w=60, h=60))
map.add_glyph(g)

g = libsbgn.glyph(class_=GlyphClass.SIMPLE_CHEMICAL, id='glyph_ethanal')
g.set_label(libsbgn.label(text='Ethanal'))
g.set_bbox(libsbgn.bbox(x=220, y=110, w=60, h=60))
map.add_glyph(g)

g = libsbgn.glyph(class_=GlyphClass.MACROMOLECULE, id='glyph_adh1')
g.set_label(libsbgn.label(text='ADH1'))
g.set_bbox(libsbgn.bbox(x=106, y=20, w=108, h=60))
map.add_glyph(g)

g = libsbgn.glyph(class_=GlyphClass.SIMPLE_CHEMICAL, id='glyph_h')
g.set_label(libsbgn.label(text='H+'))
g.set_bbox(libsbgn.bbox(x=220, y=190, w=60, h=60))
map.add_glyph(g)

g = libsbgn.glyph(class_=GlyphClass.SIMPLE_CHEMICAL, id='glyph_nad')
g.set_label(libsbgn.label(text='NAD+'))
g.set_bbox(libsbgn.bbox(x=40, y=190, w=60, h=60))
map.add_glyph(g)

g = libsbgn.glyph(class_=GlyphClass.SIMPLE_CHEMICAL, id='glyph_nadh')
g.set_label(libsbgn.label(text='NADH'))
g.set_bbox(libsbgn.bbox(x=300, y=150, w=60, h=60))
map.add_glyph(g)

# glyph with ports (process)
g = libsbgn.glyph(class_=GlyphClass.PROCESS, id='pn1',
                  orientation=Orientation.HORIZONTAL)
g.set_bbox(libsbgn.bbox(x=148, y=168, w=24, h=24))
g.add_port(libsbgn.port(x=136, y=180, id="pn1.1"))
g.add_port(libsbgn.port(x=184, y=180, id="pn1.2"))
map.add_glyph(g)

# arcs
# create arcs and set the start and end points
a = libsbgn.arc(class_=ArcClass.CONSUMPTION, source="glyph1", target="pn1.1", id="a01
↪")
```

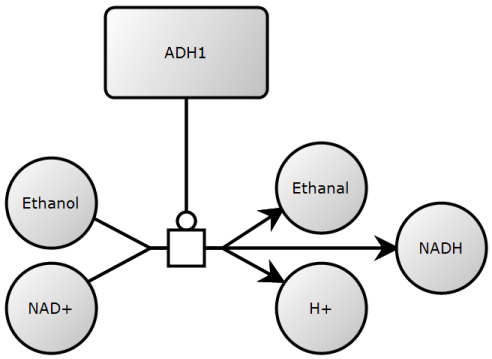
(continues on next page)

(continued from previous page)

```
a.set_start(libsbgn.startType(x=98, y=160))
a.set_end(libsbgn.endType(x=136, y=180))
map.add_arc(a)

a = libsbgn.arc(class_=ArcClass.PRODUCTION, source="pn1.2", target="glyph_nadh", id=
↪"a02")
a.set_start(libsbgn.startType(x=184, y=180))
a.set_end(libsbgn.endType(x=300, y=180))
map.add_arc(a)

a = libsbgn.arc(class_=ArcClass.CATALYSIS, source="glyph_adh1", target="pn1", id="a03
↪")
a.set_start(libsbgn.startType(x=160, y=80))
a.set_end(libsbgn.endType(x=160, y=168))
map.add_arc(a)

a = libsbgn.arc(class_=ArcClass.PRODUCTION, source="pn1.2", target="glyph_h", id="a04
↪")
a.set_start(libsbgn.startType(x=184, y=180))
a.set_end(libsbgn.endType(x=224, y=202))
map.add_arc(a)

a = libsbgn.arc(class_=ArcClass.PRODUCTION, source="pn1.2", target="glyph_ethanal",␣
↪id="a05")
a.set_start(libsbgn.startType(x=184, y=180))
a.set_end(libsbgn.endType(x=224, y=154))
map.add_arc(a)

a = libsbgn.arc(class_=ArcClass.CONSUMPTION, source="glyph_nad", target="pn1.1", id=
↪"a06")
a.set_start(libsbgn.startType(x=95, y=202))
a.set_end(libsbgn.endType(x=136, y=180))
map.add_arc(a)

