Building Petri nets tools around Neco compiler

Łukasz Fronc and Franck Pommereau
{fronc,pommereau}@ibisc.univ-evry.fr

IBISC, Université d'Évry/Paris-Saclay
IBGBI, 23 boulevard de France
91037 Évry Cedex, France

Abstract. This paper presents Neco that is a Petri net compiler: it takes a Petri net as its input and produces as its output an optimised library to efficiently explore the state space of this Petri net. Neco is also able to work with LTL formulae and to perform model-checking by using SPOT library. We describe the components of Neco, and in particular the exploration libraries it produces, with the aim that one can use Neco in one's own projects in order to speedup Petri nets executions.

Keywords: Petri nets compilation, optimised transition firing, tools development, explicit state space exploration

1 Introduction

Neco is a Petri net compiler: it takes a Petri net as its input and produces as its output a library allowing to explore the Petri net state space. Neco operates on a very general variant of high-level Petri nets based on the Python language (*i.e.*, the values, expressions, etc., decorating a net are expressed in Python) and including various extensions such as inhibitor-, read- and reset-arcs. It can be seen as coloured Petri nets [1] but annotated with the Python language instead of the dialect of ML as it is traditional for coloured Petri nets.

Firing transitions of such a high-level Petri net can be accelerated by resorting to a compilation step: this allows to remove most of the data structures that represent the Petri net by inlining the results of querying such structures directly into the generated code. Moreover, instead of relying on generic data structures and algorithms, specialisations can be performed on a per-transition and per-place basis. In particular, Neco can exploit various properties of Petri nets in order to further optimise the representation of a marking (both for execution time and memory consumption), as well as transitions firing algorithms. Finally, Neco is able to type most of the Python code embedded in a Petri net thanks to the typing information on places. This allows to generate efficient C++ code instead of relying on the interpreted Python code.

All this yields a substantial speedup in transition firing (and consequently in state-space exploration and explicit model-checking) that was evaluated in [2]. This was also confirmed by the participation of Neco to the *model-checking*

contest (satellite event of the Petri nets 2012 conference) that showed that Neco was able to compete with state-of-the-art tools [3].

The goal of this paper is to introduce the main concepts of Neco and its usage in order to enable tool developers for efficiently using the detailed online documentation and exploit Neco in their own projects. This may concern most tools that perform explicit exploration of Petri nets states spaces and take advantage of the speedup that Neco can offer.

Neco is free software released under the Gnu lgpl and it can be downloaded from `http://code.google.com/p/neco-net-compiler` where its documentation is also available, including a tutorial as well as the precise api of libraries generated by Neco and concrete examples.

2 General architecture and usage guidelines

Neco is a collection of two compilers, one exploration tool and one model-checker:

- `neco-compile` is the main compiler that produces an exploration engine of a Petri net (a library);
- `neco-explore` is a simple exploration tool that computes state spaces using the engine produced by `neco-compile`;
- `neco-check` is compiler for LTL formulae that produces a library to handle these formulae;
- `neco-spot` is a LTL model-checker that uses outputs of tools `neco-compile` and `neco-check`, as well as SPOT library for model-checking algorithms [4].

As a compiler, Neco has two backends: the *Python backend* allows to generate Python code while the *Cython backend* generates annotated Python [5] that can be compiled to C++. Each tool composing Neco is dedicated to a specific task. Here we *focus on compilation* but we will also say a few words about the rest. The detailed compilation workflow is shown in Figure 1. In this section we assume that we use the *Cython backend* which is the most efficient one. First we present how the exploration engine is built and how to use it to build state-spaces, this part remains globally valid for the *Python backend*. Next we present how to perform LTL model-checking within Neco, and this part is currently *not supported* by the Python backend. However, there are also features that are currently only available in the Python backend, like *reductions by symmetries* [6], thus not yet available for LTL model-checking.

