SNAKES: A Flexible High-Level Petri Nets Library
(Tool Paper)

Franck Pommereau(✉)

IBISC, University of Évry/Paris-Saclay, 23 bd de France,
91037 Évry Cedex, France
franck.pommereau@ibisc.univ-evry.fr

Abstract. SNAKES (SNAKES is the Net Algebra Kit for Editors and Simulators) is a general purpose Petri nets library, primarily for the Python programming language but portable to other ones. It defines a very general variant of Python-coloured Petri nets that can be created and manipulated through the library, as well as executed to explore state spaces. Thanks to a variety of plugins, SNAKES can handle extensions of Petri nets, in particular algebras of Petri nets [4,26]. SNAKES ships with a compiler for the ABCD language that is precisely such an algebra. Finally, one can use the companion tool Neco [14] that compiles a Petri net into an optimised library allowing to compute efficiently its state space or perform LTL model-checking thanks to library SPOT [8,13]. This paper describes SNAKES' structure and features.

Keywords: Petri nets library · Prototyping · Simulation · Model-checking

1 SNAKES in a Nutshell

SNAKES is a general purpose Petri net library for the Python programming language (but we show in Section 4 that it can be ported to other languages). Using SNAKES, one can create Petri nets, transform them (add/remove/... nodes, add/remove/... arcs, etc.), manipulate their markings, and also fire transitions (sequentially). SNAKES is not designed to perform analysis but because it can execute modelled nets, it may be used to explore traces or state spaces. However, a companion tool called Neco is preferred for this purpose and provides fast reachability and LTL explicit analysis.

SNAKES uses a very general variant of Python-coloured Petri nets (see Section 1.3): tokens can carry arbitrary Python objects, transitions guards are arbitrary Python expressions and arcs may be annotated with arbitrary Python variables or expressions. Moreover, SNAKES provides support for various Petri nets extensions: read arcs, whole-place arcs and inhibitor arcs. Because we use the same language for the library and the Petri nets annotations, users are provided

R. Devillers and A. Valmari (Eds.): PETRI NETS 2015, LNCS 9115, pp. 254–265, 2015.
DOI: 10.1007/978-3-319-19488-2_13

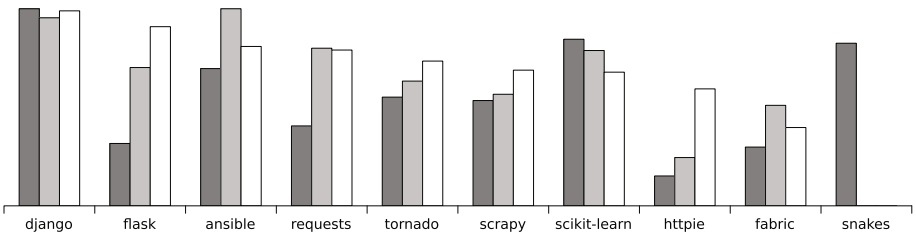

Fig. 1. SNAKES compared with the most popular Python projects on GitHub (on March 20th, 2015). From left to right, bars represent: ■ size of the project measured in number of source lines of code, ranging from 184.6k for Django to 3.7k for HTTPie; ■ number of contributors, ranging from 1.0k for Ansible to 2 for SNAKES; □ popularity measured as the number of stars (GitHub's bookmarks) times the number of forks, ranging from 13.3k stars and 5.2k forks for Django to 2 stars and 1 fork for SNAKES (however, it is worth noting that SNAKES has moved to GitHub only since March 15th, 2015).

with great flexibility. In addition to this flexibility, a general plugin mechanism is provided to allow for redefining every aspect of SNAKES, like the firing rule in particular. For instance, in [23,24] we show how SNAKES can be extended to support time Petri nets (which requires less than 100 lines of code); or in [25], we show how nets-within-nets, with transition firing synchronised between the nested levels of nets, can be implemented using less than 30 lines of code.

