Constructing Petri Net Transducers with $\text{PNT}_{\varepsilon}^{\text{ooL}}$

Markus Huber *Robert Lorenz* *Daniel Straßner*

Department of Computer Science
University of Augsburg
markus.huber@informatik.uni-augsburg.de

Abstract: Petri net transducers (PNTs) were developed for the translation of utterances into meanings within a hierarchical cognitive dynamic speech signal processing system [9, 10].

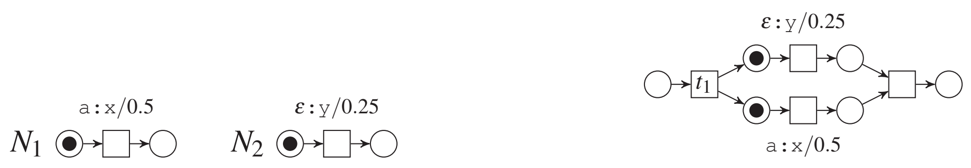
PNTs are a natural generalisation of weighted finite state transducers (FSTs) for the translation of partial languages consisting of partial words (with a partial order on their symbols) instead of (linear) words (having a total order on their symbols).

$\text{PNT}_{\varepsilon}^{\text{ooL}}$ is a python library for the modular construction of PNTs through composition operations. Constructed PNTs can be exported in an XML-format which is based on the standard PNML format developed for basic Petri net variants [16]. Moreover, PNTs can be visualised and pictures can be exported in all standard formats. $\text{PNT}_{\varepsilon}^{\text{ooL}}$ serves as a basis for the implementation and evaluation of algorithms for analysis, simulation and optimisation of PNTs.

$\text{PNT}_{\varepsilon}^{\text{ooL}}$ is implemented within the framework SNAKES aiming at the quick prototyping of new Petri net classes [12, 13]. Its basic functionality was developed in the bachelor thesis [15].

1 Introduction

In [9] we introduced Petri net transducers and showed in [7, 10] how they can be successfully applied in the field of semantic dialogue modelling for translating utterances into meanings.
In short PNTs are a formalism for the weighted translation of labelled partial orders (LPOs), which are words consisting not of a total order between their symbols but of a partial order. In this sense they are a generalisation of weighted finite state transducers translating words.
In figure 1a on the following page one can see two simple PNTs called N_1 and N_2. A PNT consists of transitions (drawn as rectangles), places (circles) and a flow relation between places and transitions (directed edges). The flow relation assigns pre- and post-conditions to transitions. The state of a PNT is given by tokens marking places (black filled circles inside places). If a place is marked by a token, then the corresponding condition is satisfied. The occurrence of a transition is possible, if all of its pre-conditions are satisfied. Its occurrence leads to a state where none of its pre-conditions and all of its post-conditions are satisfied. Like for weighted FSTs input symbols, output symbols and weights are annotated to the transitions. The weights are elements of an algebraic structure called bisemiring [9]. So the transition of N_1 reads the symbol `a` and writes the symbol `x` (with weight 0.5) – thus N_1 translates the word `a` into the word `x` (with weight 0.5). The symbol ε is used to denote empty input or output. So the PNT N_2 is a generator which produces the word `y`.
The main difference to automata is that the state of a PNT is distributed over several locations. If, in some state, two transitions do not share pre-conditions and all pre-conditions of both transitions are satisfied, then both transitions may occur independently in any order or also



(a) A transducer and a generator. (b) Parallel product of N_1 and N_2 after t_I has fired.

Figure 1: Some simple PNTs.

simultaneously. Such transitions are called concurrent (in the considered state). To be able to combine the weights of concurrent transitions, a bisemiring extends the weight structure of semirings used in the case of FSTs by an operation for parallel multiplication of weights.

