HAL open science

Quickly prototyping Petri nets tools with SNAKES

Franck Pommereau

▶ To cite this version:

Franck Pommereau. Quickly prototyping Petri nets tools with SNAKES. International Workshop on Petri Nets Tools and APplications PNTAP 2008,, Mar 2008, Marseille, France. hal-02309814

HAL Id: hal-02309814

https://hal.science/hal-02309814v1

Submitted on 9 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Quickly prototyping Petri nets tools with SNAKES

Franck Pommereau
LACL, Université Paris Est
61 avenue du Général de Gaulle
94010 Créteil, France
pommereau@univ-paris12.fr

ABSTRACT

This paper presents the toolkit SNAKES that is aimed at providing a flexible solution to the problem of quickly prototyping Petri nets tools. In particular, SNAKES is expected to have as few built-in limitations as possible with respect to the particular variant of Petri net to be used. The goal is to make SNAKES suitable for any kind of Petri net model, including new ones for which there exists no available tool. For this purpose, SNAKES is designed as a very general Petri net core library enriched with a set of extension modules to provide specialised features. On the one hand, the core library is versatile in that it defines a general Petri net structure where all the computational aspects are delegated to an interpreted programming language. On the other hand, the extension modules provide with enough flexibility to allow to redefine easily any part of the base Petri net model.

SNAKES is released under the GNU LGPL, it can be downloaded at ⟨`http://www.univ-paris12.fr/lacl/pommereau/soft/snakes`⟩ with the API documentation and a tutorial.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets, Software libraries*; D.2.1 [**Software Engineering**]: Specifications—*Rapid prototyping*

General Terms

Design,Experimentation

Keywords

Petri nets, quick prototyping

1. INTRODUCTION

There exists a wide range of Petri net tools [19], most of them (if not all) being targeted to a particular variant of Petri nets or a few ones. When a new interesting variant is defined, it is often necessary to develop a software to support it, and this tool has to be updated as the model and the associated techniques evolve. When the research is targeted on a defined usage, as model-checking for instance, the formalism is often fixed and this situation causes no problem. The tool is simply improving over the time. But when the research is centred on the evolutions of the model itself, a tool often has a very short lifetime. It becomes then very hard for the developer to keep the pace with the theory and often it does not worth the effort as the tool will not be used anymore when the next variant of the model will be defined.

SNAKES is an attempt to solve this problem by providing a general and flexible Petri net library allowing for quick prototyping and development of ad-hoc and test tools. The requirements for such a toolkit may be the following:

1. *Built-in Petri net model.* This is the most obvious need.
2. *General and flexible.* The toolkit should be able to cope with a large variety of Petri net models. Moreover, it should be easy to extend it with new variants of Petri nets.
3. *Easy to use and portable.* The goal being to be able to quickly implement new ideas, it should not be intimidating to start programming, so the toolkit must be easy to understand. It should be also easy to install it anywhere, as well as the resulting programs.[1]
4. *Intended for prototyping.* This requirement alleviates the question of performances and solves a contradiction that would arise otherwise: a flexible and general tool with dynamic reconfiguration of its features would be hard to make fast.

It is a well known pattern for programs that need to have few built-in limitations to define a general framework with the basic required features and to provide then scripting capabilities allowing to extend the tool or redefine parts of it. This is the case for instance for the text editor EMACS or the typesetting system TEX/LATEX, both being often perceived as tools with unlimited features. SNAKES follows this pattern by defining a simple but very general Petri net structure: places, transitions, arcs, tokens. This is indeed what all Petri nets have in common. At this level of generality

[1]This is actually a general requirement as if a software is complicated and works on a very specific platform, it is likely that only few people will use it.

International Workshop on Petri Nets Tools and APplications PNTAP 2008, March 3, 2008, Marseille, France.

however, not many features are available. Then, Python [28] has been chosen as the extension language. Python is a mature, well established, interpreted language, that has all the required characteristics to meet the needs expressed above. According to its web site:

> *Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code.*

It may be added that Python is free software and runs on a very wide range of platforms. In order to provide with (hopefully) unlimited scripting capabilities, SNAKES delegates all the computational aspects of Petri nets to Python. In particular, a token is an arbitrary Python object, transitions execution can be guarded by arbitrary Python Boolean expressions, and so on. As a result, a Petri net in SNAKES is mainly a skeleton with very general behavioural rules (consume and produce tokens in places through the execution of transitions) and with the full power of a programming language at any point where a computation is required. SNAKES itself is programmed in Python and uses the capability of the language to dynamically evaluate arbitrary statements. Using the same programming language for SNAKES and its extension language is a major advantage for the generality: Petri nets in SNAKES can use SNAKES as a library and work on Petri nets. For instance, as a token in SNAKES may be any Python object, it could be an instance of the Petri net class of SNAKES.

SNAKES is more particularly targeted to the family of the Petri Box Calculus (PBC) [4, 5] and M-nets [6] for which Petri nets may be composed as terms in a process algebra. Many variants of the base model exist and new ones are still under development, each being focused on the study of a particular feature (*e.g.*, time, preemption, threads, exceptions, ...). In order to provide support for these models without specialising the general framework presented above, SNAKES comes with various extension modules (also called plugins) to address the various aspects of these models. This plugin system is a simple and convenient way to extend and specialise the core library. Many short-life tools or prototypes can be developed as plugins; this was made for instance during a work about the verification of Petri nets equipped with unbounded integer variables [27]. This particular case is related in the section 3.2.3 below where the perceived benefits of prototyping with SNAKES in parallel with writing the paper are explained.

