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, extension modules provide with enough flexibility to allow to redefine easily any part of the base Petri net model. In particular, SNAKES comes with a handful of extension modules in order to handle models of the Petri Box Calculus and M-nets family.
SNAKES is released under the GNU LGPL, it can be downloaded, together with API documentation and a tutorial, at ⟨http://lacl.univ-paris12.fr/pommereau/soft/snakes⟩.

Key words: Petri nets, quick prototyping.

1 Introduction

There exists a wide range of Petri net tools [24], 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 associated techniques evolve. When 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 time. But when research is centred on evolutions of the model itself, a tool often has a very short lifetime. It becomes then very hard for developers to keep the pace with 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 as follows:

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 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.
5. *Interoperable with other tools.* One tool cannot solve all the problems, so it is necessary that a toolkit can collaborate with other tools, in particular through importing/exporting PNML.

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

[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.

to extend the tool or redefine parts of it. This is the case for instance for text editor EMACS or 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 however, not many features are available. Then, Python [35] has been chosen as 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) [5, 6] and M-nets [7, 26] 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 verification of Petri nets equipped with unbounded integer variables [34]. This particular case is related in section 3.1 below where 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 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 [27], Finkel's version of Karp & Miller's coverability graph [22], M-nets unfolding to P/T-nets [7] and Petri net semantics of MINs [31]. 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. SNAKES has been used also in a teaching context, in order to introduce practical works in a theoretical course about Petri nets. The requirement was to model and verify a simple system, *e.g.*, an elevator or a simplified ATM. SNAKES has been used to build basic Petri nets, compose them and search their reachable markings for faulty states. In this experience also, students have had no problem using SNAKES.

1.1 Use cases and performance issues

SNAKES has been used to implement various Petri nets semantics:

- the Causal Time Calculus [32] 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 [10] 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 [31];
– a Petri net semantics of the Security Protocol Language (SPL) [9] has required also about 200 lines of code;
– a compiler for the Asynchronous Box Calculus with Data (ABCD) [11] has been implemented in SNAKES and is provided in the distribution (see section 5).

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 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 Helena [19] formalism for fast verification.

One other use case of SNAKES is developed more in details in section 3.1 and consisted in implementing a state space construction. In this case also, performances were not an issue: most of computation involved when building state space was delegated to a library implemented in C. Computation performed by Python was small enough to allow us to run our examples and validate our algorithms.

More generally, SNAKES should be efficient enough when used to build Petri net semantics of systems, or to perform interactive simulation. For more computationally intensive applications, several generic solutions may speed up SNAKES:

– *Psyco* [37] is a kind of just-in-time compiler for Python, it provides a typical speedup of 4;
– *Shed Skin* [17] allows to compile restricted Python to C++, which should allow to translate parts of SNAKES. Typical speedup is about 35;
– *PyPy* [12] is able to execute full Python with a just-in-time code generator, or to compile it to C. In the later case, typical speedup is 65.

These are easy to use but limited solutions. More generally, the approach in order to get a real application from a prototype using SNAKES would be either:

– delegate heavy computation to an external fast library (like presented in section 3.1), or,
– profile the prototype and implement a fast optimised version of critical parts as an external library, which allows to fallback to previous case.

It is very unlikely that a whole prototype has to be reimplemented because there will always exists parts of it that do not need a fast implementation. For instance, this is usually the case for import/export to PNML or other formats that is I/O bound rather than CPU bound.

1.2 Related tools

Several existing tools may be related in particular to SNAKES. The first one is the *Petri net kernel* (PNK) [36] 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 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 terms of the GNU GPL, which forces tools that use the PNK to be released under the same licence. SNAKES uses GNU LGPL, which is less restrictive and allows to produce non-free software that uses SNAKES, but forces to release under GNU LGPL any change made to SNAKES.

Another tool with which SNAKES shares some goals is the *Programming Environment based on Petri nets* (PEP) [30]. 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 users. 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 theoretical developments in the PBC family, which is not surprising since PEP was never designed with this goal in mind. Like the PNK, PEP is released under GNU GPL.

Compared with *CPN tools* [14], SNAKES shares the ability to use a programming language for nets inscriptions: 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 a programming library. Finally, CPN tools is not open source.

More generally, since 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 [34], the fact that SNAKES could execute this new class of net was necessary to prototype our state graph construction (see section 3.1 for details).

1.3 Outlines

The two next sections present SNAKES: its core library and its plugin system. Then we show three typical uses of SNAKES: writing a plugin to support a new variant of Petri net, writing a compiler for the Petri net semantics of a specification language, and using SNAKES as a service from another program. We conclude the paper with a list of ongoing and future works.

