Nets in Nets with SNAKES

Franck Pommereau

Nets in nets with SNAKES

Franck Pommereau

LACL. University Paris East. 61 avenue du général de Gaulle. 94010 Créteil. France. pommereau@univ-paris12.fr

Summary. This paper presents the toolkit SNAKES, focusing on the ability to model Petri nets whose tokens are Petri nets (so called *nets in nets*). SNAKES is a general Petri net library that allows to model and execute Python-coloured Petri nets: tokens are Python objects and net inscriptions are Python expressions. Since SNAKES itself is programmed in Python, Petri net inscriptions can handle Petri net objects as data values, for instance as tokens.

1 Introduction

SNAKES is a general Petri net library designed with quick prototyping in mind. For this aim, SNAKES offers a flexible architecture based on a *core library*, that defines a basic Petri net structure, complemented with a variety of *extension modules* (*i.e.*, *plugins*), that introduce additional features. The core library provides a general model of Python-coloured Petri nets: tokens are Python objects, transitions guards are Python expressions and arcs can carry Python expressions. See for instance [3] about the well known ML-coloured Petri nets.

In this paper, we focus on the fact that, because SNAKES itself is programmed in Python, it is possible to exploit the features of SNAKES within a Petri net inscriptions or tokens. In particular, we will show how to define a Petri net whose tokens are Petri nets [9]. It is also possible to define interactions between the nets at various levels. We will show later on how to transform token nets at firing time, how to synchronise the firing of transitions in token nets with the firing of the transition that uses these token nets, and how to program transitions that modify the structure of the net that they belong to when fired. Our aim is to show the feasibility of these features, a real implementation would require to design and program dedicated plugins in order to provide more general and reusable features, which is out of the scope of the present paper.

In order to demonstrate this possibilities, the rest of the paper will focus on concrete examples. The features and architecture of SNAKES will be introduced when needed in the examples. A general presentation of SNAKES can be found in [6, 7] and a tutorial is available at SNAKES homepage [4]. This paper assumes no prior knowledge of Python and key features will be explained as needed; however, Python programs are very readable and programmers usually feel comfortable with simple Python code (like in this paper). A good Python tutorial is available at Python homepage [8]. The complete source code of the examples is provided as an appendix at the end of the paper.

2 SNAKES overview

To start with, we define a very simple Petri net, which allow to illustrate how to:

- load SNAKES with plugins;
- build a Petri net using SNAKES;
- draw a Petri net using a plugin;
- execute transitions;
- access the marking of a net;
- compute the whole state space of a Petri net.

2.1 Loading SNAKES

First, we load SNAKES. If no plugin is needed, its enough to import module snakes.nets. However, we would like to be able to draw Petri nets so we load a plugin called gv that uses GraphViz [1] in order to layout nets and print pictures of them.

Listing 1. Basic example.

```
1 import snakes.plugins
2 snakes.plugins.load("gv", "snakes.nets", "nets")
3 from nets import *
```

Line 1 loads module snakes.plugins in order to allow to call its function load in line 2. (Notice that Python statements end at the end of the line, without a semi-colon.) This function load expects three arguments:

1. The name of the plugin to load, or a list of plugin names. Here, we load a single plugin called gv.
2. The name of the module being extended. This is almost always snakes.nets since it provides all the core library.
3. The name of the module created by extending snakes.nets with plugin gv. Here we call nets this freshly created module.

Thanks to this function call, a module called nets is now loaded and line 3 allows to access directly to its content from our program.

2.2 Building a Petri net

The next step is to create a Petri net as an instance of class PetriNet (line 4), and add nodes (lines 5–7) and arcs to it (lines 8–9):

Listing 2. Basic example (continued).

```
4 n = PetriNet("mynet")
5 n.add_place(Place("p1", [dot]))
6 n.add_place(Place("p2"))
7 n.add_transition(Transition("t"))
8 n.add_input("p1", "t", Value(dot))
9 n.add_output("p2", "t", Value(dot))
```

Line 4, the created net is given the name `"mynet"`. Then, each place is created as an instance of class Place whose constructor expects a name and an optional collection of tokens. So, the first place is called `"p1"` and initially marked by a single black token (given as a list with only one dot item); the second place is called `"p2"` and is not initially marked. Each place is added to the net using the dedicated method add_place. Line 7, a transition called `"t"` is added to the net. Arcs are divided into *input* arcs, *i.e.*, from a place toward a transition, and *output* arcs, *i.e.*, from a transition toward a place. Line 8, an input arc is added from `"p1"` to `"t"` and labelled by Value(dot); this means that the arc will consume (because it is an input arc) a single token whose value is the black token dot. Similarly, an output arc is added in order to produce a black token into `"p2"` when `"t"` fires.