SNAKES has been developed since 2002, progressively growing to about 81.5k lines of portable Python, which represents quite a big effort as shown in Figure 1. One reason that increases the size of SNAKES is that it does not rely on external or system-dependant libraries and includes features that are not directly related to Petri nets, for instance: a LL(1) parser generator; tools for Python code parsing, refactoring and generation; tools for API documentation extraction and generation. On the other hand, this allows SNAKES to work out-of-the-box on any system with Python starting from version 2.5, including the 3.x series as well as alternative implementations like PyPy, Jython, IronPython, or stackless Python [27]. SNAKES is free software released under the GNU LGPL [10]. Because it is freely available, it is hard to say how many users it has, but we measured that the online documentation receives more than 300 unique visitors per month. SNAKES is available at https://github.com/fpom/snakes.

1.1 Modules and Plugins

The whole library comes as a Python package organised as a hierarchy of modules among which the main ones are:

- snakes is the top-level module that defines commonly used exceptions;
- snakes.data defines data structures like multisets, substitutions, etc.;
- snakes.typing defines a type system used to restrict the tokens in places;
- snakes.plugins gathers all the plugins provided with SNAKES (see below);
- snakes.pnml defines import/export functions to/from PNML (see below);

– snakes.nets is the main module that defines all the Petri net related structures like places, transitions, arcs, marking graph, etc.

Users typically need to import only snakes.nets that itself imports most of the other modules. At the time module snakes.pnml was written, PNML used to support only places/transitions nets and such nets are correctly imported from or exported to PNML by SNAKES. But nets with high-level features like coloured tokens are exported into a dialect that does not conform nowadays PNML, and reciprocally, high-level PNML cannot be loaded into SNAKES. Adding this support represents a huge work regarding the complexity of the latest standard.

The most useful plugins shipped with SNAKES are:

– gv allows to draw Petri nets using GraphViz [5] (see Figure 3 for pictures);
– ops provides nets compositions from algebras of Petri nets (sequence, choice, iteration and parallel composition);
– pids offers dynamic process identifiers creation and destruction [20];
– labels allows to annotate nets and their nodes with arbitrary values;
– let allows to assign variables within expressions, which is useful to avoid computing several times the same expression (more at the end of Section 1.3).

Generally, plugins are based on a set of hooks in the tools, allowing the plugin to perform a specific action when the hook is activated. SNAKES takes a more general approach: a plugin is basically a set of classes that extends the classes of a module (snakes.nets in general). This is thus much more general since anything can be extended or redefined. Moreover, it is also more flexible than standard classes inheritance because it is made dynamically, depending on which plugins are actually loaded. In order to avoid incompatible extensions and to simplify the use, plugins declare which other plugins they conflict with as well as which other they depend on.

1.2 Hello World

Figure 2 shows a simple example of SNAKES usage: this code loads snakes.nets extended with plugin gv (lines 1-3); creates a Petri net (line 4); adds three places (lines 5–7) and a transition (line 8); adds arcs (lines 9-11); draws the net once (line 12); gets the modes for the transition (line 13, the returned modes are given in the comment lines 14–17); fires the transition with one of these modes (line 18); and finally draws the net once more (line 19). The resulting pictures are displayed in Figure 3. One can note that places are here marked with string objects and that the output arc from transition “concat” to place “sentence” is labelled with a Python expression that concatenates three strings, two of which being obtained by consuming tokens in the other places.

1.3 Transition Firing

As said previously, every Petri net in SNAKES can be executed, *i.e.*, its transitions can be fired. To achieve this, we need to make a compromise between the

```
1  import snakes.plugins
2  snakes.plugins.load("gv", "snakes.nets", "snk")
3  from snk import *
4  pn = PetriNet("hello world in SNAKES")
5  pn.add_place(Place("hello", ["hello", "salut"]))
6  pn.add_place(Place("world", ["world", "le monde"]))
7  pn.add_place(Place("sentence"))
8  pn.add_transition(Transition("concat"))
9  pn.add_input("hello", "concat", Variable("h"))
10 pn.add_input("world", "concat", Variable("w"))
11 pn.add_output("sentence", "concat", Expression("h + ' ' + w"))
12 pn.draw("hello-1.eps")
13 modes = pn.transition("concat").modes()
14 # modes = [Substitution(h='salut', w='world'),
15 #    Substitution(h='salut', w='le monde'),
16 #    Substitution(h='hello', w='world'),
17 #    Substitution(h='hello', w='le monde')]
18 pn.transition("concat").fire(modes[2])
19 pn.draw("hello-2.eps")
```