As for FSTs there exist composition operations for combining simple transducers to more complex ones. For example, figure 1b shows the parallel product of the PNTs from figure 1a – an operation which does not exist for FSTs. In figures, we omit annotations of the form $\varepsilon{:}\varepsilon/\overline{1}$ where $\overline{1}$ is the neutral weight w.r.t. sequential and parallel multiplication. The PNT from figure 1b is not in its initial state – a single token in a distinct source place – but in the state after the transition t_1 has occurred. In this state there a two transitions which can occur concurrently. The shown transducer realises the translation from the word `a` into the LPO `w=x`$\|$`y` where $\|$ denotes the parallel composition of LPOs and the two symbols x and y are unordered in w.

Note that PNTs are defined to always have a single source place which holds exactly one token at the initial state. Furthermore a PNT has a single sink place and per definition only such (non-sequential) runs of the net are considered, which lead to a state where exactly the sink place is marked by one token.[1]

For such a formalism to be useful one needs a tool where PNTs can be implemented, analysed, combined, simulated, drawn and the like. Since the PNT-formalism is new we decided to start $\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$. We use the SNAKES framework [13] which is a Python [17] library supporting the rapid prototyping of new Petri net formalisms and provides many basic Petri net components and functionality. Therefore we implemented $\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$ as a Python library extending SNAKES such that we essentially can use all the functionality already provided by SNAKES. All graphics in this paper were generated by $\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$.

The paper is organised as follows: In section 2 we describe the objectives of $\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$. In section 3 we explain its functionality. In section 4 we give a brief overview of its architecture. Finally, in section 5 we compare $\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$ to other Petri net tools, give a brief outlook on further developments, and provide information on how to obtain and install $\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$.

2 Objectives

$\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$ is mainly targeted at researchers in the area of PNTs. By the use of SNAKES it is relatively easy to implement and evaluate extensions, variations and new algorithms[2], as for example: Additional composition operations, optimised algorithms for existing composition operations, new algebraic weight structures, simulation algorithms, and optimisation algorithms (ε-removal, weight-pushing).

The support of graphical output serves both as a possibility to check the implementation and as a handy utility in the process of writing scientific papers. $\mathrm{PNT}_{\varepsilon}^{\mathrm{ooL}}$'s functionality supports fast

[1]Runs are LPOs over transition names. Each run translates an input LPO into an output LPO via a projection onto input symbols resp. output symbols [9].

[2]The SNAKES website states: "SNAKES' main aim is to be a general Petri net library, being able to cope with most Petri nets models, and providing the researcher with a tool to quickly prototype new ideas."

construction of concrete example PNTs for case studies. PNML export can be used to analyse constructed example PNTs with other Petri net tools.

In the context of our research activities, PNT$_{\varepsilon}^{ooL}$ serves as a scientific prototype for the development of an open library openPNT of efficient algorithms for the construction, composition, simulation and optimisation of PNTs which can be used in real world examples.

3 Functionality

In this section we show how PNT$_{\varepsilon}^{ooL}$ is used to construct and compose PNTs. We list the source code of the examples and also show the graphics produced by the programme.

To use PNT$_{\varepsilon}^{ooL}$ one has to create a text file and put the following code into it. These lines load the SNAKES library and the `transducer`-plugin.

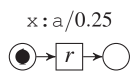
```
1  import snakes.plugins
2  snakes.plugins.load(['transducer'], 'snakes.nets', 'pnts')
3  from pnts import *
```

We now create our first PNTs and produce the two outputs from figure 1a on the previous page. For this the class method `N` from the class `PetriNet` is used. This way in lines 4 and 7 each a PNT consisting of a source place with one token, a sink place, and a single transition in between is created. As one can see the parameters of this single transition are provided as named parameters to `N`. In lines 5 and 8 graphical output of the PNTs is generated. Note that the orientation of the generated graphs is controlled by the parameter `leftright` of the method `draw` in those lines. This parameter is only effective if Ti*k*Z-output is to be created (see subsection 4.3.1 for prerequisites). The format of the output is controlled by the extension of the file-name given to the method `draw` as first argument. For example in line 17 on the following page a PNT is exported into the PNG format which is shown in figure 2c where one can see that tokens are not drawn as filled circles but instead are given by its count inside a place. For more available export formats one may consult the documentation of the Graphviz [6] package which is utilised by SNAKES for the export.