All but one of the requirements listed above have been discussed so far. The ease to use has been tested with four undergraduate students in computer science, at the end of their first year of master degree. All four were average students having no prior knowledge of Python and only basic notions about Petri nets. They were given a one hour presentation of Python and SNAKES and provided with the Python tutorial. Each of them had to implement an algorithm described in a paper: McMillan's unfolding [20], Finkel's version of Karp & Miller's coverability graph [18], M-nets unfolding to P/T-nets [6] and Petri net semantics of MINs [24]. All the four students could provide a working program after a few weeks of work, none of them did request for help except to understand the papers, and the quality of their programs distributed evenly from acceptable to excellent.

1.1 Use cases

SNAKES has been used to implement various Petri nets semantics:

- the Causal Time Calculus [25] is a PBC-like process algebra with a Petri net semantics that has been implemented on the top of SNAKES in less than 200 straightforward lines of code;
- BOON [9] is an object-oriented programming notation with a Petri net semantics that has been implemented using SNAKES called from a Java program;
- a Petri net semantics of MINs interconnection networks has been implemented and used for simulation [24];
- a Petri net semantics of the Security Protocol Language (SPL) [8] has required also about 200 lines of code.

In all these cases, performances were not an issue as the goal was to use the ability of SNAKES to perform various compositions on Petri nets. But it was not used to execute the nets produced (except for MINs where execution time was not critical). This kind of application is actually the main intended usage for SNAKES that was started in order to support models from the PBC and M-nets family. It is often the case when working with this family that the main issue is to build a Petri net that is later verified using a specialised tool. This holds for instance with the semantics of SPL that is computed using SNAKES, yielding a Petri net that is translated to the Helena [15] formalism for fast verification.

One other use case of SNAKES is developed more in details in the section 3.2.3 and consisted in implementing a state space construction. In this case also, performances were not an issue: most of the computation involved when building the state space was delegated to a library implemented in C. The computation performed by Python was small enough to allow us to run our examples and validate our algorithms. However, our implementation was only a prototype and to make it more efficient, one would need to replace SNAKES by a fast Petri net library, which would fix the variant used.

1.2 Related tools

Several existing tools may be related in particular to SNAKES. The first one is the *Petri net kernel* (PNK) [29] that shares with SNAKES the aim to provide a general framework for building Petri nets applications. The PNK provides a graphical user interface for editing and simulating Petri nets, its main aim being to provide the basis to real applications rather than to allow for quick prototyping. With respect to SNAKES, the basic model of the PNK is a less general model

of coloured Petri nets; however, this may be extended by writing Java code. Another difference is that the PNK does not provide any of the operations in order to manage models from the PBC and M-nets family, which is of course not its aim. The development of the PNK does not appear to be active anymore, the last release being dated of March 2002. The PNK is free software distributed under the terms of the GNU GPL, which forces tools that use the PNK to be released under the same licence. SNAKES uses the GNU LGPL, which is less restrictive and allows to produce non-free software that uses SNAKES, but forces to release under the GNU LGPL any change made to SNAKES.

Another tool with which SNAKES shares some goals is the *Programming Environment based on Petri nets* (PEP) [23]. The main similarity between the two tools is that both deal with PBC and M-net models. The main difference is that PEP is oriented toward model-checking, proposing a graphical user interface to model Petri nets through various ways. The Petri nets models in PEP are fixed, mainly variants of PBC and a restricted version of M-nets, which cannot be changed by the user. PEP also appears to be not maintained anymore, the last release being dated of September 2004. PEP was the tool we used before to decide to develop SNAKES. The main reason for this decision was the impossibility to update PEP quickly enough with respect to the theoretical developments in the PBC family, which is not surprising as PEP was never designed with this goal in mind. Like the PNK, PEP is released under the GNU GPL.

Compared with *CPN tools* [11], SNAKES shares the ability to use a programming language for the inscriptions of the nets: a variant of ML for CPN tools, and Python for SNAKES. This makes it possible to extend a lot the features of CPN tools. However, it uses fixed (but very general) Petri net models and does not provide support to introduce another variant. So, there might be new models of Petri nets that cannot be represented using one of those provided by CPN tools, and thus for which the tools cannot be used. Another important difference is that CPN tools feature an advanced graphical user interface while SNAKES is only a programming library. Finally, CPN tools is not open source.

More generally, as any Petri net modelled in SNAKES may be executed, it could be compared with any Petri net tool that can simulate nets or compute their reachability graph. However, executing nets is not the main purpose of SNAKES and it is not designed to do it efficiently (even if it may be efficient enough for simulation). This feature is only required because using SNAKES for prototyping may require executing Petri nets. For instance, when working on a new model of Petri nets equipped with unbounded integer variables [27], the fact that SNAKES could execute this new class of net was necessary to prototype our state graph construction (see the section 3.2.3 for details).

1.3 Outlines

The next section gives an overview of the core library. Then we present the plugin system implemented in SNAKES. We conclude the paper with a list of ongoing and future works.

2. THE CORE LIBRARY

SNAKES is organised as a hierarchy of modules:

snakes is the top-level module and defines the exceptions used in the library.

snakes.data defines the basic data types (*e.g.*, multisets and substitutions) and data manipulation functions (*e.g.*, cross product).

snakes.typing defines a typing system that can be used to restrict the tokens accepted by a place (see the section 2.2.1).

snakes.nets defines all the classes directly related to Petri nets: places, transitions, arcs, nets, markings, reachability graphs, etc. A simplified class diagram of this module is presented in the top of the figure 3. It also imports and makes available all the API from the modules above.

snakes.plugins is the root for all the extension modules of SNAKES.

The first four modules above (plus additional internal ones not listed here) form the *core library* of SNAKES which is described further in the rest of the section. (The plugin system will be described in the next section.)