Fig. 2. Python code for the "hello world" example

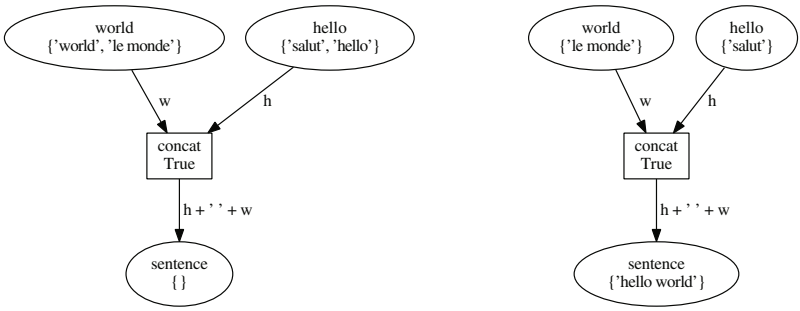

Fig. 3. Pictures generated by the "hello world" example

generality of nets definitions and some implementation restrictions. Informally, our definition is as follows: a Petri net is a tuple (S, T, ℓ, M) where,

- S is a finite set of places;
- T is a finite set of transitions, disjoint from S;
- ℓ is a labelling function such that
  - for all $s \in S$, $\ell(s)$ is the type of s, *i.e.*, a restriction on the tokens it may hold. This is implemented in snakes.typing as Boolean functions used to check whether tokens can be accepted or not,
  - for all $t \in T$, $\ell(t)$ is the guard of t, implemented as a Python expression,
  - for all $(x, y) \in (S \times T) \cup (T \times S)$, $\ell(x, y)$ is the annotation of the arc from x to y and is a multiset of expressions to specify the tokens produced or consumed through the arc;

– M is the marking, *i.e.*, a mapping from places to multisets of Python values.

In general, such a Petri net cannot be implemented, in particular in Python. For instance, imagine an arc from a place s to a transition t and labelled with a call to a function $f(x)$. To fire t, we would need, for each token value v in s to solve $v = f(x)$ in order to discover the possible bindings for variable x. This is clearly not feasible when f is an arbitrary Python function. So, SNAKES adopted the following restrictions:

– input arcs (in $S \times T$) cannot be labelled with expressions, but only with values, variables or combinations of them within structures that allow for pattern matching (currently, only tuples are implemented);
– all the variables used in a transition, its guard and surrounding arcs should appear on at least on one input arc so it can be bound.

Given this setting, the firing rule is quite straightforward and can be decomposed into two methods of a transition object `t`. First, `t.modes()` computes all the possible bindings of the transition's variables by matching input arcs annotations with respect to all the tokens available in input places. The second limitation above is not enforced but the modes of a transition that does not respect are simply not computed by SNAKES (see below about relaxing a bit this limitation); however, they could be provided by the user. Then, each such binding `m` is checked to be a mode as follows:

– for each input place `s`, check if "`eval`(ℓ(`s`,`t`), `m`)" yields a multiset of tokens actually held by `s`, where `eval` is a Python function that evaluates arbitrary Python expressions in a given environment (`m` plays this role here);
– check if "`eval`(ℓ(`t`), `m`)" returns `True`;
– for each output place `s`, check if every token in "`eval`(ℓ(`t`,`s`), `m`)" is accepted by the type of `s`.

The second method, `t.fire(m)`, actually fires the transition for a mode `m` by consuming and producing the tokens as computed above.