# write SBGN to file
f_out = tempfile.NamedTemporaryFile(suffix=".sbgn")
sbgn.write_file(f_out.name)

# render SBGN
from libsbgnpy import render
f_png = tempfile.NamedTemporaryFile(suffix=".png")
render.render_sbgn(sbgn, image_file=f_png.name, file_format="png")
Image(f_png.name, width=300)
```

```
SBGN rendered: /tmp/tmpwf2646f7.png
```

[2]:



2.2 Iterate Arcs & Glyphs

Example for iterating over glyphs and arcs in given SBGN file

[3]:

```
import libsbgnpy.libsbgn as libsbgn
from libsbgnpy import utils

# sbgn and map
sbgn = utils.read_from_file("adh.sbgn")
map = sbgn.get_map()
print('Language:', map.get_language(), "\n")

# glyphs
'''
<glyph class="simple chemical" id="glyph1">
        <label text="Ethanol"/> <!-- fontsize="" etc -->
        <!-- Line breaks are allowed in the text attribute -->
        <bbox x="40" y="120" w="60" h="60"/>
    </glyph>
'''
glyphs = map.get_glyph()
for g in glyphs:
    cls = g.get_class()
    print(cls, g.get_id())
    label = g.get_label()
    if cls == 'simple chemical':
        print('label: ', label.get_text())

    if cls == 'process':
        for p in g.get_port():
            print('port ', p.get_id(), p.get_x(), p.get_y())

    box = g.get_bbox()
    utils.print_bbox(box)
    print()

# arcs
'''
    <arc class="consumption" source="glyph_nad" target="pn1.1" id="a06">
        <start x="95" y="202" />
```

(continues on next page)

(continued from previous page)

```
        <end x="136" y="180" />
    </arc>
'''
arcs = map.get_arc()
for a in arcs:
    print(a.get_class(), a.get_source(), a.get_target(), a.get_id())
    start = a.get_start()
    print(start.x, start.y)
    end = a.get_end()
    print(end.x, end.y)
```