Thanks to plugin gv, a PetriNet instance is equipped with a method draw that allows to layout and draw the net. So, a picture of our net can be produced with a single statement:

Listing 3. Basic example (continued).

```
10 n.draw("mynet.ps")
```

The result is shown on the left of figure 1. Here, PostScript output has been asked by providing file extension "`.ps`", but many other formats are supported by GraphViz (in particular, PNG and JPEG).

2.3 Firing transitions

The firing of a transition is decomposed in two steps:

1. Possible *modes* are computed: a mode is binding of the variables in the transition guard and the annotations of its adjacent arcs that enables the transition.
2. A mode is chosen in order to actually fire the transition, *i.e.*, consume and produce tokens according to input and output arcs.

This is reflected by the following code:

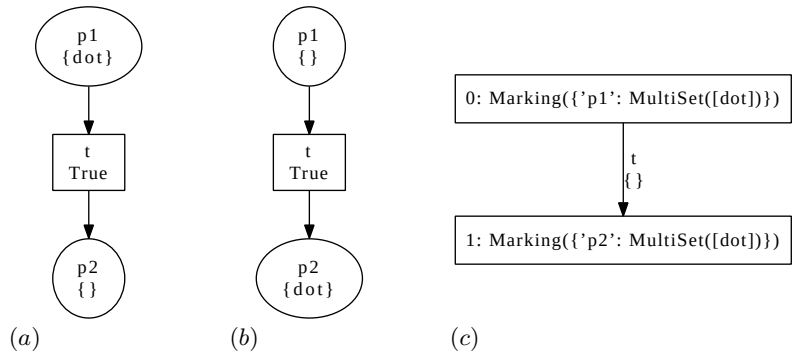

Fig. 1. (*a*) our first Petri net as drawn by GraphViz; (*b*) the same net after its transition has fired; (*c*) the corresponding marking graph. Each place is labelled by its name and marking (in set-like notation); each transition is labelled by its name and guard (True by default); each state in the marking graph is labelled by its number and marking; each edge in the marking graph is labelled by a transition name and a mode that correspond to the firing.

Listing 4. Basic example (continued).

```
11 t = n.transition("t")
12 m = t.modes()
13 t.fire(m.pop())
```

Line 11, the transition object for `"t"` is fetched from the net and stored in variable `t` for an easier access to it. Then, line 12, its modes are computed and stored into variable `m`. Since the transition involves no variable, only one mode is possible to fire it and this is the empty binding; so, `m` holds a single value. Last, line 13, the transition is fired with a mode picked from `m` (`m.pop()` removes and returns a value from `m`, that is here the unique value). If the net is drawn again (this is done in line 14 that is not displayed here), we get the picture displayed in the middle of figure 1.

2.4 Computing the state space

In order to reset the marking, we can use the following:

Listing 5. Basic example (continued).

```
15 n.set_marking(Marking(p1=MultiSet([dot]), p2=MultiSet([])))
```

Method `set_marking` expects a `Marking` instance as its argument, which is constructed by providing for each place the multiset of its tokens.

Then, instead of manually computing modes and firing transitions, we can use class `StateGraph` in order to compute all the reachable states from the

provided initial marking. The resulting graph can be draw thanks to plugin gv. For instance:

Listing 6. Basic example (continued).

```
16 g = StateGraph(n)
17 for i in g :
18     g.net.draw("mynet-%s.ps" % i)
19 g.draw("mynet-states.ps")
```

Line 16 creates the StateGraph instance; the constructor expects the marked net whose state space has to be computed. Lines 17–18, a loop iterates over the states of g, numbered from zero. The body of the loop just contains line 18 (Python defines block through indentation) that draws the net in each reached state: i is the state number and operator % allows to substitute the value of i inside the template string "mynet-%s.ps" (similarly to printf in C, with here an automatic coercion of integer i to a string). Notice that g.net.draw is called, and not n.draw as one could expect; indeed, g holds a copy of n so that n is not affected when the marking of the copy is modified as the state space is computed. The resulting pictures are exactly the same as in figure 1. Then, line 19, the state space itself is drawn, which results in the picture shown on the right of figure 1. Notice that the state space is constructed on-the-fly, *i.e.*, the successors of each state are computed only when the state is iterated over. This is useful, for instance, to display a progression of state space computation or to check a property on-the-fly.