To overcome a bit the limitation that every variable is bound from the input arcs, plugin `let` provides a function also called `let` that allows to bind new variables during the evaluation of an expression. In practice, this is useful only during the evaluation of the guard, for instance "`x` $>$ 10 **and** `let(y="f(x)", z="g(x)")`" allows to introduce two new variables `y` and `z` whose values can be computed arbitrarily (here by calling functions `f` and `g`), and that can be used in the output arcs avoiding potential redundant calls to `f` and `g`. Note that `let` returns `True` if it can successfully bind the variables, and `False` otherwise (*e.g.*, if an expression yields an exception), which is adequate for its use in guards.

2 ABCD for Friendly Modelling

SNAKES being a library, it is mainly targeted towards developers and researchers who need to program with Petri nets. However, for the modeller, defining nets

```
1 buffer hello : str = "hello", "salut"
2 buffer world : str = "world", "le␣monde"
3 buffer sentence : str = ()
4 [hello−(h), world−(w), sentence+(h + '␣' + w)]
```

Fig. 4. "Hello world" example revisited in ABCD

using SNAKES directly may be tedious. A user friendly syntax is thus desirable for users that mainly want to build models and explore them. For this purpose, SNAKES comes with a compiler for the ABCD modelling language (Asynchronous Box Calculus with Data) which is a process algebra with friendly Python-like syntax, that embeds full Python, and with a Petri nets semantics (see [26, sec. 3.3] for more details). The compiler translates ABCD code into Petri nets, called from the command line, it can draw the computed net or save it into a file (in SNAKES' PNML dialect) for a later use. It may also be called from a Python program to obtain a net object directly.

The example from Figure 2 could be expressed as shown in Figure 4. We can see that places are expressed as typed buffers (str is Python's type for strings) with an initial content (empty in the case of "sentence"), and transitions are expressed as atomic actions enclosed into square brackets within which the tokens consumed from or produced into buffers are specified. However, ABCD is not designed as a textual syntax for Petri nets and it cannot express any Petri net. Instead, it provides the modeller with a notion of control flow and parametrised processes with local data. This is illustrated in Figure 5 where two producers and two consumers share a buffer bag (defined line 1). Lines 2–4 define a net (which can be considered as a process factory) parametrised by a value mod, two instances of which being created in line 7 with distinct values for mod. Net prod declares a local buffer count, this means that every instance of prod has its own private copy of count. Line 4, the process itself consists of two atomic actions connected by an iteration operator "∗". The left action increments the value in buffer count and produces in bag the current value of count modulo mod. The right action [False] is a special one that can never be executed; because it is used here as the exit of the iteration, process prod is forced to iterate forever producing values in bag. Net cons shows two more features: guards for atomic actions, given after keyword **if**, and sequential composition ";". We can also see the parallel composition "|" in the main process line 7. A fourth composition that is not shown here is the choice "+".

The ABCD compiler also features an interactive simulator that allows step-by-step execution of an ABCD model, directly on the source code, like when using a debugger for a programming language.

3 Efficient Model-Checking

SNAKES is first designed to be flexible and general, not to be efficient: instrumenting Python code from a Python program is definitely not the fastest way

```
buffer bag : int = ()
net prod (mod) :
    buffer count : int = 0
    [count−(x), count+(x+1), bag+(x % mod)] * [False]
net cons (div) :
    ( [bag−(x) if x % div == 0] ; [bag−(x) if x % div != 0] ) * [False]
prod(5) | prod(7) | cons(2) | cons(3)
```

Fig. 5. A producer-consumer example in ABCD

to explore the state space of a Petri net. In order to do this efficiently, one can use tool Neco [14] that is available separately [11]. This tool compiles SNAKES Petri nets into fast native code with an optimised marking structure and per-transition optimised firing. Using the declared place types, it can type the variables on input arcs and generate Cython code [2], a dialect of Python extended with C types. Then, Cython code is compiled into C source code that is finally compiled into native code (all this process is automated). However, note that Neco compiles and optimises Petri nets, not the embedded Python code. So, if a Petri net embeds slow Python code and provides too few types (*e.g.*, in Figure 2 we did not provide any typing for the places, so they are constrained to the universal type object) Cython is forced to rely on the Python interpreter instead of generating fast C code. Neco can also compile ABCD models. In such a case, it exploits many properties of the resulting Petri net that are known by construction (for instance, control flow places are low-level 1-safe places and form 1-invariants on the sequential parts) and performs further optimisation during the compilation.