SNAKES is designed so that it can represent Petri nets in a very general fashion:

- each transition has a guard that can be an arbitrary Python Boolean expression;
- each place has a type that can be an arbitrary Python Boolean function that is used to accept or refuse tokens;
- tokens may be arbitrary Python objects;
- input arcs (*i.e.*, from places to transitions) can be labelled by values (to consume a known value), variables (to bind a token to a variable name) or several of these objects (to consume several tokens). New kind of arcs may be added (*e.g.*, read arcs are provided as a simple extension of existing arcs);
- output arcs (*i.e.*, from transitions to places) can be labelled the same way as input arcs, moreover, they can be labelled by arbitrary Python expressions in order to compute new values to be produced;
- a Petri net with these annotations is fully executable, the transition rule being that of coloured nets: a binding of variables must be found such that there are enough tokens in input places and the guard of the transition is respected as well as the type of the output places.

More precisely, at any marking, each transition can compute its enabling bindings (also called its modes) as follows:

- each combination of the available tokens with the variables on the input arcs provides a possible binding of these variables;

- each such binding corresponds to a Python environment (*i.e.*, a set of names associated to values) in which the guard of the transition is evaluated;
- if the guard evaluates to True, each output arc is then evaluated in the same environment and it is checked if the produced tokens are accepted by the corresponding places;
- if all these tests pass successfully, the binding is enabling.

Then, one of these enabling bindings can be used to fire the transition, which follows the same process starting from the second step and ending by actually consuming and producing the adequate tokens.

2.1 Example

A simple example of a coloured Petri net is depicted in the figure 1. In this example, place p_1 can be marked by any integer-valued token (it currently holds two such tokens) and place p_2 is restricted to non-negative integers. In order to model this net with SNAKES, one may run the following Python program:

```
1 from snakes.nets import *
2 n = PetriNet('simple net')
3 n.add_place(Place('p1', [−1, 2], tInteger))
4 n.add_place(Place('p2', [], tNatural))
5 n.add_transition(Transition('t', Expression('x>0')))
6 n.add_input('p1', 't', Variable('x'))
7 n.add_output('p2', 't', Expression('x+1'))
```

The first line imports the main module. It exports in particular the classes PetriNet, Place, Transition, Expression and Variable, and the objects tInteger and tNatural. Then, a Petri net is created, being given the name "simple net". Two places are added to it, each is an instance of the class Place whose constructor expects the name of the place, a list of tokens for its marking and an optional constraint on the accepted tokens (see the section 2.2.1). Similarly, a transition is added, being given a name and an optional Boolean expression for its guard. Finally, two arcs are created: one input arc labelled by a variable and one output arc labelled by an expression.

At the end of this program, various objects have been created, which is summarised in the figure 2. We see on this diagram that a new class appeared: instances of MultiSet (from the module snakes.data) are used to represent the marking of the places. Moreover, the attributes pre and post of places and transitions are dictionaries whose keys are node names and whose values are arc annotations. For instance p1.post['t'] is the variable x, which is used this way on the diagram (instead of depicting the dictionary objects).

In order to get the list of enabling bindings for the transition, one may use the following:

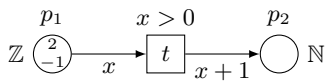

Figure 1: A simple coloured Petri net.

```
8 t = n.transition('t')
9 m = t.modes()
```

(The first line stores a reference to the transition in a variable t in order to avoid using "n.transition('t')" everywhere.) At this point, the value of m is the list [Substitution(x=2)] because only the binding $\{x \mapsto 2\}$ enables t. Then, the transition may be fired with the first binding discovered (if t had no enabling binding, this last statement would result in an exception as m would be an empty list):

```
10 t.fire(m[0])
```

A class StateGraph is provided in order to automate this process and to compute the reachability graph by executing all the possible transitions for all the possible modes at all the reached markings. The following code creates the StateGraph object, computes all the reachable markings and then iterates over the states in order to print their information (marking, successors and predecessors).

```
9  g = StateGraph(n)
10 g.build()
11 for s in g :
12     print 'state', s, 'is', g.net.get_marking()
13     print '  successors:', g.successors()
14     print '  predecessors:', g.predecessors()
```

Executing this code after the line 7 above prints the following (except that lines have been wrapped below) where each arc in the marking graph is labelled by the corresponding transition and its mode at firing time:

```
state 0 is Marking({'p1': MultiSet([2, -1])})
  successors: {1: (Transition('t', Expression('x>0')),
                   Substitution(x=2))}
  predecessors: {}
state 1 is Marking({'p2': MultiSet([3]),
                    'p1': MultiSet([-1])})
  successors: {}
  predecessors: {0: (Transition('t', Expression('x>0')),
                     Substitution(x=2))}
```

2.2 Other features

2.2.1 *Type system for the places*

As seen above, places in a Petri net are given a type that is used to control the accepted tokens. We have used the types tInteger and tNatural from the module snakes.typing. This module actually provides a more general type system that one can use to build complex type checkers for places. In this system, a type is understood as a set of values, a type checker being a test that decides whether a given value belongs to the type or not.

Several type constructors are provided in order to build basic types:

Instances(c) builds a type whose elements are instances of the class c.

OneOf(a, b, ...) creates a type whose values are just those enumerated, *i.e.*, a, b, etc.

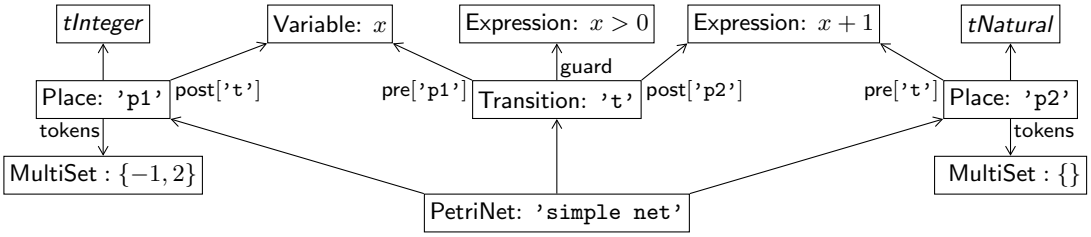

Figure 2: The objects diagram after executing the line 7 of the program. Some links indicate the names of the attributes that hold the references.