2 Core library

SNAKES is organised as a hierarchy of modules:

- snakes is the top-level module and defines exceptions used throughout the library;
- snakes.data defines 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 tokens accepted by a place (see section 2.2);
- 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 top of figure 3. It also exposes 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 this section. (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 that can be arbitrary Python object (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;

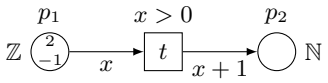

Fig. 1. A simple coloured Petri net.

– 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.

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 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 build this net within SNAKES, one may run the following Python code:

```
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'))
```

Line 1 imports the main module. It exposes in particular classes PetriNet, Place, Transition, Expression and Variable, and 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 class Place whose constructor expects the name of the place, a list of tokens for its marking and an optional constraint on accepted tokens (see section 2.2). 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 figure 2. We see on this diagram that a new class appeared: instances of MultiSet (from module snakes.data)

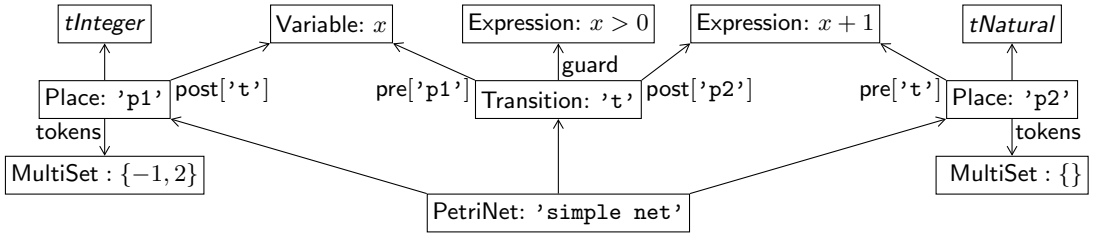

Fig. 2. Objects diagram after executing line 7 of the program. Some links indicate the names of the attributes that hold the references.

are used to represent places markings. Moreover, 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 variable x, which is denoted 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:

```
8  t = n.transition('t') # alternatively, one could use:
9  m = t.modes()         # m = n.transition('t').modes()
```

At this point, the value of m is list [Substitution(x=2)] because only 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).

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

Executing this code after line 7 above prints the following, 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

Type system for places. As seen above, places in a Petri net are given a type that is used to control accepted tokens. We have used types tInteger and tNatural from 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 class c.
- "OneOf(a, b, ...)" creates a type whose values are just those enumerated, *i.e.*, a, b, etc.
- "Collection(container, items, min, max)" creates a type for collections of objects whose values are objects in type container (usually list, set or tuple) and contain at least min and at most max values accepted by 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 (included) to last (excluded) by steps of step.
- "Greater(min)" accepts all the values greater than min. Similarly, there are type constructors Less, GreaterOrEqual and LessOrEqual.
- "CrossProduct(t1, t2, ...)" accepts tuples of values from the given types t1, t2, ...
- "TypeCheck(fun)" creates a type whose values are those for which 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, module snakes.typing defines:

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

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 value 1 and the other being an arbitrary value bound to 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 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 abstract class ArcAnnotation.[2] Moreover, like Test or MultiArc the new class may encapsulate existing arc classes.

Support for the Petri Net Markup Language. Every object in SNAKES can be exported to or imported from PNML. SNAKES provides a function dumps that takes an object as argument and returns its representation in PNML. It also provides a function loads that does the reverse, *i.e.*, building an object from its PNML representation.

The PNML standard is still quite unstable with respect to many extensions of Petri nets, in particular coloured Petri nets. So, the implementation of PNML import/export in SNAKES has been made very flexible in order to allow easy updates when new standards will be published. When an object has no standard PNML representation, SNAKES uses either its own XML representation for objects defined in SNAKES, or Python serialisation (embedded in XML) for unknown objects. This is convenient because any object can be saved to PNML; but in such a case, there is few chance for the produced PNML to be compatible with an other tool. However, when a net can be considered has a place/transition Petri net, SNAKES reads and produces PNML that is conform to standards, which has been tested compatible with other Petri net tools that support PNML.

Controlling Python execution environment. It has been explained above how a transition binds variables on its input arcs in order to build an environment that is used to evaluate Python expressions in its guard and output arcs. When one of these expressions needs functions or modules that are not available in 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 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 next evaluations. For instance, one may run n.declare('import␣math') in order to make module math available to all the evaluations occurring in net n after that.