Apart from its compiler, Neco also features a tool to build the state space of a compiled net, and a tool to perform LTL model-checking on-the-fly. For the latter purpose, it relies on library SPOT [8] that is exactly the complement to Neco: on the one hand, Neco is able to construct a Kripke structure by firing the transitions of the compiled Petri net; on the other hand, SPOT can turn a LTL formula into a Bchi automaton and check on-the-fly the emptiness of its product with the Kripke structure.

Neco was awarded at the Model-Checking Contest 2013 (satellite event of the PETRI NETS conference) as the most efficient explicit LTL model-checker. Moreover, in many cases, it was the only tool to actually provide a result, which assesses its robustness. A tutorial for using Neco is available online [11].

4 SNAKES Out of Python

Using Cython [2] again, it is easy to create a C binding for SNAKES (*i.e.*, export its API to a C library) so it can be called from another programming language. This is not provided by default because there is not one unique binding of SNAKES, but instead one possible binding for every combination of plugins. Fortunately, writing such a binding is easy when we know where the technical difficulties are.

```
# here we write regular Python code to import SNAKES or other
# modules, load plugins, define functions, ...
cdef public int newnet (char *name) :
    # here we just write regular Python code that uses SNAKES
cdef public int addplace (char *net, char *name, int tokens) :
    # and so on...
```

Fig. 6. Cython source code of the binding (file `libsnk.pyx`)

Moreover, one advantage of writing the binding for each such use case is that we are able to produce an API that is exactly suited to our particular need.

The main difficulty is that, when calling SNAKES from another programming language, we shall send references to Python objects outside of the Python runtime. If it happens that an object is no more referenced from the Python runtime, it is garbage collected and the outer reference becomes dangling, which is likely to crash the program with a segmentation fault. To avoid this, we have to provide a storage for the objects with our own references. For instance, we may store net objects and provide access to them through their names.

So, basically, our binding consists of a Cython file `libsnk.pyx` as sketched in Figure 6 (see [22] for the full details). The Cython tool allows to compile this source into a dynamic library (`libsnk.so` under Linux) along with a C header file `libsnk.h` that can be used from a C program. The only constraint is to take care to initialise the Python runtime and the library before to call its functions.

To use SNAKES from another programming language than C, a simple possibility is to rely on SWIG that allows to automatically generate bindings of C libraries for almost 20 programming languages [1].

5 Use Cases

As explained already, it is very hard to have a clear picture of who is using SNAKES because it is freely available and very few users actually ask for support. Fortunately, there are works we known well about [6,7,12,14,15,21,29,30] and that illustrate typical use cases for SNAKES as listed below.

Prototyping tools. A prototype implementation of a massively parallel CTL* model-checking algorithm for ABCD models of security protocols allowed to assess scalability [15]. A new approach to process-symmetry reductions initially defined in [20] has been prototyped in Neco by generating Python code, showing a dramatic performance boost [12], and can be now ported to Cython.

Compilation from/to Petri nets. Neco compiler is entirely implemented in Python using SNAKES to handle the Petri nets and ABCD as an input language [14]. Apart from ABCD, the Petri net semantics of various other formalisms has been implemented using SNAKES, recently: a graphical variant of the π-calculus [30] and a modelling language dedicated to toxic risk assessment in biological and bio-synthetic systems [7].

Modelling. SNAKES is also used to create Petri net models, like in [21] where models of cloud services are represented as token-nets instrumented by a system-net that models the elasticity mechanism. For this task, SNAKES is presumably the only tool available because it allowed to create token-nets whose structures and markings are determined during the firing of the transitions in system-net. More often, an input language is used, like in [7,30], or ABCD is used like in [15]. ABCD has also been used to model peer-to-peer protocols for an industrial case of distributed storage system [6,29].