Collection(container, items, min, max) creates a type for collections of objects whose values are objects in the type container (usually list or tuple) and contain at least min and at most max values accepted by the type items. There is similarly a type constructor Mapping for dictionary-like objects.

Range(first, last, step) returns a type that accepts all the values ranging from first to last by steps of step.

Greater(min) accepts all the values greater than min. Similarly, there are type constructors GreaterOrEqual, Less and LessOrEqual.

CrossProduct(t1, t2, ...) creates a type that accepts tuples of values from the given types t1, t2, ...

TypeCheck(fun) creates a type whose values are those for which the function fun returns True. This allows to build a type from an arbitrary Boolean function.

On this basis, types may be combined using various sets operators: & (intersection), | (union), − (difference), ^ (disjoint union) and ∼ (complement). For instance, the module snakes.typing defines:

```
tInteger = Instance(int)
tNatural = tInteger & GreaterOrEqual(0)
```

2.2.2 Arcs

We have seen so far that arcs may be labelled by values, variables or expressions (only on output arcs). It is also possible to create multi-arcs that transport multiple values. For instance, MultiArc([Value(1), Variable('x')]) can be the label of an input arc which is able to consume two tokens, one being the value 1 and the other being an arbitrary value bound to the variable x.

SNAKES also provides test arcs that never transport values. On an input arc, this corresponds to a read arc; on an output arc, it is used to check the type of a place with respect to the annotation. For instance, creating an output arc with the label Test(Expression('x**2')) will never produce a token in the corresponding place but will allow the transition to check if the place type accepts the value computed from the expression.

New kind of arcs may be created, it is only necessary to derive a class from the abstract class ArcAnnotation.[2] Moreover, like Test or MultiArc the new class may encapsulate existing arc classes.

[2]There is no proper notion of abstract class in Python, however, this can be simulated using a class whose methods that should be abstract raise NotImplementedError if called.

2.2.3 Support for the Petri Net Markup Language

Every object that may be part of a Petri net in SNAKES can be exported to or imported from PNML. The required module for this purpose is snakes.pnml. It provides in particular a function dumps that takes an object as argument and returns its representation in PNML. The module also provides the function loads that does the reverse, *i.e.*, building an object from its PNML representation.

When an object has no known PNML representation, SNAKES uses standard Python serialisation (embedded in XML). This is convenient because any object can be saved to PNML; but in such a case, there is no chance for the produced PNML to be compatible with an other tool.

2.2.4 Controlling the execution environment

It has been explained above how a transition binds the variables on its input arcs in order to build an environment that is used to evaluate the Python expressions in the guard and output arcs. When one of these expressions needs functions or modules that are not available in the default environment, the evaluation fails. In such a case, it is necessary to declare the needed objects before to start executing transitions. There are two ways to do so.

One is to use the method declare of a PetriNet instance that expects an arbitrary Python statement given as a string and executes it. If this statement has some side effects, this will be recorded for the next evaluation. For instance, one may run n.declare('import␣math') in order to make the module math available to all the evaluations occurring in the net n after that.

The other solution is to access directly the evaluation environment that is a dictionary stored as an attribute globals. This attribute exists for any object that needs to evaluate Python code and is shared over a whole Petri net. For instance, in order to declare a global variable x, the method described above should be used as "n.declare('global␣x;x=2')" (first state that x is global and then assign it) or more simply, using the attribute globals: "n.globals['x']=2".

3. EXTENSION MODULES

An extension module is meant to extend an existing module from the core library (usually snakes.nets) by subclassing some of its classes. Because we do not know in advance

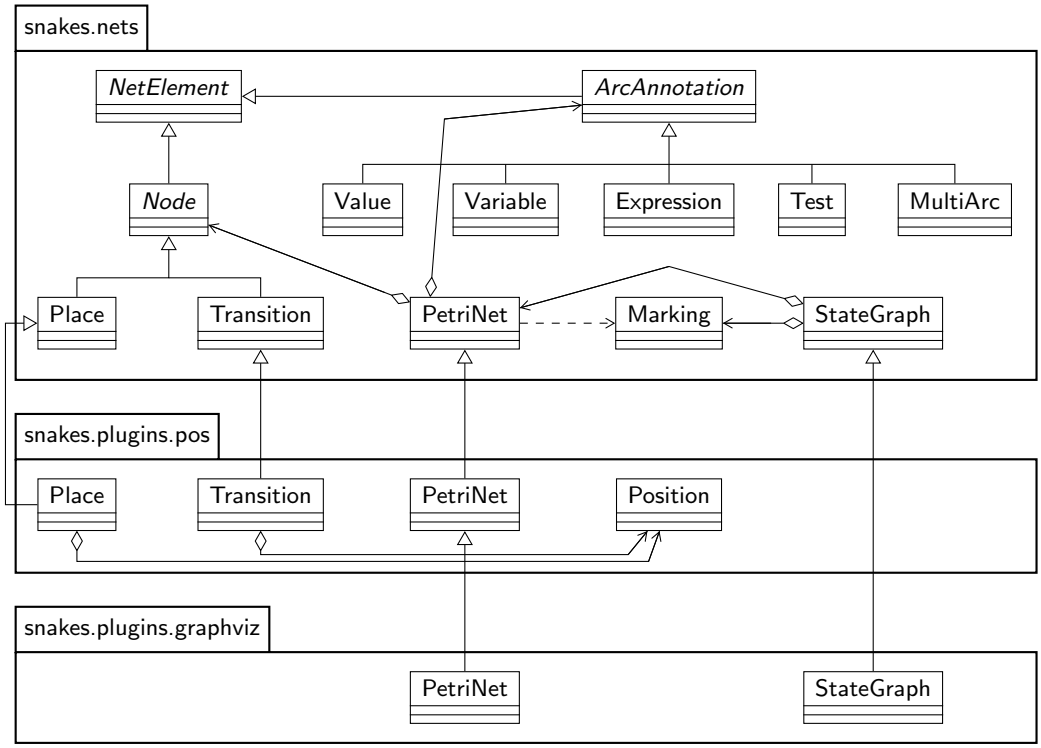

Figure 3: The loading of the plugin graphviz, that depends on pos, on the top of snakes.nets.