The other solution is to access directly to 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 attribute globals: "n.globals['x']=2".

3 Extension modules

An extension module, or plugin, is meant to extend an existing module from the core library (usually snakes.nets) by subclassing some of its classes or by defining new classes or functions. Because we do not know in advance which plugins will be loaded by users and in which order, 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 and auxiliary material. (Python allows to create classes at run time.) Module snakes.plugins provides some functions to make this easy, allowing programmer to concentrate on writing the subclasses.

This approach is summarised in figure 3 that illustrate loading of plugin gv on the top of module snakes.nets. First, gv depends on plugin clusters, so, clusters is loaded on the top of module snakes.nets,

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

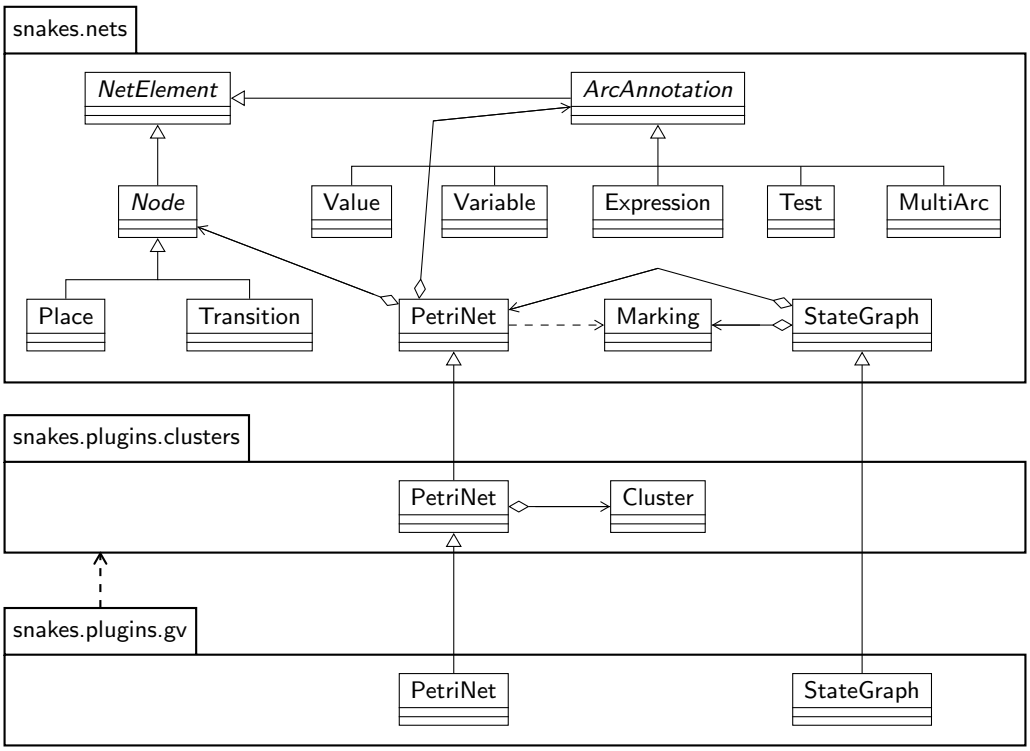

Fig. 3. Loading of plugin gv, that depends on clusters, on the top of snakes.nets.

which transparently calls snakes.plugins.clusters.extend(snakes.nets). This returns a new module whose classes are those from snakes.nets except for PetriNet that come from snakes.plugins.clusters, with inheritance relations as depicted in figure 3. This new module also has class Cluster from plugin clusters. Then, on the top of the new module, gv is loaded, following a similar process. The result is a module that exposes the same interface as snakes.nets with two changes:

- its classes PetriNet and StateGraph come from snakes.plugins.gv;
- it has a new class Cluster that comes from snakes.plugins.clusters.