Analysis. Many modelling works are made with model-checking in perspective, but very often, only reachability analysis is performed to check safety properties and this is surprisingly often made directly using SNAKES [6,21,29]. Neco is also used to speedup state space computation [6] or to perform LTL model-checking [30]. Another kind of analysis is to collect data along a collection of randomly generated traces and to perform statistical analysis, either to assess performances [21] or to evaluate other quantitative information, like in [6] where the number of file loss of a peer-to-peer storage system is evaluated with respect to the percentage of malicious peers present in the system.

6 Conclusion

We have presented SNAKES that enables to develop Petri net tools with great flexibility regarding the variant of Petri nets, and allowing their execution for simulation purpose or for limited reachability analysis. Efficient LTL model-checking can be performed using Neco. SNAKES also ships with a compiler for the ABCD algebra of Petri nets allowing user-friendly modelling of high-level systems.

Ongoing and Future Work. Despite its age and reported stability, SNAKES is still considered as a beta software because it lacks a real development team to meet the standard expectations from a stable software. In particular, it is very hard to provide a roadmap of planned features because they are added in a demand-driven fashion and depend a lot on the time the author can spend. So, current version is 0.9.17 and is slowly converging towards 1.0, which will be reached when at least the following features will be covered:

- replace current PNML support that does not conform to the standards with simpler file formats and rely on third-party tools [17] to handle PNML;
- integrate Neco through a plugin to allow its use transparently and bring LTL model-checking directly to the users;
- fill a few holes in the documentation and perform minor code cleanup and simplification.

This does not mean that no other features will be introduced in the meantime, some in particular are very much desired:

- interactive simulation of any Petri net (in addition to ABCD processes), and fast automatic simulation coupled with statistical analysis of the traces;

– integration with other tools (user interfaces, analysers, etc.), in particular with CosyVerif [17], notably by supporting more inputs/outputs languages;
– genericity with respect to the annotation language by generalising the compilation approach;
– automated API generation to other languages by extending the API documentation extraction tool to generate Cython bindings as presented above;
– extend ABCD with a syntax for raw Petri nets, and with support for thread-like processes as defined in [20] and [26, sec. 4.3].

Interface with other tools and integration with other programming languages are two crucial features to open SNAKES to more researches out of the Python ecosystem. It looks very useful not to limit its use to one particular programming language so it can be helpful for a broader community.

Related Works. To the best of our knowledge, SNAKES is quite a unique tool in that there is no other such general purpose Petri net library aimed at tools developers, that is still actively developed and maintained. The Petri Net Kernel [19] used to have similar goals for Java or Python, depending on the version, but it received no update since October 2003.

Taking apart its purpose and considering only the Petri net variant proposed in SNAKES, we may find similarities with other high-level Petri nets tools. In particular, the coloured Petri nets [18] implemented in CPN tools [16] are also Petri nets annotated with a programming language which is a variant of ML in this case. A variant of coloured Petri nets coloured with the Haskell programming language was proposed in [28] but the project appears stopped since 2004. The TINA toolbox [3] supports interfacing with C code, allowing to implement guards for the transitions of a time Petri net, and to perform computation on transition firing. But this is quite far from providing C-coloured Petri nets because the net and C parts remain separated and the data is attached to the state instead of to the tokens, which is a serious limitation from a modelling perspective.

When considering Neco together with SNAKES, it becomes relevant to compare with explicit model-checkers for high-level Petri nets. CPN tools cited above can perform CTL-like model-checking on a fully computed state space, while Neco uses SPOT to perform LTL model-checking on-the-fly. This is similar to Helena [9] that also works with Petri nets annotated with an ad-hoc language; moreover, like Neco, Helena compiles the Petri net into C code in order to speedup transitions firing. However, this compilation is limited to the annotations and the marking, but not generalised to the whole Petri net structure like in Neco.