which plugins will be loaded by the user and in which order, the classes hierarchy cannot be fixed statically. In order to do it dynamically, an extension module provides a function extend that takes as its single argument the module to extend, which may be snakes.nets or a version of it already extended, and returns a new module with proper subclasses. (Python allows to create classes at run time.) The module snakes.plugins provides some functions to make this easy, allowing the programmer to concentrate on writing the subclasses. This approach is summarised in the figure 3 that illustrate the loading of the plugin graphviz on the top of the plugin pos (on which it depends). First, the plugin pos is loaded on the top of the module snakes.nets, which transparently calls snakes.plugins.pos.extend(snakes.nets). This returns a new module whose classes are those from snakes.nets except for Place, Transition and PetriNet that come from snakes.plugins.pos, with the inheritances depicted in the figure 3. Then, on the top of this new module, the plugin graphviz is loaded, following a similar process. The result is a module that exhibits the same interface as snakes.nets and whose classes come from snakes.plugins.graphviz for PetriNet, snakes.plugins.pos for Place and Transition and snakes.nets for the other ones.

The loading of plugins is supported by helper functions, so, the example described above would simply result in the following code:

```
import snakes.plugins
snakes.plugins.load('graphviz', 'snakes.nets', 'nets')
from nets import *
```

The first statement is a regular Python module import. The second statement is the loading of the plugin: the extension module called `'graphviz'` is loaded on the top of the module called `'snakes.nets'` and the result is imported as the module called `'nets'`. This allows to execute the last statement that imports every visible name from the newly constructed module nets, thus avoiding to use the prefix "nets." in order to access its content.

Several plugins may be loaded using a single load statement, in order to do so, it is just needed to give a list of plugin names to be loaded instead of only one. For instance:

```
snakes.plugins.load(['graphviz', 'ops', 'synchro'],
                    'snakes.nets', 'nets')
```

3.1 Defining new Petri net variants

A new class of Petri nets can be easily defined using a plugin. Let's consider for example Merlin & Farber's time Petri nets [21]. A plugin is being developed to support them in SNAKES. In its current state, it extends classes Transition, Place, and PetriNet. In order to simplify the presentation, the version presented here does not support transitions with multiple enabling, it is discussed below how this is supported in the actual implementation.

First, each transition is given a earliest and latest firing time as well as a timer (*i.e.*, its current time value). The constructor of class Transition adds an attribute time for the timer, and two attributes min_time and max_time initialised from arguments added to the constructor. The default value for max_time is None, which is considered as an infinite boundary. Then, the method enabled (that checks whether a binding enables or not a transition) is redefined in order to take time into account. It accepts an additional optional argu-

ment untimed that allows to avoid checking time boundaries. The method copy that duplicates a transition is also redefined in order to properly copy timing information.

Next, class Place is extended so that whenever the marking of a place is changed, its successor transitions are examined in order to reset their timer if their enabling is changed. This implies to redefine four methods: add that adds tokens to a place, remove that removes tokens from a place, reset that replaces the marking of a place, and empty that removes all the tokens held by a place. In all cases, when a transition is newly enabled its timer is reset to zero. When a transition becomes disabled by its input marking, its timer is set to None, which avoids to consider it when time passes. As a result, the value None for a timer indicates that the transition is not enabled because of the marking, but when the timer is not None, its value has to be compared with the earliest and latest firing time of the transition to know if it is enabled or not. Thus, the method Transition.enabled is written as follows:

```
1  def enabled (self, untimed=False) :
2      if self.time is None :
3          return False
4      elif untimed :
5          return super(Transition, self).enabled(binding)
6      elif self.max_time is None :
7          return (self.min_time <= self.time) \
8              and super(Transition, self).enabled(binding)
9      else :
10         return (self.min_time <= self.time) \
11             and (self.time <= self.max_time) \
12             and super(Transition, self).enabled(binding)
```

The condition on the line 2 checks if the timer is None, in which case the transition is disabled because of the marking. Otherwise, the condition on the line 4 checks whether the new argument untimed has been set to True when calling the method. If so, the enabling is tested by the parent class super(Transition, self), thus ignoring all timing information. Then, the line 6 corresponds to the case where no latest firing time has been given and the **else** case when it has been given. The assumption that a transition is not enabled then its timer is None is safe because this value is set by the preplaces of each transition when their markings are changed. For instance, the methods Place.add and Place.remove are programmed as follows:

```
def add (self, tokens) :
    enabled = self._post_enabled()
    super(Place, self).add(tokens)
    for name in self.post :
        if not enabled[name] :
            trans = self.net.transition(name)
            if len(trans.modes()) > 0 :
                trans.time = 0.0
def remove (self, tokens) :
    enabled = self._post_enabled()
    super(Place, self).remove(tokens)
    for name in self.post :
        if enabled[name] :
            trans = self.net.transition(name)
            if len(trans.modes()) == 0 :
                trans.time = None
```

The method _post_enabled returns a dictionary whose keys are the names of the transitions in the post-set of a place, associated to Boolean values indicating whether each transition is currently enabled or not (which is checked by comparing its timer to None). Then, after adding the tokens line 3, the method add checks if a transition becomes enabled by its marking, *i.e.*, if at least one mode can be found for it. If so, its timer is set to 0.0. Symmetrically, the method remove checks if a transition that was enabled becomes disabled, in which case its timer is set to None. In order to lift the implementation to the case where transition can be multiply enabled, it is enough to store one timer for each mode of each transition, which is quite a simple change with respect to the simplified implementation presented here.