Loading of plugins is supported by helper functions, so, the example described above would simply reduces to the following code:

```
1 import snakes.plugins
2 snakes.plugins.load('gv', 'snakes.nets', 'nets')
3 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 `'gv'` is loaded on the top of the module called `'snakes.nets'` and the result is imported as a 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 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:

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

3.1 Main available plugins

Drawing nets and state graphs. Plugin gv used above as illustration is dedicated to draw PetriNet and StateGraph objects using GraphViz [1] (see examples on figures 4 and 5). This plugin is a replacement of the former plugin graphviz. On the one hand, the new one introduces a dependency on Python module pygraphviz [23], but on the other hand, it provides much more control on drawing options and removes a call to an external program (which was both a security and a portability issue).

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 corresponds to the starting of the latter. Moreover, statuses distinguish data places that are given a name, which corresponds to 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 plugin status, then plugin 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 create 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 transitions synchronisation, restriction and scoping. They are inspired from CCS [29] action synchronisation since 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 synchronisation that enables synchronised behaviour, from restriction that forbids independent behaviour (scoping is the successive application of both operations). Finally, these operations are purely statical and construct explicitly synchronised transitions that may fire or not at execution time. With respect to PBC, M-nets also define parameters for actions, allowing exchange of information between synchronised transitions.

This aspect is implemented in plugin synchro that defines classes for actions and multi-actions and adds methods to 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 [16]. 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 control flow operations (sequence, choice, etc.), and colour-safe execution of multiple instances of the same net. With new models of the family, both these effects are now feasible without using the general refinement, see [33] for instance.

Handling unbounded counters. Plugin lashdata has been developed while working on a model of P/T Petri nets equipped with unbounded integer variables [34]. 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 theoretical definitions and implementing them in parallel helped a lot to clarify and simplify definitions. Then, possible optimisations were discovered during implementation. Indeed, going to detailed program level allowed to identify where programming choices had to be made and thus to investigate 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 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 confidence that no mistake was introduced in 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. An other work [25] about verification of multi-threaded systems modelled by coloured Petri nets has provided a similar experience.

Plugin lashdata relies on library Lash [8] to represent integers variables added to a Petri net. (With respect to [34], 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:

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

Then, the plugin extends transitions in order to take the integer variables into account: 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:

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

Finally, 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 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 compression when general loops are detected (*i.e.*, cycles in the marking 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 [34] and was added to the plugin after publication. Figure 4 shows an example of a state graph at different levels of compression.

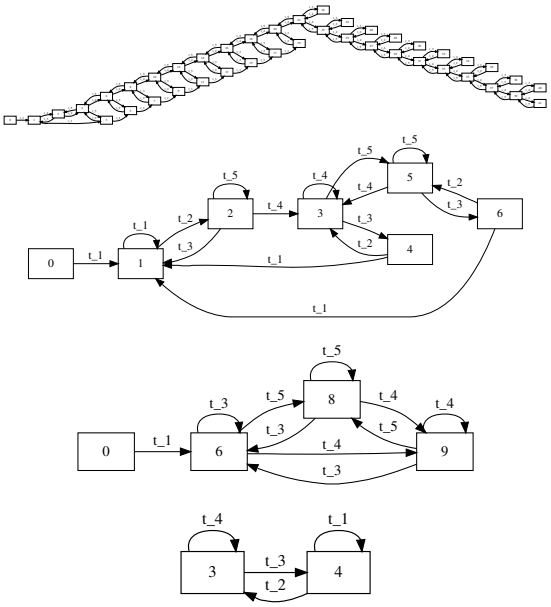

Fig. 4. On top: a concrete marking graph. Below: its compact versions when the various compression options are activated (top-down: loops, cycles or remove, and fold). On this example, option remove does not provide more compression than cycles.

One hidden but important task of plugin lashdata is to perform the translation between the variable-based representation of data in 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 syntactical constraints in order to ensure linearity) the corresponding matrices as expected by Lash. This work is facilitated by module snakes.compyler that was designed to handle Python code 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. More details about snakes.compyler is given in section 5 below.

4 Defining new Petri net variants

This section intends to illustrate how flexible is SNAKES with respect to the class of Petri net. For this purpose, support for a new class of Petri nets will be added using a plugin. We consider Merlin & Farber's time Petri nets [28]. 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 is simplified, in particular it does not support transitions with multiple enabling, it is discussed below how this is supported in the actual implementation.

First, each transition is given an earliest and latest firing time as well as a timer (*i.e.*, its current time value). 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, 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 argument untimed that allows to avoid checking time boundaries. 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 from 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, value None for a timer indicates that the transition is not enabled because of marking, but when the timer is not None, its value has to be compared with earliest and latest firing time of the transition to know if it is enabled or not. Thus, method Transition.enabled is written as follows:

```
1  def enabled (self, binding, 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) and super(Transition, self).enabled(binding)
8      else :
9          return (self.min_time <= self.time <= self.max_time) \
10             and super(Transition, self).enabled(binding)
```

The condition on line 2 checks if the timer is None, in which case the transition is disabled because of marking. Otherwise, the condition on line 4 checks whether the new argument untimed has been set to True when calling the method. If so, the enabling is tested using the method of parent class super(Transition, self), thus ignoring all timing information. Then, 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 when its timer is None is safe because this value is set by the pre-places of each transition when their markings are changed. For instance, methods Place.add and Place.remove are programmed as follows:

```
1  def add (self, tokens) :
2      enabled = self._post_enabled()
3      super(Place, self).add(tokens)
4      for name in self.post :
```

```
5            if not enabled[name] :
6                trans = self.net.transition(name)
7                if len(trans.modes()) > 0 :
8                    trans.time = 0.0
9    def remove (self, tokens) :
10       enabled = self._post_enabled()
11       super(Place, self).remove(tokens)
12       for name in self.post :
13           if enabled[name] :
14               trans = self.net.transition(name)
15               if len(trans.modes()) == 0 :
16                   trans.time = None
```

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, method add checks if a transition that was disabled becomes enabled by the new marking, *i.e.*, if at least one mode can be found for it (line 7). If so, its timer is set to 0.0. Symmetrically, 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 an easy change with respect to the simplified implementation presented here.

Finally, class PetriNet is given two new methods step and time. The former computes the maximal delay that can pass until the enabling changes, 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 a 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 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 features have been implemented in less than 100 lines of code, including support for transitions with multiple enabling as described above. 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 [4]).

5 Building a compiler for ABCD

A compiler for ABCD formalism has been first introduced in SNAKES to support a course about formal specifications and Petri nets semantics of programming languages. We present it here in order to illustrate how one can use module snakes.compyler to easily define the syntax and semantics of a formal language that may include fragment of Python code. snakes.compyler can be considered as an internal class of SNAKES core library because it is not exposed to end-users who deal with Petri nets only. However, it is worth knowing it is here and what it proposes because it can solve many concrete problems for one who wants to build a Petri net tool.

The syntax of ABCD defines atomic actions, enclosed in square brackets, that involve access to data in buffers and an optional condition. For instance, $[a{-}(x), b?(y), c{+}(x+y)$ if $x < y]$ is such an action that performs the following atomically:

- consume a value in a buffer a and bind it to variable x;
- test the presence of a value in a buffer b and bind it to y;
- send the result of $x + y$ in a buffer c;
- all this is guarded by condition $x < y$ and the process is blocked until the guard becomes true.

Two distinguished atomic actions that perform no data access are [True] that can be executed unconditionally and [False] that is a deadlocked process.

Processes can be composed using four binary control flow operators: "+" is a choice, ";" is a sequence, "|" is a parallel and "$*$" is an iteration (execute repeatedly first operand and exit by executing

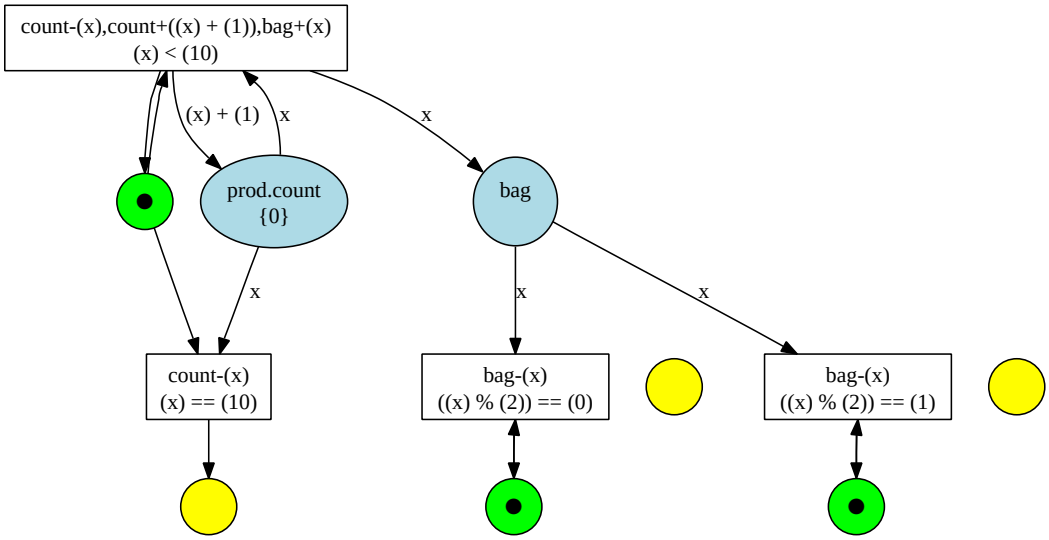

Fig. 5. Petri net semantics of 1-producer/2-consumers specification.

second operand). A sub-process may be declared as a "net" and reused later in a process expression. Finally, typed buffers may be declared globally to a specification or locally to net processes.

Below is a simple example of a 1-producer/2-consumers system specification. The producer puts in a buffer "bag" the integers ranging from 0 to 9. To do so, it uses a counter "count" that is repeatedly incremented until it reaches value 10, which allows to exit the loop. The first consumer consumes only odd values from the buffer, the second one consumes only even values. Both never stop looping.