```
Language: Language.PD

GlyphClass.SIMPLE_CHEMICAL glyph1
x, y, w, h :  40.0 120.0 60.0 60.0

GlyphClass.SIMPLE_CHEMICAL glyph_ethanal
x, y, w, h :  220.0 110.0 60.0 60.0

GlyphClass.MACROMOLECULE glyph_adh1
x, y, w, h :  106.0 20.0 108.0 60.0

GlyphClass.SIMPLE_CHEMICAL glyph_h
x, y, w, h :  220.0 190.0 60.0 60.0

GlyphClass.SIMPLE_CHEMICAL glyph_nad
x, y, w, h :  40.0 190.0 60.0 60.0

GlyphClass.SIMPLE_CHEMICAL glyph_nadh
x, y, w, h :  300.0 150.0 60.0 60.0

GlyphClass.PROCESS pn1
x, y, w, h :  148.0 168.0 24.0 24.0

ArcClass.CONSUMPTION glyph1 pn1.1 a01
98.0 160.0
136.0 180.0
ArcClass.PRODUCTION pn1.2 glyph_nadh a02
184.0 180.0
300.0 180.0
ArcClass.CATALYSIS glyph_adh1 pn1 a03
160.0 80.0
160.0 168.0
ArcClass.PRODUCTION pn1.2 glyph_h a04
184.0 180.0
224.0 202.0
ArcClass.PRODUCTION pn1.2 glyph_ethanal a05
184.0 180.0
224.0 154.0
ArcClass.CONSUMPTION glyph_nad pn1.1 a06
95.0 202.0
136.0 180.0
```

2.3 SBGN notes

The optional SBGN element named 'notes', present on every major SBGN component type, is intended as a place for storing optional information intended to be seen by humans. An example use of the 'notes' element would be to contain formatted user comments about the model element in which the 'notes' element is enclosed. Every object derived directly or indirectly from type SBase can have a separate value for 'notes', allowing users considerable freedom when adding comments to their models.

The format of 'notes' elements must be XHTML 1.0 (http://www.w3.org/1999/xhtml).

2.3.1 Write SBGN notes

[4]:
```
from libsbgnpy import libsbgn, Notes, Language
from libsbgnpy import utils

sbgn = libsbgn.sbgn()
map = libsbgn.map()
map.set_language(Language.PD)
sbgn.set_map(map)

# create a glyph with an id and class "macromolecule"
g = libsbgn.glyph()
g.set_id("g1")

# define a label for this glyph
label = libsbgn.label()
label.set_text("INSR")

bbox = libsbgn.bbox(x=100, y=100, w=80, h=40)
g.set_bbox(bbox)
map.add_glyph(g)

notes = Notes("""
<body xmlns="http://www.w3.org/1999/xhtml">
    This is an example note describing the INSR glyph.
</body>""")
g.set_notes(notes)

pprint_xml(utils.write_to_string(sbgn))
```

```
<IPython.core.display.HTML object>
```

2.3.2 Read SBGN notes

SBGN notes can be read via the `get_notes` function which returns a `Notes` instance. We just display the note we wrote before on one of the glyphs.

[5]:
```
map = sbgn.get_map()

glyphs = map.get_glyph()
for g in glyphs:
    notes = g.get_notes()
    if notes:
        print('* {} *'.format(g.get_id()))
        print(notes)
```

```
* g1 *
<body xmlns="http://www.w3.org/1999/xhtml">
    This is an example note describing the INSR glyph.
</body>
```

2.4 SBGN extension

SBGN allows to write extension information. This can be any well-formed XML content. Whereas Notes is a container for content to be shown directly to humans, Extension is a container for optional software-generated content not meant to be shown to humans. Every SBGN object can have its own Extension object instance. In XML, the Extension content type is any, allowing essentially arbitrary well-formed XML data content.

2.4.1 Write SBGN extension

[6]:
```
from libsbgnpy import libsbgn
from libsbgnpy import utils
from libsbgnpy import Extension, Language
sbgn = libsbgn.sbgn()
map = libsbgn.map()
map.set_language(Language.PD)
sbgn.set_map(map)

extension = Extension("""<renderInformation id="example" programName="SBML Layout"␣
↪programVersion="3.0"
 xmlns="http://projects.eml.org/bcb/sbml/render/level2">
    <listOfColorDefinitions>
    <colorDefinition id="yelloComp" value="#ffffccff" />
    <colorDefinition id="grayComp" value="#e0e0e0ff" />
    <colorDefinition id="orange" value="#fa9e2fff" />
    <colorDefinition id="blue" value="#2958acff" />
    <colorDefinition id="green" value="#378f5cff" />
    <colorDefinition id="Color_0" value="#969696" />
    <colorDefinition id="Color_1" value="#ff9900" />
    <colorDefinition id="Color_2" value="#000000" />
    </listOfColorDefinitions>
    <listOfGradientDefinitions>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"LinearGradient_0" spreadMethod="reflect">
        <stop offset="0%" stop-color="#ccffff" />
        <stop offset="100%" stop-color="#ffffff" />
    </linearGradient>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"OrangeGradient_0" spreadMethod="reflect">
        <stop offset="0%" stop-color="#ffffff" />
        <stop offset="100%" stop-color="#fa9e2fff" />
    </linearGradient>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"BlueGradient_0" spreadMethod="reflect">
        <stop offset="0%" stop-color="#ffffff" />
        <stop offset="100%" stop-color="#2958acff" />
    </linearGradient>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"GreenGradient_0" spreadMethod="reflect">
```

(continues on next page)

(continued from previous page)