Finally, the class PetriNet is given two new methods step and time. The former computes the maximal delay that can pass until the enabling is changed, either by enabling a transition (when its timer reaches its min_time), or by requiring a transition to fire (when its timer reaches its max_time). The latter can be used to let time pass, which corresponds to increase all the timers that are not None. This method time expects a duration as argument and returns the actual duration that could be used. For instance, if the user request an increasing of 1.0 but if after 0.4 time units a transition becomes newly enabled, then the method will only increase time by 0.4 and return this value to inform the user. Similarly, time will never be increased enough to overcome the latest firing time of a transition. So, when a transition has to fire because of time, any call to time or step will return 0.0, corresponding to the fact that time cannot pass before the urgent transition is fired.

These changes have been implemented in less than 100 lines of code. In order to have a complete plugin, it will be necessary to implement an extended StateGraph class to construct a valid state space for time Petri nets (*e.g.*, one of those described in [3]). Support for transitions with multiple enabling as described above will be included as well. When completed, the plugin will be included in SNAKES.

3.2 Main available plugins

3.2.1 Drawing nets and state graphs

The plugin graphviz used above in the example is dedicated to draw PetriNet and StateGraph objects using GraphViz [1]. There are two ways to layout a Petri net: either by giving an explicit position to each node,[3] or by letting GraphViz compute a layout using one of the five available engines. Only the latter solution is available for a state graph. The figure 4 shows examples of state graphs drawn by GraphViz.

3.2.2 PBC and M-nets operations

The PBC and M-nets models define two kinds of operations, relying respectively on a labelling of places or transitions.

First comes a family of control flow compositions for which places are given statuses indicating their roles. In particular, one distinguishes entry places and exit places that correspond to the initial and final markings of a net. For instance, a sequential composition of two nets consists in combining the exit places of the first net with the entry places of the second net in such a way that the termination of the former

[3]This feature is provided by the plugin pos on which the plugin graphviz depends.

will correspond to the starting of the latter. Moreover, statuses distinguish data places that are given a name, which corresponds to the variables in a program. When nets are composed, data places with the same name are merged in order to ensure a unique representation of each variable.

Places statuses are implemented in the extension module status, then the extension module ops relies on it in order to implement the usual PBC control flow operations (with automatic merging of data places): sequence, iteration, choice and parallel. A name hiding operation is also provided, it removes the name of a data place so that it cannot be merged anymore, which corresponds to a local variable. All these operations are implemented as Python operators, so, for instance, if n1 and n2 are two PetriNet objects, one may write:

```
p = n1 | n2   # parallel composition
s = n1 & n2  # sequential composition
c = n1 + n2  # choice
i = n1 * n2   # iteration
h = n1 / 'var'  # hiding of name 'var'
```

The other set of operations comprises the transitions synchronisation, restriction and scoping. They are inspired from the CCS [22] action synchronisation as transitions can synchronise on conjugated actions. However, with respect to CCS, PBC and M-nets use multi-actions, allowing more than two transitions to synchronise at the same time. Moreover, these models distinguish the synchronisation that enables the synchronised behaviour, from the restriction that forbids the independent behaviour (the scoping is the successive application of both operations). Finally, these operations are purely statical and construct explicitly the synchronised transitions that may fire or not at execution time. With respect to PBC, M-nets also define parameters for the actions, allowing to exchange information between the synchronised transitions.

This aspect is implemented in the plugin synchro that defines classes for actions and multi-actions and adds methods to the class PetriNet in order to perform the corresponding operations. SNAKES generalises the models by not imposing a fixed number of parameters for each action name. Instead, matching the number of parameters becomes a part of the unification process that takes place when two conjugated actions participate in a synchronisation.

The M-nets model also includes a general refinement operation that allows to replace a transition with an arbitrary M-net [13]. This operation has not been implemented in SNAKES and it is not intended to add it since it leads to very complex nets that are not tractable in practice. For instance, place types, tokens and arc annotations become trees after a refinement, so firing a transition implies matching trees against trees. Moreover, the refinement is always used for two purposes: synthesis of the control flow operations (sequence, choice, etc.), and colour-safe execution of multiple instances of the same net. With the new models of the family, both these effects are now feasible without using the general refinement, see [26] for instance.

3.2.3 Handling unbounded counters

The plugin lashdata has been developed while working on a model of P/T Petri nets equipped with unbounded integer variables [27]. This model has been given a semantics in terms of compact state graphs where one abstract state encodes possibly infinitely many concrete states that differ only by the values of the integers variables. The fact that SNAKES was available and allowed to implement this model has been a benefit at several points. First, working on the theoretical definitions and implementing them in parallel helped a lot to clarify and simplify the definitions. Then, possible optimisations were discovered during the implementation of the compact state graph construction. Indeed, going to the detailed program level allowed to identify where programming choices had to be made and thus to investigate the consequences of different choices. Moreover, running the prototype on various examples allowed to exhibit cases where the current construction were not satisfactory, which led to improve the algorithm at the theoretical level. Another source of satisfaction was the ability to produce automatically examples for illustrating the paper and the presentation, which was not only convenient but also increased the confidence that no mistake was introduced in the examples. Finally, the fact that a prototype existed for the construction described in the paper was perceived as very positive by the referees as well as by the audience when the paper has been presented.