3 Petri nets as tokens

In this section, we use instances of class PetriNet as tokens. We consider two levels: the token level where Petri nets use dot as their tokens, and the container level that uses the nets of the token level as tokens. In order to produce the token nets, we define a function token_net:

Listing 7. Nets as tokens.

```
1  import snakes.plugins
2  snakes.plugins.load("gv", "snakes.nets", "nets")
3  from nets import *
4
5  def token_net (name) :
6      a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
7      net = PetriNet(name)
8      net.add_place(Place(a, [dot]))
9      net.add_place(Place(b))
10     net.add_transition(Transition(t))
11     net.add_input(a, t, Value(dot))
```

```
12      net.add_output(b, t, Value(dot))
13      return net
```

This function (whose body, as usual, is delimited by indentation) expects as its sole argument a string that is used to define nodes names and that becomes the name of the returned Petri net. For instance, if `name="foo"` then line 6 results in `a="afoo"`, `b="bfoo"` and `t="tfoo"`.

Then, a contained Petri net is defined with a similar structure:

Listing 8. Nets as tokens (continued).

```
15  n = PetriNet("container")
16  n.add_place(Place("p1", [token_net("Toknet")]))
17  n.add_place(Place("p2"))
18  n.add_transition(Transition("t"))
19  n.add_input("p1", "t", Variable("x"))
20  n.add_output("p2", "t", Expression("x.copy()"))
```

Line 16, a new token net is created for the marking of place `"p1"`. Line 19, the input arc is labelled by a variable called `"x"`; this allows to consume one token from the input place while binding its name to a variable `x`. Line 20, the output arc is labelled by an expression; this allows to compute a new token to be produced in the output place. Here, we simply copy the token net bound to `x` by calling its method `copy`.

Copying net tokens may be important in order to avoid unwanted side effects. For instance: suppose that we add an output arc from the transition toward `"p1"` in order to reproduce the consumed token net, and suppose that we do not copy token nets. Then, after firing the container's transition, the two places of the container net would hold each a reference to the same token net. Firing the transition of one of these token nets would thus modify the marking in both places of the container net. This may be desirable or not, and this can be controlled by choosing to produce either copies of or references to token nets.

The rest of the program builds the marking graph, drawing the container net in each state as well as all the token nets in each place:

Listing 9. Nets as tokens (continued).

```
22  g = StateGraph(n)
23  for i in g :
24      g.net.draw("%s-state%s.ps" % (g.net.name, i))
25      for place in g.net.place() :
26          for j, tok in enumerate(place.tokens) :
27              tok.draw("%s-state%s-place%s-token%s.ps"
28                       % (g.net.name, i, place.name, j))
```

(Notice that line 27 is automatically continued on line 28 thanks to the parenthesising.) The resulting pictures are displayed in figure 2. Line 25, we can

see how to iterate over the places of a PetriNet object: method place returns the list of Place objects when no argument is given; otherwise, it returns the place whose name is provided (similarly, we have used method transition to fetch a transition from a net). Line 26, function enumerate is used to iterate over the tokens in a place, providing a number to each one; for instance, enumerate([x, y, z]) results in the sequence of pairs (0,x), (1,y) and (2,z).

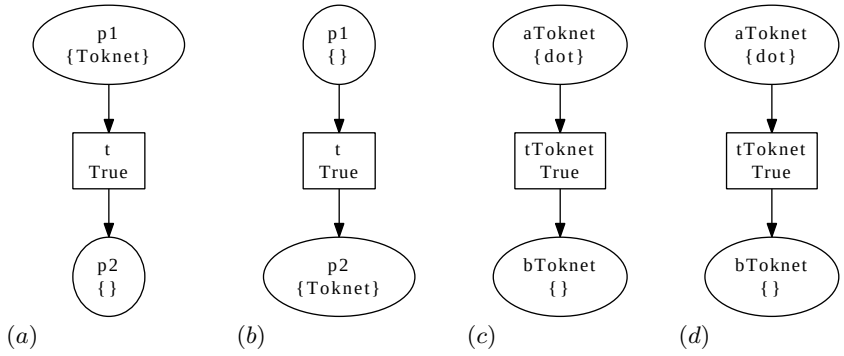

Fig. 2. (*a*) the initial state of the container net; (*b*) the state of the container net after firing; (*c*) the token net from place `"p1"` in (*a*); (*d*) the token net from place `"p2"` in (*b*), which is exactly the same as in (*c*).

4 Transforming token nets

In this section, we elaborate on the previous example. In a first step, we introduce material to rename a token net when it is copied. Then, we consider a more complex example where two token nets are composed in order to produce a new one.

4.1 Simple transformation

As a first example, we consider a program that starts exactly as the previous one (listing 7). Then, we define a function copy_rename that takes a Petri net and a string as arguments and returns a renamed copy of the net whose nodes are also renamed.