```
        <stop offset="0%" stop-color="#ffffff" />
        <stop offset="100%" stop-color="#378f5cff" />
    </linearGradient>
    </listOfGradientDefinitions>
    <listOfStyles>
    <style idList="glyph0 glyph2 glyph14 glyph34 ">
        <g stroke="Color_2" stroke-width="5" fill="yelloComp"  />
    </style>
    <style idList="glyph1">
        <g stroke="Color_2" stroke-width="5" fill="grayComp"  />
    </style>
    </listOfStyles>
</renderInformation>""")
map.set_extension(extension)

pprint_xml(utils.write_to_string(sbgn))
```

```
<IPython.core.display.HTML object>
```

2.4.2 Read SBGN extension

We now read the map extension information written in the example above.

[7]:
```
# map is a container for the glyphs and arcs
map = sbgn.get_map()

extension = map.get_extension()
if extension:
    print(extension)
```

```
<renderInformation id="example" programName="SBML Layout" programVersion="3.0"
 xmlns="http://projects.eml.org/bcb/sbml/render/level2">
    <listOfColorDefinitions>
    <colorDefinition id="yelloComp" value="#ffffccff" />
    <colorDefinition id="grayComp" value="#e0e0e0ff" />
    <colorDefinition id="orange" value="#fa9e2fff" />
    <colorDefinition id="blue" value="#2958acff" />
    <colorDefinition id="green" value="#378f5cff" />
    <colorDefinition id="Color_0" value="#969696" />
    <colorDefinition id="Color_1" value="#ff9900" />
    <colorDefinition id="Color_2" value="#000000" />
    </listOfColorDefinitions>
    <listOfGradientDefinitions>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"LinearGradient_0" spreadMethod="reflect">
        <stop offset="0%" stop-color="#ccffff" />
        <stop offset="100%" stop-color="#ffffff" />
    </linearGradient>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"OrangeGradient_0" spreadMethod="reflect">
        <stop offset="0%" stop-color="#ffffff" />
        <stop offset="100%" stop-color="#fa9e2fff" />
    </linearGradient>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"BlueGradient_0" spreadMethod="reflect">
        <stop offset="0%" stop-color="#ffffff" />
```

(continues on next page)

(continued from previous page)

```
        <stop offset="100%" stop-color="#2958acff" />
    </linearGradient>
    <linearGradient x1="0%" y1="0%" z1="0%" x2="100%" y2="0%" z2="100%" id=
↪"GreenGradient_0" spreadMethod="reflect">
        <stop offset="0%" stop-color="#ffffff" />
        <stop offset="100%" stop-color="#378f5cff" />
    </linearGradient>
    </listOfGradientDefinitions>
    <listOfStyles>
    <style idList="glyph0 glyph2 glyph14 glyph34 ">
        <g stroke="Color_2" stroke-width="5" fill="yelloComp"  />
    </style>
    <style idList="glyph1">
        <g stroke="Color_2" stroke-width="5" fill="grayComp"  />
    </style>
    </listOfStyles>
</renderInformation>
```

2.5 Read & Write SBGN

Helper functions for reading, writing and printing SBGN are provided in the `utils` module.

2.5.1 Write SBGN to string

The SBGN string can be generated from the `sbgn` object via

[8]:
```
from libsbgnpy import utils
sbgn_str = utils.write_to_string(sbgn)
```

[9]:
```
pprint_xml(sbgn_str)
```

```
<IPython.core.display.HTML object>
```

2.5.2 Write SBGN to file

[10]:
```
from libsbgnpy import utils
sbgn_str = utils.write_to_string(sbgn)
f_out = tempfile.NamedTemporaryFile(suffix=".sbgn")
sbgn.write_file(f_out.name)
```

2.5.3 Read SBGN from file

[11]:
```
from libsbgnpy import utils
sbgn = utils.read_from_file("glycolysis.sbgn")
print(sbgn)
```