The plugin lashdata relies on the library Lash [7] to represent integers variables added to a Petri net. (With respect to the paper [27], this plugin allows for any kind of Petri net and not only for P/T nets.) The plugin defines a class Data that encapsulates the data structures of Lash in order to store the values of the variables. For instance, the creation of a Petri net equipped with two variables x and y initialised to zero requires the following Python code:

```
import snakes.plugins
snakes.plugins.load('lashdata', 'snakes.nets', 'nets')
from nets import *
n = PetriNet('N', lash=Data(x=0, y=0))
```

Then, the plugin extends transitions in order to take the integer variables into account: the firing is subject to a condition on the variables (which is independent of the guard) and it is allowed to update their values. For instance one may write:

```
n.add_transition(Transition('t'),
                 condition='x<y',
                 update='x=x+1; y=y-1')
```

Finally, the class StateGraph is extended in order to implement the construction defined in the paper: several options are added to the constructor in order to enable various levels of compression. With no option, no compression is performed and the abstract state graph corresponds to the concrete reachability graph. Using the option "loops=True" enables compression when a side-loop is detected (*i.e.*, a transition that changes the variables but not the marking). Using "cycles=True" also enables the compression when general loops are detected (*i.e.*, cycles in the state graph). Using "remove=True" then enables the removing of covered states (*i.e.*, existing abstract states that are included in newly computed ones). Finally, using "fold=True" adds additional compression when sequences of the same transition are detected. This last option is not described in [27] and was added to the

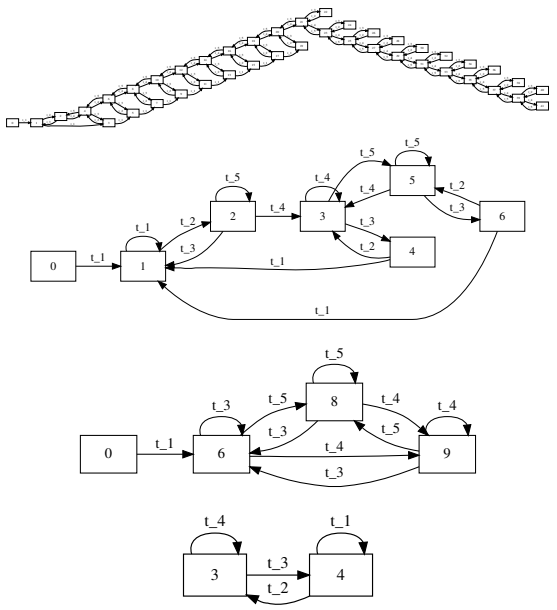

Figure 4: On the top: a concrete marking graph. Below: its compact versions when the various compression options are activated (top-down: loops, cycles or remove, and fold). On this example, the option remove does not provide more compression than cycles.

plugin after the publication. The figure 4 shows an example of a state graph at the different levels of compression.

One hidden but important task of the plugin lashdata is to perform the translation between the variable-based representation of data in the Petri nets and its vector-based implementation in Lash. Indeed, Lash actually handles sets of integer-vectors and matrix-based linear conditions and updates. The plugin thus assigns to each variable an index in such a vector and computes for each condition or update expressed in Python (with constraints on the syntax in order to ensure the linearity) the corresponding matrices as expected by Lash. This work is facilitated by the module snakes.compyler that was designed to handle Python expressions in SNAKES. For instance, it is used to correctly rename variables in expressions (*e.g.*, in guards); it may be used also to translate Python expression to another language like C (provided in snakes.compyler for illustration) or CPN-ML. To do so, the module compyler defines a class Parser that uses the standard Python library to parse Python code and build abstract syntax trees. This trees are then traversed and each node is examined by a dedicated method of the class Parser. By subclassing it, one may perform any operation on any node of the abstract trees.

4. FUTURE AND ONGOING WORKS

SNAKES is still under active development and is still considered as a beta software at its current version (0.8.8). It is planned to release the first stable version (numbered 1.0) after a few features will be implemented (flagged "⋆" below) and after some positive feedback will be received from the users.

⋆ *Improving the documentation.* The current documentation for SNAKES is composed of an API reference manual and a tutorial. The former is automatically generated from the comments in the source code (Python "docstrings") that also contains the code used for unit testing. There are currently some modules, classes, methods or functions that lack a proper documentation with sensible examples and unit-test, which must be fixed in order to improve the usability. However, the core library is currently quite complete to this respect: 87% is documented, 47% has unit test and 49% has detailed API specification. The tutorial is written separately and currently does not introduce all the plugins of SNAKES, but its is quite complete about the core library.

⋆ *Better PNML support.* The current support of PNML is not well tested, in particular for import. Moreover, many extension modules do not properly support PNML export or import. For the first stable release, it will be necessary to check that every (reasonable) Petri net in SNAKES can be exported to a PNML code that is compatible with the major tools available. Moreover, it will be necessary to check that import from these tools is possible for reasonable cases (*i.e.*, classes of Petri nets that naturally map to SNAKES core library).

Support for other formats. Apart for PNML, other file formats should be supported by SNAKES. In particular, those of various model checkers that do not support PNML. The idea is that SNAKES can be used to build a model taking advantage of the PBC and M-nets compositions, then this model can be model-checked with a specialised tool. This work requires first a survey of the existing tools to see which ones do not support PNML and which formats are already supported by some conversion tool. The goal is to minimise the number of output formats to add to SNAKES in order to avoid to duplicate existing tools. For instance, PEP or the model-checking kit [14] provide quite a lot of such translation tools.

There may exist also formats for which it may be interesting to have an input filter to SNAKES, but this will not be a priority as SNAKES is not intended to be used at the end of a tool chain but rather at the beginning.

Support for more Petri nets variants. From the beginning, SNAKES has been designed to be able to support as many variants of Petri nets as possible. This aim should be turned into acts by providing extension modules for popular Petri net models, in particular, those with time (see [10] for a comprehensive survey), stochastic Petri nets [2] and object Petri nets [17]. As presented above, Merlin & Farber's time Petri nets are being implemented, which causes no particular difficulty.