Listing 10. Renaming token nets.

```
15 def copy_rename (net, new) :
16     old = net.name
17     net = net.copy()
```

```
18      net.rename(new)
19      for node in net.node() :
20          net.rename_node(node.name, node.name.replace(old, new))
21      return net
```

Line 16, the current name of the net to process is saved in variable old. Then, the net is copied (line 17) and the copy is renamed (line 18). Notice that line 17 just results in loosing the reference to the original net: a reference to the copied net is stored in the local variable net that hides parameter net. The loop lines 19–20 iterates over all the nodes in the copy and rename them by substituting the old net name with the new one in their names.

Then, a container net (here called `"renamer"`) is defined and function copy_rename is used on the output arc.

Listing 11. Renaming token nets (continued).

```
23  n = PetriNet("renamer")
24  n.globals["copy_rename"] = copy_rename
25  n.add_place(Place("p1", [token_net("Toknet")]))
26  n.add_place(Place("p2"))
27  n.add_transition(Transition("t"))
28  n.add_input("p1", "t", Variable("x"))
29  n.add_output("p2", "t", Expression("copy_rename(x,␣'Mynet')"))
```

Line 24, function copy_rename is declared as a global name in the Petri net n. As a result, Python expressions used in this net will be able to use copy_rename, as in particular in line 29. Without line 24, the evaluation of the output arc specified line 29 would result in a exception complaining about the fact that name copy_rename is unknown. Notice line 29 the string `"copy_rename(x,␣'Mynet')"`: strings in Python can be freely delimited with single or double quotes, this allows to nest strings easily. In this example, copy_rename(x, `'Mynet'`) is a valid Python expression where `'Mynet'` is just a regular string.

The rest of the program is exactly as listing 9 in the previous section. An execution results in the pictures shown in figure 3.

4.2 Composition of token nets

In our next example, we make use of Petri net compositions defined in the *Petri Box Calculus* (PBC) and its variants [2, 5] and implemented in SNAKES. In a nutshell, places in PBC are labelled with *statuses*, allowing to distinguish *entry* places, *internal* places and *exit* places. All together, these places form the *control flow* of a Petri net and PBC defines binary operators to compose nets with respect to their control flow. If N_1 and N_2 are two PBC nets:

- the *sequential composition* $N_1 \,\&\, N_2$ allows to execute N_1 once, followed by one execution of N_2;

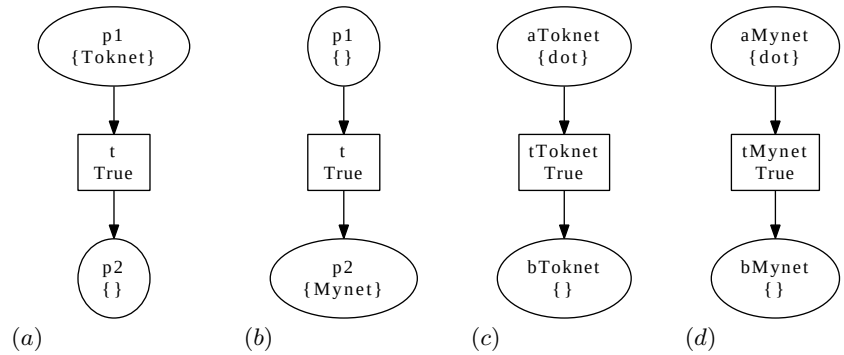

Fig. 3. (*a*) the initial state of the renamer net; (*b*) the state of the renamer net after firing; (*c*) the token net from place `"p1"` in (*a*); (*d*) the token net from place `"p2"` in (*b*). With respect to figure 2, notice the renaming operated at (*b*) and (*d*).

- the *choice* $N_1 + N_2$ allows to execute once either N_1 or N_2;
- the *iteration* $N_1 * N_2$ allows to execute repeatedly N_1, followed by one execution N_2;
- the *parallel composition* $N_1 \mid N_2$ allows to execute both N_1 and N_2 concurrently.

Notice that the operators presented above are denoted as implemented in SNAKES, instead of as originally defined in PBC (SNAKES uses Python operators).

We now consider token nets similar to the previous ones, with input places (the `a`$\cdots$ ones) being labelled as entry places, and output places (the `b`$\cdots$ ones) labelled as exit places. The container net consists in one transition with two input places holding one token net each, and one output place to receive the parallel composition of the two consumed token nets upon firing.