```
<libsbgnpy.libsbgn.sbgn object at 0x7fb5c0d388b0>
```

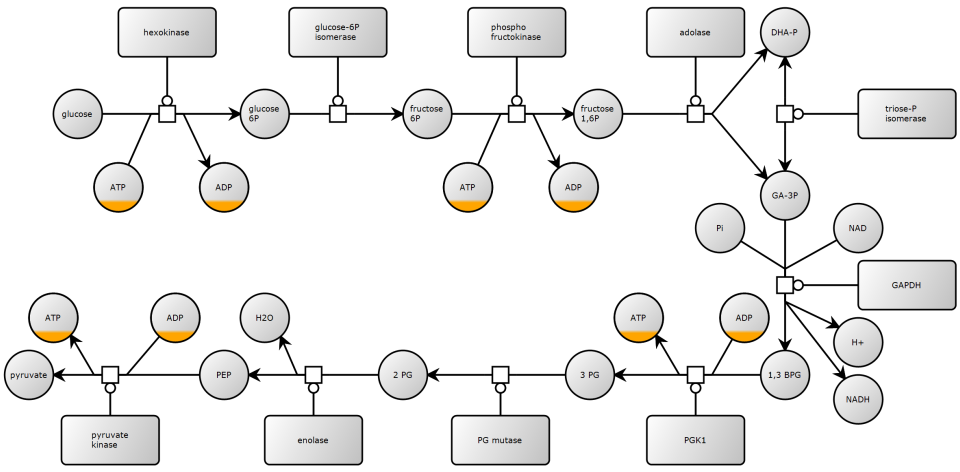
2.6 Render SBGN

SBGN images can be renderd via the `render` module.

[12]:
```
## Render SBGN
from libsbgnpy import render, utils
sbgn = utils.read_from_file("glycolysis.sbgn")
f_png = tempfile.NamedTemporaryFile(suffix=".png")
render.render_sbgn(sbgn, image_file=f_png.name,
                   file_format="png")
Image(f_png.name, width=500)
```