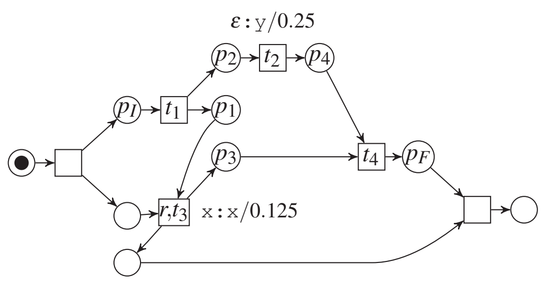
```
4  n1 = PetriNet.N('N1', weight = .5, input_symbol = 'a',
       output_symbol = 'x')
5  n1.draw("N1.tikz", leftright = True)
6
7  n2 = PetriNet.N('N2', weight = .25, output_symbol='y')
8  n2.draw("N2.tikz", leftright = True)
```

Now we use the operator of parallel composition in line 9 to create the PNT from figure 1b on the previous page. After the composition we change the left one of the new transitions. By default, transitions which are added during compositions get the name `t_I` if they are connected to the source place and `t_F` if they are connected to the sink place. If two transitions are added they are named `t_1I` and `t_2I` resp. `t_1F` and `t_2F`. By using the method `transition` in line 10 with a name as parameter we receive the transition object corresponding to this name. We change the label of the transition – its `print_name` – to `t_1` in line 11 which gets rendered as t_1 by LaTeX if our `sty`-file is used with the option `petrimath` putting the text of places and transitions in math-mode. In line 12 the transition is fired according to the description of *occurrence* in the introduction. To finally get the picture from figure 1b one has to adjust the position of the label `a:x`/0.5 in the generated Ti*k*Z-Code (output of line 14) since it is positioned above the transition by default (see subsection 4.3.1 for more information).

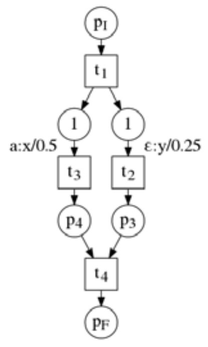
```
9  n = n1 | n2
```



(a) PNT N_3.



(b) Language composition.



(c) PNT from figure 1b with new names as PNG.

Figure 2: Some more PNTs.

```
10    t = n.transition('t_I')
11    t.set_print_name('t_1')
12    t.fire()
13
14    n.draw("N1␣par␣N2.tikz", leftright = True)
```

Subscripts can be also rendered in graphical output if the version of Graphviz supports HTML. The HTML output of $\mathrm{PNT}_\varepsilon^{\mathrm{ooL}}$ can be switched on and off according to lines 15 and 18. In line 16 we change the `print_names` of all places and transitions via topological sorting. If the transducer contains any cycles an error is raised. The output of line 17 is shown in figure 2c.

```
15    activateSub()
16    n.clean_names(print_names = True)
17    n.draw("N1␣par␣N2.png")
18    deactivateSub()
```

We create another PNT in line 19 and change the label of its only transition which is named `t` by default in line 20. The drawing of `n3` is shown in figure 2a. In line 23 the language composition of the newly created PNT and the transducer from figure 2c is computed. The result is depicted in figure 2b (output of line 24). Note that the `print_names` were retained resp. merged by the composition operation. The weight of the transition labelled r,t_3 is computed from the weights of the transitions labelled r and t_3 using the sequential product of the bisemiring associated to the PNTs. This structure defaults to a bisemiring based on the Viterbi-semiring with min as parallel multiplication. Whenever transitions are merged their weights are also merged and an error is raised if the bisemirings of the operands are different.

```
19    n3 = PetriNet.N('N3', weight = .25, input_symbol = 'x',
          output_symbol = 'a')
20    n3.transition('t').set_print_name('r')
21    n3.draw("N3.tikz", leftright = True)
22
23    comp = n3 > n
24    comp.draw("N3␣comp␣N1␣par␣N2.tikz", leftright = True)
```

As a last example we export a PNT with the function `savePNML` – a wrapper of SNAKES methods – taking a file-name as second argument. For loading we provide `loadPNML`.

```
25    savePNML(n, "N1␣par␣N2.pnml")
```

There are more operators defined for PNTs namely `*` for concatenation, `+` for union, `~` for closure, and `^` for synchronous product as defined in [10].