The program starts with the loading a SNAKES extended with plugins `gv` (to draw nets) and `ops` (to support PBC operations). Then, function `token_net` is defined just as before except that it assigns statuses to the places (lines 8–9):

Listing 12. Composing token nets.

```
1  import snakes.plugins
2  snakes.plugins.load(["gv", "ops"], "snakes.nets", "nets")
3  from nets import *
4
5  def token_net (name) :
6      a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
7      net = PetriNet(name)
8      net.add_place(Place(a, [dot], status=entry))
9      net.add_place(Place(b, status=exit))
```

```
10      net.add_transition(Transition(t))
11      net.add_input(a, t, Value(dot))
12      net.add_output(b, t, Value(dot))
13      return net
```

Then, the container net, called `"composer"`, is created as explained above:

Listing 13. Composing token nets (continued).

```
15  n = PetriNet("composer")
16  n.add_place(Place("left", [token_net("foo")]))
17  n.add_place(Place("right", [token_net("bar")]))
18  n.add_place(Place("parall"))
19  n.add_transition(Transition("t"))
20  n.add_input("left", "t", Variable("x"))
21  n.add_input("right", "t", Variable("y"))
22  n.add_output("parall", "t", Expression("x|y"))
```

Lines 20–21, notice the arcs labelled by variables allowing to bind the two token nets to different names `x` and `y`. Line 22, the parallel composition of these two token nets is computed by simply providing expression "`x|y`" on the output arc.

The rest of the program, as in listing 9, builds the state space and draws all the container and token nets at each state. This results in the pictures shown in figure 4. It may be noted that node names in a compound net are systematically obtained from the node names of its components; for instance, "$[x|]$" denotes a new node obtained from a node called x in the left operand of the parallel composition (read this name as "x" on the left of "$|$", with "$[\cdots]$" around to handle nested compositions). Similarly, the name of the compound net "foo|bar" is derived from that of its components.

5 Synchronising firing

Let us consider again the example from section 3. In this section, we elaborate on it in order to enforce synchronised firing between transitions of the two levels.

The beginning of this program is exactly as in listings 7. Then, after the definition of function `token_net`, we also define a function to check whether transitions from a token net are ready to fire:

Listing 14. Synchronised firing.

```
15  def ready (net, *names) :
16      for n in names :
17          if not net.transition(n).modes() :
18              return False
19      return True
```

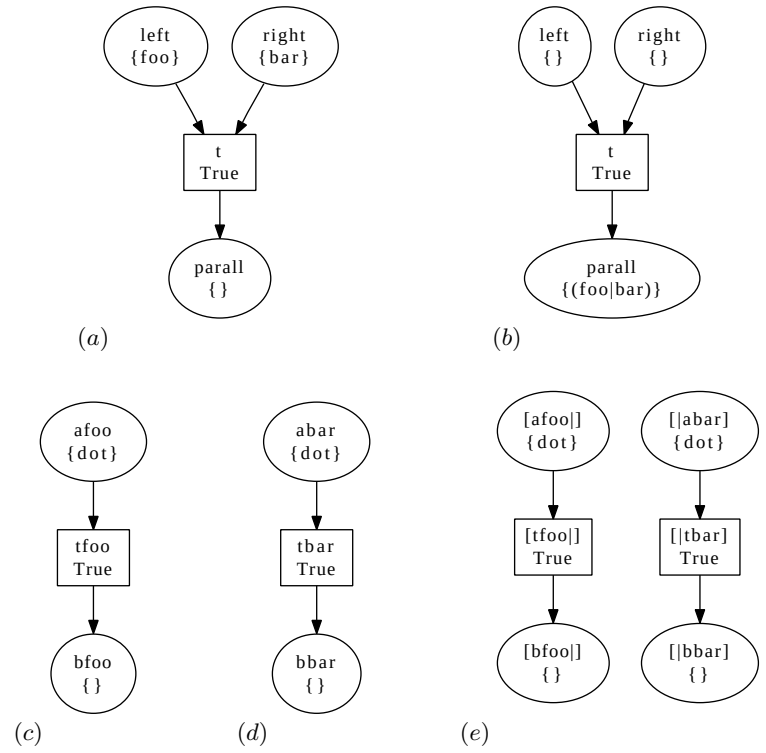

Fig. 4. (*a*) the initial state of the composer net; (*b*) the composer net after firing; (*c*) the token net from place `"left"` in (*a*); (*d*) the token net from place `"right"` in (*a*); (*e*) the token net from place `"parall"` in (*b*), which is the parallel composition of the token nets in (*c*) and (*d*).

Line 15, parameter ∗names denotes that several values may be passed to the function and are grouped as a sequence called names. For instance, if one calls ready(x,"a","b","c") then we have names=("a","b","c"). The function consists in iterating over all the given names (line 16) and for each, test if it has no mode (line 17) and if so, returns False. If none of the provided transition is inactive, then function ready returns True (line 19). Notice that in line 17, we use a sequence as a truth value; this is allowed in Python where a sequence is true iff its non-empty.