```
SBGN rendered: /tmp/tmpwsj4rf6v.png
```

[12]:



[ ]:

CHAPTER

THREE

API

3.1 libsbgnpy.libsbgn

class libsbgnpy.libsbgn.**SBGNBase**(*notes=None*, *extension=None*, *extensiontype_=None*)

The SBGNBase type is the base type of all main components in SBGN. It supports attaching metadata, notes and annotations to components.

write_file(*outfile*, *namespace='sbgn'*)

Write SBGN to file.

This also fixes the issues of the sbgn namespace prefix.

Parameters

- **outfile** – SBGN file to write
- **namespace** –

Returns

class libsbgnpy.libsbgn.**arc**(*notes=None*, *extension=None*, *class_=None*, *id=None*, *source=None*, *target=None*, *glyph=None*, *port=None*, *start=None*, *next=None*, *end=None*)

The arc element describes an SBGN arc between two SBGN nodes. It contains: For PD: an optional stoichiometry marker, For ER: an optional cardinality marker, zero or more ports (influence targets), and zero or more outcomes, a mandatory source and target (glyph or port), a geometric description of its whole path, from start to end. This path can involve any number of straight lines or quadratic/cubic Bezier curves. The class attribute defines the semantic of the arc, and influences: the way that arc should be rendered, the overall syntactic validity of the map. The various classes encompass all possible types of SBGN arcs: production and consumption arcs, all types of modification arcs, logic arcs, equivalence arcs. To express a reversible reaction, use production arcs on both sides of the Process Node. The xsd:ID type is an alphanumeric identifier, starting with a letter. The source attribute can refer: either to the id of a glyph, or to the id of a port on a glyph. The target attribute can refer: either to the id of a glyph, or to the id of a port on a glyph.

get_class()

Get the ArcClass.

set_class(*class_*)

Set the ArcClass. :param **class_**: :return:

superclass

alias of *SBGNBase*

class libsbgnpy.libsbgn.**arcgroup**(*notes=None*, *extension=None*, *class_=None*, *glyph=None*, *arc=None*)

The arc group describes a set of arcs and glyphs that together have a relation. For example For ER: interaction

arcs around an interaction glyph, ... Note that, in spite of the name, an arcgroup contains both arcs and glyphs. The class attribute defines the semantic of the arcgroup.

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**bbox**(*x=None*, *y=None*, *w=None*, *h=None*, *notes=None*, *extension=None*)
The bbox element describes a rectangle. This rectangle is defined by: PointAttributes corresponding to the 2D coordinates of the top left corner, width and height attributes. The rectangle corresponds to the outer bounding box of a shape. The shape itself can be irregular (for instance in the case of some compartments). In the case of process nodes, the bounding box only concerns the central glyph (square, or circle), the input/output ports are not included, and neither are the lines connecting them to the central glyph. A bbox is required for all glyphs, and is optional for labels.

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**endType**(*x=None*, *y=None*, *point=None*)
List of control points, used when the path describes a curve. The number of points describes the degree of the Bezier curve: linear (0), quadratic (1) or cubic (2)

class libsbgnpy.libsbgn.**glyph**(*notes=None*, *extension=None*, *class_=None*, *orientation=<Orientation.HORIZONTAL: 'horizontal'>*, *id=None*, *compartmentRef=None*, *compartmentOrder=None*, *label=None*, *state=None*, *clone=None*, *callout=None*, *entity=None*, *bbox=None*, *glyph_member=None*, *port=None*)
The glyph element is: either a stand-alone, high-level SBGN glyph (EPN, PN, compartment, etc), or a sub-glyph (state variable, unit of information, inside of a complex, ...) In the first case, it appears directly in the glyph list of the map. In the second case, it is a child of another glyph element. The text inside a glyph is described: either by a label element (optional) [process nodes can't have one], or by a state element (optional) [for state variables only]. The class attribute defines the semantic of the glyph, and influences: the way that glyph should be rendered, the overall syntactic validity of the map. The various classes encompass the following PD SBGN elements: Entity Pool Nodes (EPN), Process Nodes (PN), Logic Operator Nodes, Sub-glyphs on Nodes (State Variable, Unit of Information), Sub-glyphs on Arcs (Stoichiometry Label), Other glyphs (Compartment, Submap, Tag, Terminal). And the following ER SBGN elements Entities (Entity, Outcome) Other (Annotation, Phenotype) Auxiliary on glyps (Existence, Location) Auxiliary on arcs (Cardinality) Delay operator implicit xor The orientation attribute is used to express how to draw asymmetric glyphs. In PD, the orientation of Process Nodes is either horizontal or vertical. It refers to an (imaginary) line connecting the two in/out sides of the