```
1  def bag : int = ()  # buffer of integers declared empty
2
3  net prod :
4      def count : int = 0  # buffer of integers initialised with the single value 0
5      [count−(x), count+(x+1), bag+(x) if x < 10] ∗ [count−(x) if x == 10]
6
7  net odd :
8      [bag−(x) if (x % 2) == 1] ∗ [False]
9
10 net even :
11     [bag−(x) if (x % 2) == 0] ∗ [False]
12
13 prod | odd | even
```

The Petri net resulting from this specification is draw in figure 5. It is interesting to note that parts of this ABCD specification are actually Python code and could be arbitrarily complex:

- initial values of buffers ("()" and "0");
- buffer accesses parameters ("x" and "x+1");
- actions guards ("x<10", "(x%2)==1", ...).

In order to handle these parts, the ABCD compiler relies on module snakes.compyler that provides a series of tools to manipulate Python code. The module proposes two parsers: one is based on the native Python parser that is included in standard library, the other is a PLY [3] based parser that has been programmed independently. The former is fast and does not require maintaining if Python evolves, the latter can be extended and reused with much more flexibility. We will see here how it can be reused to parse ABCD specifications. The other tools in snakes.compyler are a set of classes to browse and transform abstract syntax trees (AST) returned by any of the parsers, and an structure of AST for constructs not included in Python, like those of ABCD. To observe the effect of using these tools, let us look at the guards of transitions in figure 5: they are not exactly those written is the specification. The reason is that they have been parsed, and their AST has been converted back to

text using one of the transformer classes called AstPrinter. Similarly, there exist a class AstRenamer that extends AstPrinter and allows for correctly renaming variables.

Extending the PLY parser of Python requires two steps: first, extend the lexer, then the parser. Indeed, we first need to add two tokens that do not exist in Python: "?" to parse buffer accesses like "$b?(y)$", and "net" to parse net definitions. To do so, only a few lines are necessary:

```
class Lexer (PlyLexer) :
    def t_QUESTION (self, t) :
        r"\?" # prefix 'r' indicates a "raw" string where '\' is a regular character
        return t
    def t_NET (self, t) :
        "net"
        return t
```

Then we extend the parser: we use a new start symbol and new rules, but reuse symbols from Python grammar when we need to parse Python code. A PLY parser is a class in which each method corresponds to a rule in the grammar, which is very convenient and easy to use. For instance, here are two rules for parsing an atomic action:

```
    def p_abcd_action_true (self, toks) :
        "abcd_action␣:␣LSQB␣abcd_access_list␣RSQB"
        toks[0] = Tree("action", toks[2], test=True, net=None, lineno=toks.lineno(1))
    def p_abcd_action_if (self, toks) :
        "abcd_action␣:␣LSQB␣abcd_access_list␣IF␣testlist␣RSQB"
        toks[0] = Tree("action", toks[2], test=toks[4], net=None, lineno=toks.lineno(1))
```

Tokens `LSQB`, `RSQN` and `IF` correspond respectively to left and right square brackets that enclose an action, and to keyword **if**. Symbol abcd_access_list is a non-terminal of ABCD grammar that is handled by another rule. Finally, symbol testlist is a non-terminal of Python grammar that corresponds to a Boolean condition. Both these rules build an AST for the recognised action. The full parser has 31 rules, including one to handle parsing errors (plus all the rules inherited from Python grammar).

The last component of the ABCD compiler is in charge to transform the AST from the parser into a Petri net, which is quite a simple task because the operators in the language directly correspond to those implemented at Petri net level. Additionally, this translation attaches many information to the produced Petri net in order to expose its hierarchical structure and relate it to the ABCD code from which it was built.

This translation component requires 165 lines of code. All together, including command line handling and such auxiliary parts, the ABCD compiler is 450 lines long. So, we feel that it is indeed quick and simple to build such a tool with SNAKES in hands.

6 Using SNAKES as a service

SNAKES can be used as a service, which allows any program to use its features without the need to be written in Python. To do so, SNAKES includes a simple server that waits queries formatted as XML messages and answers with XML messages too. This system can be seen as a simplified XML RPC mechanism. The communication is made over UDP, which reduces to the minimum the complexity of writing a client: each query or answer is transported in a single UDP datagram. This makes this system suitable for communication local to a computer between one client and one server, but it is not intended to work over the Internet or with multiple clients: indeed, no reliability layer is added over UDP and there is no support for session nor authentication. Format of queries and answers is as simple as possible, and it is PNML as much as possible when Petri net related information is exchanged.

The query language is very simple but yet powerful, and it is easy to extend it by adding methods to class Query that is defined in plugin query. Only four kind of queries are currently recognised: "set", "get", "del" and "call".

The "set" query assigns a value to a name at server side; the value may be any Python object formatted in PNML (including extensions recognised by SNAKES), for instance, it may be a Petri net.

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
```

```
  <query name="set">
   <argument>
    <object type="str">name</object>
   </argument>
   <argument>
     ... any PNML formatted value ...
   </argument>
  </query>
 </pnml>
```