A second helper function is needed to produce a copy of a net in which specified transitions have been fired. The arguments are the same as for function ready:

Listing 15. Synchronised firing (continued).

```
21 def synchro (net, *names) :
```

```
22      net = net.copy()
23      for n in names :
24          t = net.transition(n)
25          m = t.modes()
26          t.fire(m.pop())
27      return net
```

Line 22, the net is copied, then for each given transition name (line 23), the corresponding transition object is fetched from the net (line 24), its modes are computed (line 25) and the transition is fired with one of these modes (line 26). In a real model, an appropriate mode should be chosen more carefully, for instance by considering one mode that is compatible with the container's mode. Finally, the net, modified by the firings, is returned (line 27). Notice that it is not check whether firing the given transitions is possible (if not, an exception will occur) and we assume that function ready as been used appropriately for this purpose. Indeed, this is checker in the guard of the transition in the container net:

Listing 16. Synchronised firing (continued).

```
29  n = PetriNet("synchroniser")
30  n.globals["ready"] = ready
31  n.globals["synchro"] = synchro
32  n.add_place(Place("p1", [token_net("Toknet")]))
33  n.add_place(Place("p2"))
34  n.add_transition(Transition("t", Expression("ready(x,␣'tToknet')")))
35  n.add_input("p1", "t", Variable("x"))
36  n.add_output("p2", "t", Expression("synchro(x,␣'tToknet')"))
```

Lines 30–31, the defined functions are declared in the execution environment of the container net. Function ready is used line 34 in the guard of the transition (provided by an Expression object as the second argument of the constructor of class Transition). Line 36, function synchro is used on the output arc in order to produce a copy of the net taken from the input place (as x) in which the transition `"tToknet"` has fired. We known that this firing is possible since transition `"t"` is guarded by ready(x,'tToknet') that exactly checks this.

The rest of the program is taken from listing 9 in order to produce the pictures shown in figure 5.

6 Self-modifying nets

As a last example, we consider a Petri net whose structure is modified by the firing of its transition. We start with a net that has the same structure as that of figure 1(a). When its transition `"t"` fires, it creates a new transition (called `"back"`) and arcs in order to reset the marking. When this transition `"back"` fires, it removes itself from the net.

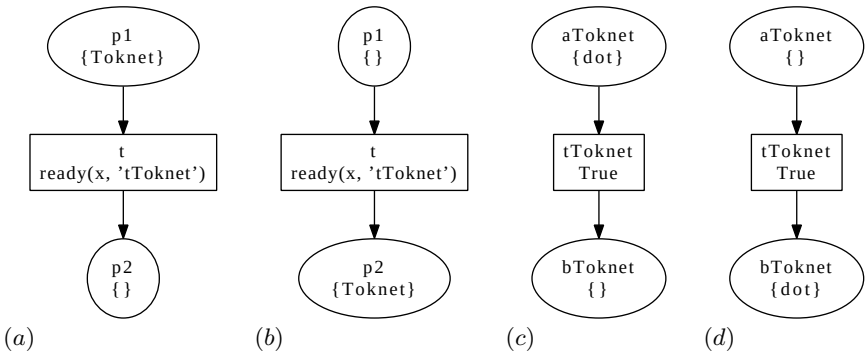

Fig. 5. (a) the initial state of the synchroniser net; (b) the state of the synchroniser net after firing; (c) the token net from place `"p1"` in (a); (d) the token net from place `"p2"` in (b), which is exactly that from (c) after the firing of its transition. With respect to figure 2, notice in (d) that the transition as fired.

In order to achieve such an effect, we extend twice class Transition in order to redefine its method fire: a first class BackwardTransition is used to implement the transition added to the net; a second class ForwardTransition is used to implement the adding of a BackwardTransition.

Listing 17. Self-modifying net.

```
1  import snakes.plugins
2  snakes.plugins.load("gv", "snakes.nets", "nets")
3  from nets import *
4
5  class BackwardTransition (Transition) :
6      def fire (self, binding) :
7          Transition.fire(self, binding)
8          self.net.remove_transition(self.name)
9
10 class ForwardTransition (Transition) :
11     def fire (self, binding) :
12         Transition.fire(self, binding)
13         self.net.add_transition(BackwardTransition("back"))
14         self.net.add_input("p2", "back", Value(dot))
15         self.net.add_output("p1", "back", Value(dot))
```

Line 5, the new class BackwardTransition is defined as a sub-class of Transition; it redefined method fire on line 6. This method expects two arguments: self is the instance whose method has been called (this is an explicit argument in Python) and binding is the mode under which the transition is fired. Line 7, method fire from the super-class Transition is called in order to achieve the

actual firing. Line 8, the transition removes itself from the Petri net that is accessible under the attribute net. Notice that removing a transition also remove all its arcs. Lines 10–15, class ForwardTransition is defined similarly, the only difference is that it adds a BackwardTransition to the net instead of removing it.