API to other programming languages. As every library, SNAKES is bound to a particular programming languages, Python in its case. This may be a limitation to its usage, even if Python is a language that is very easy to learn. In

particular, this is a limitation for a programmer that would like to use SNAKES as the back-end of another application not written in Python. For instance, a graphical editor could use SNAKES as its data model in order to provide simulation capabilities.

In order to remove this limitation, a C binding of SNAKES is being developed. The approach chosen is to generate it automatically by inspecting the actual Python code. Wrappers for each class, method and function are generated as Pyrex [16] code. This is a dialect of Python mixed with C that can be compiled to pure C code. It is primarily intended for building Python extension but is also suitable for embedding Python into C programs.

Then, with a C API available, many other languages can be mapped. Thin bindings can be easily obtained for a variety of languages using a tool like SWIG [12]. But it is often more interesting to produce a thick binding that is oriented toward the programming style enforced by a particular language. For instance, a Java binding should be object-oriented as the original API is. (Note that there is no a priory limitation with Java that can be interfaced to native libraries using the JNI framework [30].)

5. ACKNOWLEDGEMENTS

The author would like to thank Hanna Klaudel for her good advices about how to present this paper, and Emmanuel Polonowski who helped with the UML notation.

6. REFERENCES

[1] AT&T Research. Graphviz, graph visualization software. ⟨`http://www.graphviz.org`⟩.

[2] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets (2nd Edition)*. Vieweg Verlag, 2002.

[3] B. Berthomieu and F. Vernadat. State space abstractions for time Petri nets. *Handbook of Real Time Systems*, to appear, 2006.

[4] E. Best, R. Devillers, and J. Hall. The Petri box calculus: a new causal algebra with multilabel communication. In *Advances in Petri Nets 1992*, volume 609 of *LNCS*. Springer-Verlag, 1992.

[5] E. Best, R. Devillers, and M. Koutny. *Petri net algebra*. Springer-Verlag, 2001.

[6] E. Best, W. Fraczak, R. P. Hopkins, H. Klaudel, and E. Pelz. M-nets: an algebra of high-level Petri nets with an applications to the semantics of concurrent programming languages. *Acta Informatica*, 35(10), 1998.

[7] B. Boigelot. The Liège automata-based symbolic handler. ⟨`http://www.montefiore.ulg.ac.be/~boigelot/research/lash`⟩.

[8] R. Bouroulet, H. Klaudel, and E. Pelz. A semantics of security protocol language (SPL) using a class of composable high-level Petri nets. In *Proc. of ACSD'04*. IEEE Computer Society, 2004.

[9] C. Bui Thanh. Generating coloured Petri nets of concurrent object-oriented programs. In *Proc. of ESMc'04*. EUROSIS, 2004.

[10] A. Cerone and A. Maggiolo-Schettini. Time-based expressivity of time Petri nets for system specification. *Theoretical Computer Science*, 216(1-2), 1999.

[11] CPN Group, University of Aarhus. CPN Tools. ⟨`http://wiki.daimi.au.dk/cpntools`⟩.

[12] D. Beazley & al. Simplified wrapper and interface generator. ⟨`http://www.swig.org`⟩.

[13] R. Devillers, H. Klaudel, and R.-C. Riemann. General parameterised refinement and recursion for the M-net calculus. *Theoretical Computer Science*, 300(1-3), 2003.

[14] J. Esparza, C. Schröter, and S. Schwoon. The model-checking kit. ⟨`http://www.fmi.uni-stuttgart.de/szs/tools/mckit`⟩.

[15] S. Evangelista. High level petri nets analysis with Helena. In *Proc. of ICATPN'05*, volume 3536 of *LNCS*. Springer-Verlag, 2005.

[16] G. Ewing. Pyrex, a language for writing Python extension modules. ⟨`http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex`⟩.

[17] B. Farwer and K. Misra. Modelling with hierarchical object Petri nets. *Fundamenta Informaticae*, 55(2), 2003.

[18] A. Finkel. The minimal coverability graph for Petri nets. In *Papers from the 12th International Conference on Applications and Theory of Petri Nets*. Springer-Verlag, 1993.

[19] F. Heitmann. Petri nets tool database at the Petri net world. ⟨`http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools`⟩.

[20] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[21] P. M. Merlin and D. J. Farber. Recoverability of communication protocol. *IEEE Trans. on Communications*, 24(9), 1976.

[22] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[23] Parallel Systems Group, University of Oldenburg. The PEP tool. ⟨`http://peptool.sourceforge.net`⟩.

[24] E. Pelz and D. Tutsch. Formal models for multicast traffic in network on chip architectures with compositional high-level Petri nets. In *Proc. of ICATPN'07*, volume 4546 of *LNCS*. Springer-Verlag, 2007.

[25] F. Pommereau. Causal Time Calculus. In *Proc. of FORMATS'03*, volume 2791 of *LNCS*. Springer-Verlag, 2003.

[26] F. Pommereau. Versatile boxes: a multi-purpose algebra of high-level Petri nets. In *Prof of DASDS'04*. SCS/ACM, 2004.

[27] F. Pommereau, R. Devillers, and H. Klaudel. Efficient reachability graph representation of Petri nets with unbounded counters. In *Proc. of Infinity'07*, volume to appear of *ENTCS*. Elsevier, 2007.

[28] Python Software Foundation. Python programming language. ⟨`http://www.python.org`⟩.

[29] Research Group Petri Net Technology, Humboldt Universität of Berlin. The Petri net kernel. ⟨`http://www.informatik.hu-berlin.de/top/pnkn`⟩.

[30] Sun Microsystems, Inc. Java native interface. ⟨`http://java.sun.com/j2se/1.5.0/docs/guide/jni`⟩.