PN. In PD, the orientation of Tags and Terminals can be left, right, up or down. It refers to the direction the arrow side of the glyph is pointing at. The xsd:ID type is an alphanumeric identifier, starting with a letter. It is recommended to generate meaningless IDs (e.g. "glyph1234") and avoid IDs with a meaning (e.g. "epn_ethanol") Reference to the ID of the compartment that this glyph is part of. Only use this if there is at least one explicit compartment present in the diagram. Compartments are only used in PD and AF, and thus this attribute as well. For PD, this should be used only for EPN's. In case there are no compartments, entities that can have a location, such as EPN's, are implicit member of an invisible compartment that encompasses the whole map. In that case, this attribute must be omitted. The compartment order attribute can be used to define a drawing order for compartments. It enables tools to draw compartments in the correct order especially in the case of overlapping compartments. Compartments are only used in PD and AF, and thus this attribute as well. The attribute is of type float, the attribute value has not to be unique. Compartments with higher compartment order are drawn on top. The attribute is optional and should only be used for compartments.

get_class()
Get the Language. :return: Language instance.

get_orientation()
Get orientation. :return: Orientation instance.

set_class(*class_*)
Sets the class and checks that in allowed GlyphClasses :param class_: :return:

set_orientation(*orientation*)
Sets orientation and checks that allowed Orientation.

Parameters **orientation** –

Returns

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**label**(*notes=None*, *extension=None*, *text=None*, *bbox=None*)
The label element describes the text accompanying a glyph. The text attribute is mandatory. Its position can be specified by a bbox (optional). Tools are free to display the text in any style (font, font-size, etc.) The bbox element of a label is optional. When no bbox is defined, the bbox of the parent glyph is inherited. The label should be drawn centered horizontally and vertically in the bbox. When the bbox is inherited, the label can freely spill outside (just like it can spill outside its parent glyph). An explicit bbox provides a stronger hint regarding what surface the label should cover. It defines an upper boundary outside of which the label should (ideally) not spill. It also represents a preferred size: the surface covered by the label can be smaller, but should ideally be as close as possible to the bbox. In most glyphs (EPNs, unit of information, etc.), the label is supposed to be centered, so the bbox is usually omitted (unless there's a specific hint to be shared concerning the area the label should ideally cover). However, labels can be drawn anywhere inside compartments or complex, so these should preferably have an explicit bbox. Multi-line labels are allowed. Line breaks are encoded as
 as specified by the XML standard.

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**map**(*notes=None*, *extension=None*, *language=None*, *bbox=None*, *glyph=None*, *arc=None*, *arcgroup=None*)
The map element describes a single SBGN PD map. It contains a list of glyph elements and a list of arc elements. These lists can be of any size (possibly empty). Language of the map: one of three sublanguages defined by SBGN. Different languages have different restrictions on the usage of sub-elements (that are not encoded in this schema but must be validated with an external validator)

get_language()
Get the Language. :return: Language instance.

set_language(*language*)
Sets the language and checks that within allowed values. :param language: :return:

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**nextType**(*x=None*, *y=None*, *point=None*)
List of control points, used when the path describes a curve. The number of points describes the degree of the Bezier curve: linear (0), quadratic (1) or cubic (2)

class libsbgnpy.libsbgn.**point**(*notes=None*, *extension=None*, *x=None*, *y=None*)
The point element is characterized by PointAttributes, which describe absolute 2D cartesian coordinates. Namely: x (horizontal, from left to right), y (vertical, from top to bottom). The origin is located in the top-left corner of the map. There is no unit: proportions must be preserved, but the maps can be drawn at any scale. In the test files examples, to obtain a drawing similar to the reference *.png file, values in the corresponding *.sbgn file should be read as pixels.

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**port**(*notes=None*, *extension=None*, *id=None*, *x=None*, *y=None*)