The rest of the code is dedicated to build the net and to draw the two states of the net, resulting in the pictures shown in figure 6.

Listing 18. Self-modifying net (continued).

```
17 n = PetriNet("modifier")
18 n.add_place(Place("p1", [dot]))
19 n.add_place(Place("p2"))
20 n.add_transition(ForwardTransition("t"))
21 n.add_input("p1", "t", Value(dot))
22 n.add_output("p2", "t", Value(dot))
23
24 n.draw("modifier-0.ps")
25 n.transition("t").fire(n.transition("t").modes().pop())
26 n.draw("modifier-1.ps")
```

Notice line 20 that we use ForwardTransition instead of Transition.

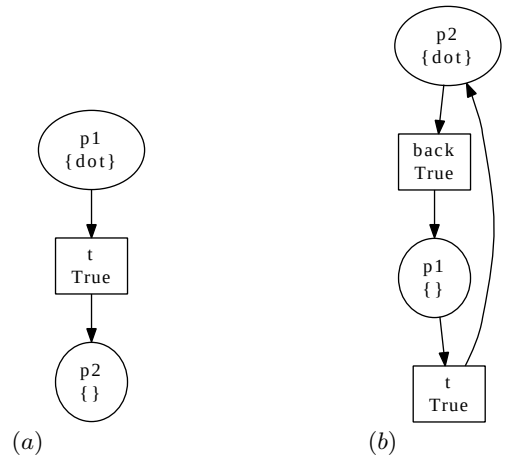

Fig. 6. (*a*) the initial state of the modifier net; (*b*) the state of the modifier net after the firing of the forward transition. Firing the backward transition from (*b*) would result in net (*a*) again.

7 Conclusion

In this paper, we have presented the toolkit SNAKES, focusing on its ability to represent and execute Petri whose tokens are Petri nets. In particular, we have shown that it is easy to perform transformations on the token nets at firing time, to synchronise the firing of the transitions at different levels. We have shown also how self-modifying nets can be defined.

The code presented throughout the paper should be considered as a proof of concept rather than as a guideline about how to implement these features in a real tool. For such a use case, it would be better to carefully design and program a plugin for SNAKES in order to provide a solution that would be both more general and easier to use. The design phase in particular seems crucial for both generality and robustness, it's likely that it requires a good background about nets in nets. Any contribution to SNAKES or collaboration proposal toward adding such a plugin will be welcomed.

References

1. AT&T Research. Graphviz, graph visualization software. ⟨`http://www.graphviz.org`⟩.
2. 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.
3. K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.
4. F. Pommereau. SNAKES is the Net Algebra Kit for Editors and Simulators. ⟨`http://lacl.univ-paris12.fr/pommereau/soft/snakes`⟩.
5. F. Pommereau. Versatile boxes: a multi-purpose algebra of high-level Petri nets. In *Proc. of DASDS'04*. SCS/ACM, 2004.
6. F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, September 2008.
7. F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. In *Proc. of PNTAP'08*, ACM Digital Library. ACM, 2008.
8. Python Software Foundation. Python programming language. ⟨`http://www.python.org`⟩.
9. R. Valk. Petri nets as token objects: An introduction to elementary object nets. In *ICATPN '98*. Springer-Verlag, 1998.

A Source code of the examples

Listing 19. Basic example.

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *
n = PetriNet("mynet")
n.add_place(Place("p1", [dot]))
n.add_place(Place("p2"))
n.add_transition(Transition("t"))
n.add_input("p1", "t", Value(dot))
n.add_output("p2", "t", Value(dot))
n.draw("mynet.ps")
t = n.transition("t")
m = t.modes()
t.fire(m.pop())
n.draw("mynet-bis.ps")
n.set_marking(Marking(p1=MultiSet([dot]), p2=MultiSet([])))
g = StateGraph(n)
for i in g:
    g.net.draw("mynet-%s.ps" % i)
g.draw("mynet-states.ps")
```

Listing 20. Nets as tokens.

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *

def token_net(name):
    a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
    net = PetriNet(name)
    net.add_place(Place(a, [dot]))
    net.add_place(Place(b))
    net.add_transition(Transition(t))
    net.add_input(a, t, Value(dot))
    net.add_output(b, t, Value(dot))
    return net

n = PetriNet("container")
n.add_place(Place("p1", [token_net("Toknet")]))
n.add_place(Place("p2"))
n.add_transition(Transition("t"))
n.add_input("p1", "t", Variable("x"))
```

```
n.add_output("p2", "t", Expression("x.copy()"))

g = StateGraph(n)
for i in g :
    g.net.draw("%s-state%s.ps" % (g.net.name, i))
    for place in g.net.place() :
        for j, tok in enumerate(place.tokens) :
            tok.draw("%s-state%s-place%s-token%s.ps"
                     % (g.net.name, i, place.name, j))
```

Listing 21. Renaming token nets.