References

1. Beazley, D.M., Fulton, W.: SWIG – Simplified Wrapper and Interface Generator. http://www.swig.org
2. Behnel, S., Bradshaw, R., Dalcín, L., Florisson, M., Makarov, V., Seljebotn, D.S.: Cython – C-extensions for Python. http://cython.org

3. Berthomieu, B., Vernadat, F.: Time Petri nets analysis with TINA. In: Proc. of QEST 2006. IEEE Computer Society (2006)
4. Best, E., Devillers, R., Koutny, M.: Petri net algebra. Springer (2001)
5. Bilgin, A., Ellson, J., Gansner, E., Hu, Y., North, S.: Graphviz – Graph Visualization Software. http://graphviz.org
6. Chaou, S., Utard, G., Pommereau, F.: Evaluating a peer-to-peer storage system in presence of malicious peers. In: Proceedings of HPCS 2011. IEEE Computer Society (2011)
7. Di Guisto, C., Klaudel, H., Delaplace, F.: Systemic approach for toxicity analysis. In: Proc. of BioPPN 2014. Workshop Proceedings, vol. 1159. CEUR (2014)
8. Duret-Lutz, A.: LTL translation improvements in Spot. In: Proc. of VECoS 2011. Electronic Workshops in Computing, British Computer Society (2011)
9. Evangeliste, S.: HELENA, a high level net analyzer. http://lipn.univ-paris13.fr/evangelista/helena
10. Free Software Foundation: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
11. Fronc, Ł.: Neco net compiler. http://code.google.com/p/neco-net-compiler
12. Fronc, Ł.: Effective marking equivalence checking in systems with dynamic process creation. In: Proc. of Infinity 2012. Electronic Proceedings in Theoretical Computer Science (2012)
13. Fronc, Ł., Duret-Lutz, A.: LTL model checking with neco. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 451–454. Springer, Heidelberg (2013)
14. Fronc, Ł., Pommereau, F.: Building Petri nets tools around Neco compiler. In: Proc. of PNSE 2013 (2013)
15. Gava, F., Pommereau, F., Guedj, M.: A BSP algorithm for on-the-fly checking CTL* formulas on security protocols. The Journal of Supercomputing (2014)
16. Group, T.C.: CPN tools. http://cpntools.org
17. Haddad, S., Kordon, F., Petruci, L.: CosyVerif. http://cosyverif.org
18. Jensen, K., Kristensen, L.M.: Coloured Petri Nets, Monographs in Theoretical Computer Science, vol. 2. Springer (1997)
19. Kindler, E., Weber, M.: The Petri Net Kernel: An infrastructure for building Petri net tools. Software Tools for Technology Transfer 3 (1999)
20. Klaudel, H., Koutny, M., Pelz, E., Pommereau, F.: State space reduction for dynamic process creation. Scientific Annals of Computer Science 20 (2010)
21. Mohamed, M., Amziani, M., Belaïd, D., Tata, S., Melliti, T.: An autonomic approach to manage elasticity of business processes in the Cloud. Future Generation Computer Systems (2014) (To appear)
22. Pommereau, F.: SNAKES out of Python. http://www.ibisc.univ-evry.fr/fpommereau/SNAKES/snakes-out-of-python.html
23. Pommereau, F.: Quickly prototyping Petri nets tools with SNAKES. Petri net newsletter 10 (2008)
24. Pommereau, F.: Quickly prototyping Petri nets tools with SNAKES. In: Proc. of PNTAP 2008. ACM Digital Library. ACM (2008)
25. Pommereau, F.: Nets in nets with SNAKES. In: Proc. of MOCA 2009. Universität Hamburg, Dept. Informatik, Hamburg (2009)
26. Pommereau, F.: Algebras of coloured Petri nets. Lambert Academic Publishing (2010)
27. Python Software Foundation: Alternative Python implementations. http://www.python.org/download/alternatives

28. Reinke, C.: Haskell-coloured petri nets. In: Koopman, P., Clack, C. (eds.) IFL 1999. LNCS, vol. 1868, pp. 165–180. Springer, Heidelberg (2000)
29. Sanjabi, S., Pommereau, F.: Modelling, verification, and formal analysis of security properties in a P2P system. In: Proceedings of COLSEC 2010. IEEE Digital Library. IEEE Computer Society (2010)
30. Van Pham, V.: Modelling and analysing open reconfigurable systems. Ph.D. thesis, Univ. Évry / Paris-Saclay (2014)