2.1 Exploration engine builder and state-space construction

The first step using Neco is to create a module that provides exploration primitives: a marking structure, successor functions specific to transitions, and a global successor function that calls the transition specific ones [2]. As shown in Figure 2, this exploration engine can be used by a client program (*e.g.*, a model-checker or a simulator) to perform its task. The generated library directly

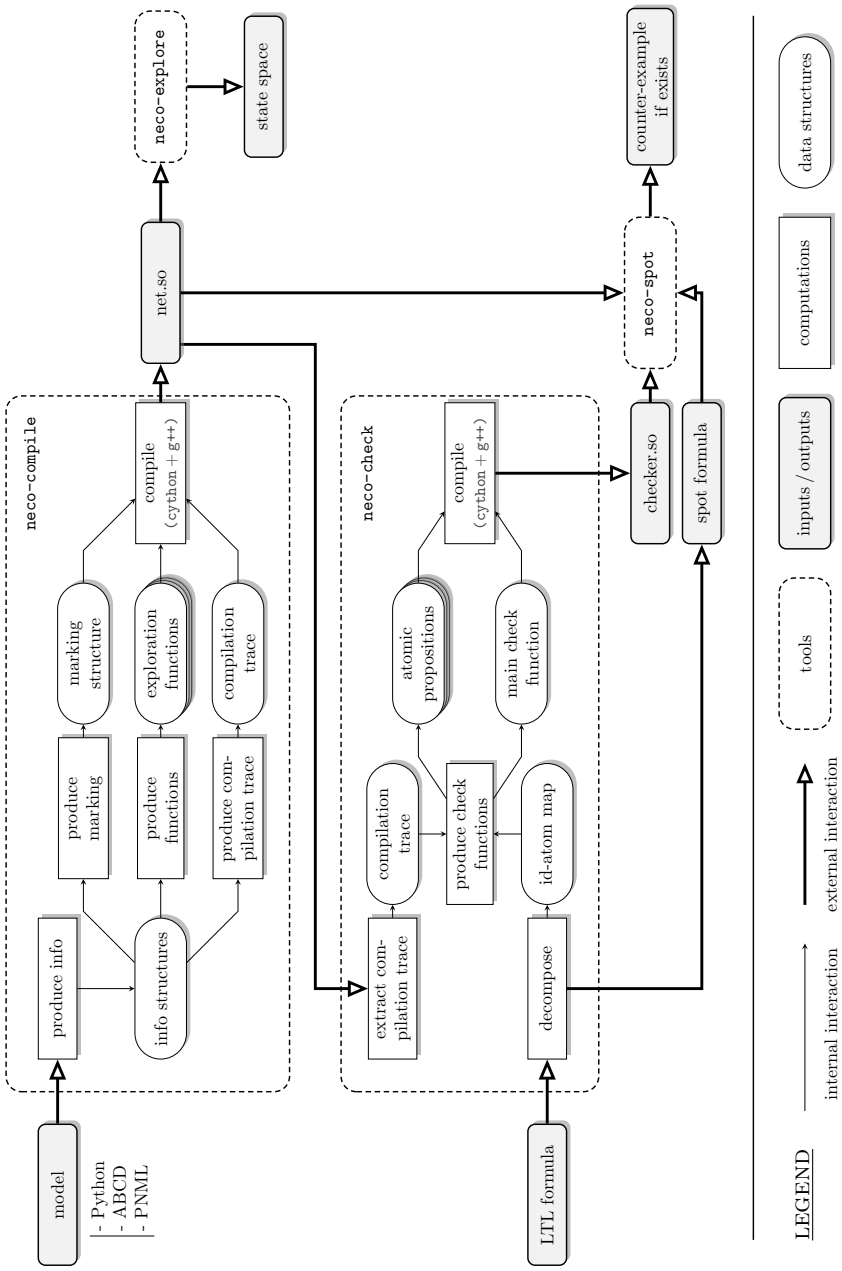

Fig. 1. Compilation pipeline and exploration tools within Neco (Cython backend).

embeds code from the model (*i.e.*, Petri net annotations) but also relies on existing data structures (in particular, sets and multisets) forming core libraries, and accesses them through normalized interfaces. Model code itself has very few constraints and may use existing libraries. This is detailed in [2].