4 Architecture

PNT$_{\varepsilon}^{ooL}$ is implemented as a bunch of plugins on top of SNAKES and thus as a Python library. At first we describe some aspects of SNAKES itself and afterwards of its plugin-system. After that we present all the plugins that form PNT$_{\varepsilon}^{ooL}$.

4.1 The Python Library SNAKES

In SNAKES Petri nets are implemented as a collection of objects. The class `PetriNet` puts them all together as it contains objects of the classes `Place` and `Transition` – through the methods `add_place` and `add_transition` – and provides the methods `add_input` and `add_output` to link places and transitions.
Actually SNAKES implements so-called coloured Petri nets [8] where Python objects and expressions are used for the annotations. However PNT$_{\varepsilon}^{ooL}$ does not use most of these features.
SNAKES allows for firing of transitions by the method `fire` of class `Transition`. The class `StateGraph` provides a method `build` which can compute the state graph for a given net. As nets, state graphs can also be exported to pictures by the use of the method `draw`. Another feature also present in the core of SNAKES is the export in PNML format.
For extensibility SNAKES implements a plugin-system briefly described in the next subsection.

4.2 The Plugin-System of SNAKES

A plugin for SNAKES is a separate Python library which specialises already defined classes or adds new classes to SNAKES. Plugins can be loaded and are stacked onto each other. This way a class hierarchy is established.
A function `extend` has to be implemented and some rules have to be followed for which the interested reader should refer to the SNAKES homepage.
A plugin can depend on other plugins and can even mention conflicting plugins.

4.3 The Plugins of PNT$_{\varepsilon}^{ooL}$

The following subsections describe each of the plugins that comprise PNT$_{\varepsilon}^{ooL}$. The first and fourth plugins can be used independently from the other plugins while the remaining three build upon each other.

4.3.1 The `d2t`-Plugin

This plugin extends the features of the `gv`-plugin which is delivered with SNAKES. By the use of that plugin a representation of Petri nets in the `dot`-language from Graphviz [6] can be produced which is then processed by Graphviz to compute a layout and eventually produce a graphical output. Our `d2t`-plugin adds four features to the graphical output routine.
The first one hooks into the code of `gv`-plugin. The `dot`-language is able to embed HTML constructs inside of labels. We utilise this to produce subscripts by replacing for example a label `t_I` with the text `<t<subscript>I</subscript>>` which produces the nice label $\mathtt{t}_{\mathtt{I}}$. This functionality can be switched on and off by the use of the functions `activateSub` and `deactivateSub` which are both exported into to global namespace by our plugin.

The second feature adds an attribute `_print_name` to places and transitions which can be changed by the method `set_print_name` and queried by the method `print_name` for any object of class `Place` or `Transition`. This attribute can be used by the graphical output routine for example to label only those nodes which have assigned a `_print_name`. This can lead to more lucid graphics.

As a third feature the plugin adds an attribute `_draw_defaults` to Petri nets which is set to a new object of class `DrawDefaults`. Such an object contains class-wide default values, per object default values and per object functions to influence the output of the method `draw` if all of its `*_attrs`-parameters have a value of `None`. The purpose of this attribute is to associate certain defaults to a net which change the graphical output of its components. All values and functions can be changed by the methods `set_draw`, `del_draw`, `set_draw_attr`, and `del_draw_attr` of class `PetriNet`.