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *

def token_net (name) :
    a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
    net = PetriNet(name)
    net.add_place(Place(a, [dot]))
    net.add_place(Place(b))
    net.add_transition(Transition(t))
    net.add_input(a, t, Value(dot))
    net.add_output(b, t, Value(dot))
    return net

def copy_rename (net, new) :
    old = net.name
    net = net.copy()
    net.rename(new)
    for node in net.node() :
        net.rename_node(node.name, node.name.replace(old, new))
    return net

n = PetriNet("renamer")
n.globals["copy_rename"] = copy_rename
n.add_place(Place("p1", [token_net("Toknet")]))
n.add_place(Place("p2"))
n.add_transition(Transition("t"))
n.add_input("p1", "t", Variable("x"))
n.add_output("p2", "t", Expression("copy_rename(x,␣'Mynet')"))

g = StateGraph(n)
for i in g :
    g.net.draw("%s-state%s.ps" % (g.net.name, i))
```

```
    for place in g.net.place() :
        for j, tok in enumerate(place.tokens) :
            tok.draw("%s-state%s-place%s-token%s.ps"
                     % (g.net.name, i, place.name, j))
```

Listing 22. Composing token nets.

```
import snakes.plugins
snakes.plugins.load(["gv", "ops"], "snakes.nets", "nets")
from nets import *

def token_net (name) :
    a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
    net = PetriNet(name)
    net.add_place(Place(a, [dot], status=entry))
    net.add_place(Place(b, status=exit))
    net.add_transition(Transition(t))
    net.add_input(a, t, Value(dot))
    net.add_output(b, t, Value(dot))
    return net

n = PetriNet("composer")
n.add_place(Place("left", [token_net("foo")]))
n.add_place(Place("right", [token_net("bar")]))
n.add_place(Place("parall"))
n.add_transition(Transition("t"))
n.add_input("left", "t", Variable("x"))
n.add_input("right", "t", Variable("y"))
n.add_output("parall", "t", Expression("x|y"))

g = StateGraph(n)
for i in g :
    g.net.draw("%s-state%s.ps" % (g.net.name, i))
    for place in g.net.place() :
        for j, tok in enumerate(place.tokens) :
            tok.draw("%s-state%s-place%s-token%s.ps"
                     % (g.net.name, i, place.name, j))
```

Listing 23. Synchronised firing.

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *

def token_net (name) :
    a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
```

```
    net = PetriNet(name)
    net.add_place(Place(a, [dot]))
    net.add_place(Place(b))
    net.add_transition(Transition(t))
    net.add_input(a, t, Value(dot))
    net.add_output(b, t, Value(dot))
    return net

def ready (net, *names) :
    for n in names :
        if not net.transition(n).modes() :
            return False
    return True

def synchro (net, *names) :
    net = net.copy()
    for n in names :
        t = net.transition(n)
        m = t.modes()
        t.fire(m.pop())
    return net

n = PetriNet("synchroniser")
n.globals["ready"] = ready
n.globals["synchro"] = synchro
n.add_place(Place("p1", [token_net("Toknet")]))
n.add_place(Place("p2"))
n.add_transition(Transition("t", Expression("ready(x, 'tToknet')")))
n.add_input("p1", "t", Variable("x"))
n.add_output("p2", "t", Expression("synchro(x, 'tToknet')"))

g = StateGraph(n)
for i in g :
    g.net.draw("%s-state%s.png" % (g.net.name, i))
    for place in g.net.place() :
        for j, tok in enumerate(place.tokens) :
            tok.draw("%s-state%s-place%s-token%s.png"
                     % (g.net.name, i, place.name, j))
```

Listing 24. Self-modifying net.

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *

```

```
class BackwardTransition (Transition) :
    def fire (self, binding) :
        Transition.fire(self, binding)
        self.net.remove_transition(self.name)

class ForwardTransition (Transition) :
    def fire (self, binding) :
        Transition.fire(self, binding)
        self.net.add_transition(BackwardTransition("back"))
        self.net.add_input("p2", "back", Value(dot))
        self.net.add_output("p1", "back", Value(dot))

n = PetriNet("modifier")
n.add_place(Place("p1", [dot]))
n.add_place(Place("p2"))
n.add_transition(ForwardTransition("t"))
n.add_input("p1", "t", Value(dot))
n.add_output("p2", "t", Value(dot))

n.draw("modifier-0.ps")
n.transition("t").fire(n.transition("t").modes().pop())
n.draw("modifier-1.ps")
```