The port element describes an anchor point which arcs can refer to as a source or target. It consists in: absolute 2D cartesian coordinates (PointAttribute), a unique id attribute. Two port elements are required for process nodes. They represent the extremity of the two "arms" which protrude on both sides of the core of the glyph (= square or circle shape). Other glyphs don't need ports (but can use them if desired). The xsd:ID type is an alphanumeric identifier, starting with a letter. Port IDs often contain the ID of their glyph, followed by a local port number (e.g. glyph4.1, glyph4.2, etc.) However, this style convention is not mandatory, and IDs should never be interpreted as carrying any meaning.

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**sbgn**(*notes=None*, *extension=None*, *map=None*)
The sbgn element is the root of any SBGNML document. Currently each document must contain exactly one map element.

superclass
alias of *SBGNBase*

class libsbgnpy.libsbgn.**stateType**(*value=None*, *variable=None*)
The value attribute represents the state of the variable. It can be: either from a predefined set of string (P, S, etc.) which correspond to specific SBO terms (cf. SBGN specs), or any arbitrary string. The variable attribute describes the site where the modification described by the value attribute occurs. It is: optional when there is only one state variable on the parent EPN, required when there is more than one state variable the parent EPN.

3.2 libsbgnpy.libsbgnTypes

Definition of Language, GlyphClass and ArcClass types. Created manually from schema file.

class libsbgnpy.libsbgnTypes.**ArcClass**(*value*)
Enumeration with all possible values for the class attribute of Arcs in SBGN-ML.

class libsbgnpy.libsbgnTypes.**GlyphClass**(*value*)
Enumeration with all possible values for the class attribute of Glyphs in SBGN-ML. This includes both top-level glyphs and sub-glyphs.

class libsbgnpy.libsbgnTypes.**Language**(*value*)
Enum representing the three languages of SBGN.

class libsbgnpy.libsbgnTypes.**Orientation**(*value*)
An enumeration.

3.3 libsbgnpy.render

Helper functions for rendering SBGN.

Currently uses the webservice provided at "http://sysbioapps.dyndns.org/Layout/GenerateImage". For documentation see http://sysbioapps.dyndns.org/Home/Services

libsbgnpy.render.**render_sbgn**(*sbgn*, *image_file*, *file_format='png'*)
Render given sbgn object to image.

Currently supports the following file_formats: - "png" The image file must end in .file_format, e.g. in '.png'

Performs a request analogue to: curl -X POST -F file=@".BorisEJB.xml" http://sysbioapps.spdns.org/Layout/GenerateImage -o out.png

Parameters

- **sbgn** – sbgn object
- **image_file** – image to create

Returns None

3.4 libsbgnpy.utils

Helper functions to work with SBGN.

libsbgnpy.utils.**get_language**(*f*)
SBGN language of the map. Returns a Language value.

Parameters **f** –

Returns

libsbgnpy.utils.**get_version**(*f*)
SBGN version.

1: xmlns="http://sbgn.org/libsbgn/0.1 2: xmlns="http://sbgn.org/libsbgn/0.2 3: xmlns="http://sbgn.org/libsbgn/0.3

Parameters **f** – file for which version should be found.

Returns version as an integer, i.e. 1, 2, 3

libsbgnpy.utils.**print_bbox**(*b*)
Print bounding box representation.

Parameters **b** –

Returns

Return type

libsbgnpy.utils.**read_from_file**(*f*, *silence=True*)
Read an sbgn file (without validating against the schema).

Parameters

- **silence** – display no information
- **f** – file to read

Returns parsed SBGN

Return type

libsbgnpy.utils.**write_to_file**(*sbgn*, *f*)
Write sbgn object to file.

Parameters

- **sbgn** –
- **f** –

Returns

libsbgnpy.utils.**write_to_string**(*sbgn*)
Write SBGN to string. Returns None if problems.

Parameters **sbgn** – sbgn object

Returns SBGN xml string

3.5 libsbgnpy.validation.validator

This is an example that shows both low-level validation against the XSD Schema, and high-level validation using schematron.

At this point only the XSD validation is implemented. see [#7](https://github.com/matthiaskoenig/libsbgn-python/issues/7)

class libsbgnpy.validation.validator.**Issue**(*role*, *rule_id*, *diagnostic_id*, *message*)
Describes one issue found during schematron validation. One validation run may produce multiple issues.

get_diagnostic_id()
Identifier of the element that this issue is about.

get_message()
Human readable description of the issue.

get_rule_id()
Identifier of the issue

get_severity()
Severity of the issue, i.e.: is it an error, or a warning?

class libsbgnpy.validation.validator.**Severity**(*value*)
An enumeration.

libsbgnpy.validation.validator.**validate_schematron**(*f_sbgn*)
Validate SBGN file with schematron.

In Java this does the XSL Transformations on the ruleset and the exported Pathway Object and then invokes the SAX parser through "parseSVRL" method on the transformation's result.

Parameters **f** – SBGN file

Returns

libsbgnpy.validation.validator.**validate_xsd**(*f*)
Validate SBGN file against XSD schema.

Parameters **f** – file to validate

Returns Returns None if valid, the error log otherwise.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

V

W