The “get” query returns the value previously assigned to a name.

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
 <query name="get">
  <argument>
   <object type="str">name</object>
  </argument>
 </query>
</pnml>
```

The “del” query removes a name from server’s memory.

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
 <query name="del">
  <argument>
   <object type="str">name</object>
  </argument>
 </query>
</pnml>
```

Finally, the “call” query performs a function or method call at server side and returns to the client what the call returns.

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
 <query name="call">
  <argument>
   <object type="str">method_or_function_name</object>
  </argument>
  <argument>
     ... any PNML formatted value as first argument ...
  </argument>
   ... as many arguments as needed ...
 </query>
</pnml>
```

Even if very simple, this scheme is powerful enough to allow great flexibility because queries can be arbitrarily nested. So it is easy for instance to assign the result of a computation by nesting a “call” inside a “set”. Similarly, it is possible to call a method of an object returned by another call. It is also possible to execute arbitrary Python statements using exec or eval built-in routines.

By using an appropriate combination of those basic queries, one may build a set of template complex queries with placeholders where changing data has to be used. For instance, it is easy to build a query that lists the enabled transitions of a net, the only part that need to change is the name of the net. So, running this query from the client point of view is just a question of inserting a name at the right position into an XML formatted text, sending the resulting string over UDP and parsing the XML that it gets back in order to extract the appropriate information.

Like queries, answers have a simple formatting. There is basically two answers, depending on whether the query could be successfully executed or not. In case of an error, the answer is:

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
 <answer status="error" error="ExceptionName">Exception message</answer>
</pnml>
```

When there is no error, status is set to "ok", and the answer may include data or not, depending on whether the execution of the query returned data or not. This is for instance an answer without data:

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
 <answer status="ok"/>
</pnml>
```

If some return value has to be passed to the client, it is encoded in PNML and nested in tag <answer>:

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
 <answer status="ok">
  ... PNML data ...
 </answer>
</pnml>
```

7 Future and ongoing works

SNAKES is still under active development and is still considered as a beta software at its current version (0.9.3). It is planned to release the first stable version (numbered 1.0) when the documentation will be complete as well as the unit tests.

The current documentation for SNAKES is composed of an API reference manual, a tutorial and a couple of text files. The former is automatically generated from comments in 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. This must be fixed in order to improve usability and increase the confidence about the absence of bugs. However, the core library is currently quite complete to this respect: 96% is documented, 87% has unit test and 95% has detailed API specification. The rest of the documentation is written separately; currently, the tutorial does not introduce all the plugins of SNAKES, but its is quite complete about the 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 [18] 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 it is likely that very few ones are worth implementing. Indeed, 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 [13] for a comprehensive survey), stochastic Petri nets [2] and object Petri nets [21]. As presented above, Merlin & Farber's time Petri nets can be implemented with 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 integrate SNAKES with another application not written in Python. For instance, a graphical editor could use SNAKES as its data model in order to provide simulation capabilities. This need is partially addressed by the server mode of SNAKES but this solution involves many XML encoding and decoding; using SNAKES through an API would be much simpler and more efficient.

In order to remove this limitation, a C binding of SNAKES is being developed. By automatically inspecting the actual Python code, wrappers for each class, method and function can be generated as Pyrex [20] 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 extensions but is also suitable for embedding Python into C programs. A module called `apix` has been introduces recently to extract and explore SNAKES' API. Exporting to Pyrex will be the next step.

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 [15]. 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 JNI framework [38].)

8 Getting involved

Anybody who wish to contribute to SNAKES is welcome. It is not necessary to produce Python code for SNAKES in order to make a useful contribution: experiment reports, bug reports, documentation, feature requests, translations, advertising, etc., are as important as source code. (Source code is obviously welcome too.)