The last feature adds Ti*k*Z as a new export format to the `draw`-method of Petri nets. The export utilises the package `dot2tex` [1]. This package can convert the `dot`-language into various other formats. However we only use the ability to convert into the Ti*k*Z-language. Using this feature we are able to integrate the graphical output of PNT$_{\varepsilon}^{\text{ool}}$ seamlessly into papers written in LaTeX – like this one. Note that the position of annotations of transitions is not retained by this conversion and requires manual intervention. For convenience we provide the possibility to change the orientation of the generated Ti*k*Z-output to left-to-right.

4.3.2 The `pt`-Plugin

As already said in subsection 4.1 SNAKES implements coloured Petri nets. Since the underlying net of a PNT is actually a place/transition Petri net we decided to write a plugin which restricts the nets of SNAKES by only allowing those constructs which are needed for them.

At first, the type of tokens is restricted to `tBlackToken`, the type of simple tokens of SNAKES. An error is raised if one tries to install other type restrictions for tokens. The given tokens are revised and an error is raised if other than simple tokens are to be used. For convenience any list of integers can be used instead of simple tokens. The sum of the list gives the number of simple tokens for the place.

Transitions received a restriction on their guards. Since the firing of transitions in place/transition Petri nets only depends on enough tokens in all places which are predecessors of the transition there is no guard needed. In SNAKES this means the guard is `Expression("True")` which is the default if `None` is given as a parameter. Consequently only these two values are allowed and an error is raised otherwise. The method `fire` gets `None` as a default value of its parameter `binding` since only simple tokens are used and no variables are involved.

The last restriction concerns edges. Place/transition Petri nets only allow for positive integers which state the number of tokens to be consumed resp. produced.

4.3.3 The `terminal`-Plugin

By using the `pt`-plugin and adding a few features to the class `PetriNet`, this plugin implements Petri nets that have a single source place and single sink place. Although SNAKES delivers the `label`-plugin to add properties to any node of a net we decided to implement our own mechanism because we only need a fraction of its functionality.

With these properties it is possible to define several composition operations. While SNAKES delivers the `ops`-plugin which implements composition operations according to [3] we implemented our own mechanism because the definitions of the operations defer. Our plugin implements `union`, `sequential_product`, `parallel_product`, and `closure`. By the use of the `cluster`-plugin delivered with SNAKES the operands get preserved in the layout.

Additionally we provide a notation to create a *singleton* net consisting of a source and a sink place and a transition in between. This feature is implemented as the class method `N` of class `PetriNet`. To support future extensions we use factory methods [5] for the creation of the places – `create_source` and `create_sink` – and the transition – `create_link`.

4.3.4 The `bisemiring`-Plugin

We implemented the class `Bisemiring` and the association of weights to transitions in a separate plugin. `Bisemiring` objects hold definitions of the set of weights, neutral elements and functions for binary addition, sequential and parallel multiplication of a bisemiring.

A Petri net now contains an object of class `Bisemiring` – as default a Viterbi-based bisemiring which uses min for parallel multiplication. Every transition has a weight which can be checked against the bisemiring of the net. If transitions get merged by the method `merge_transitions` an operation can be specified to compose the individual weights.

4.3.5 The `transducer`-Plugin

The last plugin builds on top of the `terminal`- and `bisemiring`-plugin and equips transitions with input- and output-symbols which can be arbitrary Python objects. An additional class implements the ε-symbol. The `N`-notation of subsection 4.3.3 is extended to support weights and input- and output-symbols for the single transition.

This plugin implements the additional composition operations of synchronisation and language composition [10] as methods `synchronous_product` and `language_composition`. As syntactic sugar we overloaded some operators for the class `PetriNet`.

Also the graphical output of transitions is changed using the functionality of `d2t`-plugin. The generated Ti*k*Z-code uses definitions from a separate `sty`-file to provide adaptable graphics.

5 Outlook

Up to our best knowledge, $\text{PNT}^{\text{ooL}}_{\varepsilon}$ is the first tool concerning PNTs. There are several other Petri net dialects dealing with transitions annotated with weights, such as stochastic Petri nets, time Petri nets, hybrid Petri nets or continuous Petri nets. But these values have a strictly different semantics (they represent e.g. time or priorities) and need no underlying algebraic structure.