In order to be notified of SNAKES' releases, one can register at its FreshMeat page at ⟨`http://freshmeat.net/projects/snakes`⟩.

In order to report bugs, request features, ask questions or contribute code, one may use SNAKES' LaunchPad page at ⟨`https://launchpad.net/snakes`⟩. This page also gives access to a Bazaar VCS repository where latest changes to SNAKES are made available in (almost) real-time.

9 Acknowledgements

I would like to thank Hanna Klaudel for her good advices about how to present this paper, and Emmanuel Polonowski who helped with UML notation. Let me also thank the organisers of PNTAP 2008 for having proposed this paper for publication into the Petri net newsletter.

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. D. M. Beazley. Python lex yacc. ⟨`http://www.dabeaz.com/ply`⟩.
4. B. Berthomieu and F. Vernadat. State space abstractions for time Petri nets. *Handbook of Real Time Systems*, to appear, 2006.
5. 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.
6. E. Best, R. Devillers, and M. Koutny. *Petri net algebra*. Springer-Verlag, 2001.
7. 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.
8. B. Boigelot. The Liège automata-based symbolic handler. ⟨`http://www.montefiore.ulg.ac.be/~boigelot/research/lash`⟩.
9. 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.
10. C. Bui Thanh. Generating coloured Petri nets of concurrent object-oriented programs. In *Proc. of ESMc'04*. EUROSIS, 2004.
11. C. Bui Thanh, H. Klaudel, and F. Pommereau. Box calculus with high-level buffers. In *Design, Analysis, and Simulation of Distributed Systems*. SCS, 2004.
12. A. Burt, S. Busemann, and al. Pypy. ⟨`http://codespeak.net/pypy`⟩.
13. A. Cerone and A. Maggiolo-Schettini. Time-based expressivity of time Petri nets for system specification. *Theoretical Computer Science*, 216(1-2), 1999.
14. CPN Group, University of Aarhus. CPN Tools. ⟨`http://wiki.daimi.au.dk/cpntools`⟩.
15. D. Beazley & al. Simplified wrapper and interface generator. ⟨`http://www.swig.org`⟩.
16. 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.

17. M. Dufour. Shed skin. ⟨http://code.google.com/p/shedskin⟩.
18. J. Esparza, C. Schröter, and S. Schwoon. The model-checking kit. ⟨http://www.fmi.uni-stuttgart.de/szs/tools/mckit⟩.
19. S. Evangelista. High level petri nets analysis with Helena. In *Proc. of ICATPN'05*, volume 3536 of *LNCS*. Springer-Verlag, 2005.
20. G. Ewing. Pyrex, a language for writing Python extension modules. ⟨http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex⟩.
21. B. Farwer and K. Misra. Modelling with hierarchical object Petri nets. *Fundamenta Informaticae*, 55(2), 2003.
22. 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.
23. A. Hagberg, D. Schult, and M. Renieris. Pygraphviz. ⟨https://networkx.lanl.gov/wiki/pygraphviz⟩.
24. F. Heitmann. Petri nets tool database at the Petri net world. ⟨http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools⟩.
25. H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau. Towards efficient verification of systems with dynamic process creation. volume 5160 of *LNCS*. Springer, 2008.
26. H. Klaudel and F. Pommereau. M-nets, a survey. *Acta Informatica*, 7(236), 2008.
27. K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1), 1995.
28. P. M. Merlin and D. J. Farber. Recoverability of communication protocol. *IEEE Trans. on Communications*, 24(9), 1976.
29. R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.
30. Parallel Systems Group, University of Oldenburg. The PEP tool. ⟨http://peptool.sourceforge.net⟩.
31. 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.
32. F. Pommereau. Causal Time Calculus. In *Proc. of FORMATS'03*, volume 2791 of *LNCS*. Springer-Verlag, 2003.
33. F. Pommereau. Versatile boxes: a multi-purpose algebra of high-level Petri nets. In *Prof of DASDS'04*. SCS/ACM, 2004.
34. 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.
35. Python Software Foundation. Python programming language. ⟨http://www.python.org⟩.
36. Research Group Petri Net Technology, Humboldt Universität of Berlin. The Petri net kernel. ⟨http://www.informatik.hu-berlin.de/top/pnkn⟩.
37. A. Rigo. Psyco. ⟨http://psyco.sourceforge.net⟩.
38. Sun Microsystems, Inc. Java native interface. ⟨http://java.sun.com/j2se/1.5.0/docs/guide/jni⟩.