Some of the algorithms we implemented or plan to implement for PNTs are similar as in the case of other Petri net classes and tools. For example there are several Petri net classes supporting composition operations which are similar to some of those for PNTs (as for example the PBC [3]). As another example, simulation of PNTs is based on unfolding algorithms for place/transition Petri nets as implemented for example in VipTool [4]. Nevertheless, it is not possible to use these similar constructions directly, since syntax and semantics are different and cannot be translated to PNTs.

Other PNT algorithms as for example concerning some special composition operations like language composition, optimisation of PNT models or the algebraic weight structure are not comparable to any functionality of existing Petri net tools. Concerning such algorithms, $\text{PNT}^{\text{ooL}}_{\varepsilon}$ is more related to the open library openFST [2] for FST algorithms. As already mentioned, we plan to develop a similar open library of PNT algorithms based on the prototype $\text{PNT}^{\text{ooL}}_{\varepsilon}$.

$\text{PNT}^{\text{ooL}}_{\varepsilon}$ can be downloaded from our website `www.informatik.uni-augsburg.de/lehrstuehle/inf/mitarbeiter/huber/software/` as a ZIP-archive. Assumed you have a working installation of Python, SNAKES, Graphviz, and dot2tex you only need to copy the `py`-files into the `plugins` sub-directory of your SNAKES installation and you should be able to reproduce the examples from section 3. The mentioned `sty`-file is also included.

References

[1] dot2tex. `http://code.google.com/p/dot2tex/`, 1 2014.

[2] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. Openfst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Twelth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11 – 23. Springer, 2007.

[3] E. Best, R. R. Devillers, and J. G. Hall. The box calculus: a new causal algebra with multi-label communication. In Rozenberg [14], pages 21 – 69.

[4] J. Desel. VipTool-Homepage., 2010.

[5] E. Gamma, R. Johnson, R. Helm, and J. Vlissides. *Design Patters: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 2003.

[6] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203 – 1233, 2000.

[7] M. Huber, C. Kölbl, R. Lorenz, and G. Wirsching. Konstruktion von UMP-Transduktoren aus Wizard-of-Oz Daten. In *Proceedings of "Elektronische Sprachsignalverarbeitung (ESSV)"*, pages 111 – 118, 2013.

[8] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EATCS Monographs in Theoretical Computer Science*. Springer, 1992.

[9] R. Lorenz and M. Huber. Petri net transducers in semantic dialogue modelling. In *Proceedings of "Elektronische Sprachsignalverarbeitung (ESSV)"*, volume 64 of *Studientexte zur Sprachkommunikation*, pages 286 – 297, 2012.

[10] R. Lorenz and M. Huber. Realizing the Translation of Utterances into Meanings by Petri Net Transducers. In *Proceedings of "Elektronische Sprachsignalverarbeitung (ESSV)"*, volume 65 of *Studientexte zur Sprachkommunikation*, 2013.

[11] S. Molnár, J. R. Heath, O. Dalle, and G. A. Wainer, editors. *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems, SimuTools 2008, Marseille, France, March 3-7, 2008*. ICST, 2008.

[12] F. Pommereau. Quickly Prototyping Petri Nets Tools with SNAKES. In Molnár et al. [11], page 17.

[13] F. Pommereau. The SNAKES toolkit. `https://www.ibisc.univ-evry.fr/~fpommereau/SNAKES`, 2013.

[14] G. Rozenberg, editor. *Advances in Petri Nets 1992, The DEMON Project*. Springer, 1992.

[15] D. Straßner. Prototypische Implementierung von Petrinetz-Transduktoren mit SNAKES. Bachelor thesis, Augsburg University, 2013.

[16] P. team. Pnml.org: The petri net markup language home page. `http://www.pnml.org`, 8 2011.

[17] G. van Rossum and F. L. Drake. *The Python Language Reference Manual*. PythonLabs, Virginia, USA, 2001.