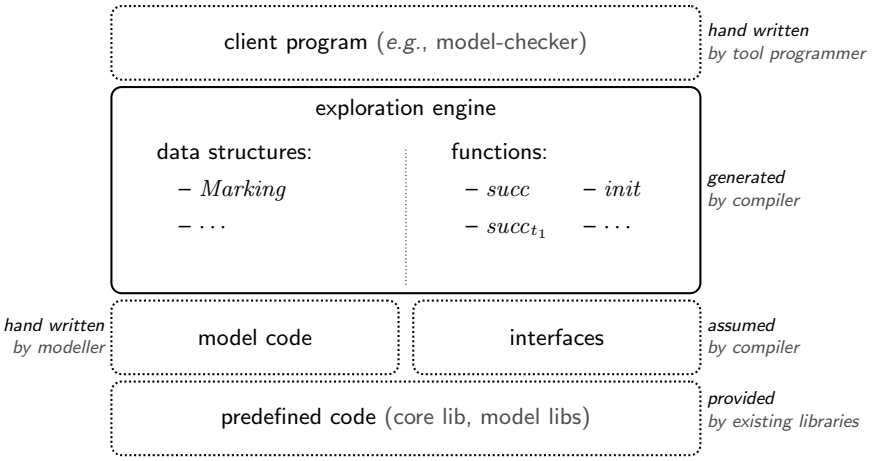

Fig. 2. The exploration engine (plain-line box) and its context (dotted boxes). [2]

This module is built using command `neco-compile`. To do so, Neco takes a Petri net model as input which can be described programmatically in Python using the *SNAKES* toolkit [7], or using the *ABCD* formalism [8], or specified in *PNML* [9]. Once the model is loaded, some types are *inferred* allowing to statically type Python code later, which is an important feature because Cython language can produce optimized C++ code from annotated Python code [5]. However, because we allow a high degree of expressivity, all source code cannot be typed and Neco falls back to calling the Python interpreter in such cases. Basically, if a net contains only black tokens, integers or Boolean values, and static strings, as well as collections (tuples, lists, sets, dictionaries) of such values, it will be fully translated into C++.

The next step is to produce a *marking structure* to represent Petri net states. It is optimized based on previously discovered types. This allows to use native types or to generate per-place specialised implementations. Then, we can produce *exploration functions* specific to the model (mainly an initial marking function and successor functions), this allows to efficiently produce state spaces [2].

An additional step is to produce a *compilation trace* which contains information about the marking structure and the model. This metadata is essential for consistency preservation among tools, and it prevents the user from having to call each tool with exactly the same options which is error-prone.

The last step is to compile generated code producing a *native Python module* that is a *shared library* which can be used from C++ as a regular library as well as from Python as a regular module. This is actually done with *Cython compiler* and a C++ compiler.

State spaces can be built using `neco-explore` tool. This tool builds sets of reachable states, and reachability graphs using a simple exploration algorithm that aggregates discovered states by repeatedly calling successor functions.

2.2 LTL model checking

LTL model checking is performed using *SPOT library* [4], however, SPOT cannot directly handle atomic propositions appearing in LTL formulae which are specific to the used formalism. Moreover, because our marking structures are model specific, we also need to generate an atomic proposition checker module for each compiled net. This is made by `neco-check` compiler.

This tool takes two inputs, a LTL formula in Neco compatible LTL syntax [10], and compilation metadata extracted from an exploration module (previously created with `neco-compile`).

The first step is to decompose the formula, extract atomic propositions and map them to unique identifiers ("id-atom map" on Figure 1). A simplified formula where all atomic propositions have been replaced by these identifiers is stored as a file. This way, atomic propositions can be abstracted away leading to a simple interface with the checking module. Basically, the interface is a function `check` that takes a state and an atomic proposition identifier, and returns the truth value of the atomic proposition at the provided state.

The next step is the creation of one check function for each atomic proposition, plus the generic check function exposed to users. During this step, using the compilation trace is essential because we need to create functions that are compatible with the optimized marking structure, and thus be aware of used types and memory layout. Finally the generated code is compiled using *Cython compiler* and a C++ compiler.

The checker module finalized, it can be used together with the formula file by `neco-spot` tool and it will output a counter-example if one exists, *i.e.*, if the formula is not satisfied.

3 Perspectives

Several new features are already planned for Neco. First, a method to reduce symmetries based on [6] has been already prototyped in the Python backend. We would like to implement it in the Cython backend also to achieve better performance. Next, Neco will be adapted to compute unfoldings *à la* McMillan using the approach described in [11, chap.6]. This should be feasible by reusing most of the code that Neco already generates to discover bindings. Finally, we would like to implement fast simulation in Neco, which could be a variant of the current exploration algorithm that would compute only one successor for a

state instead of all its successors. However, for better performance, we would like to experiment with a co-routine based implementation of Python [12] in order to define a highly concurrent architecture while avoiding the overhead of using threads.

Neco will also participate to the 2013 edition of the model-checking contest. As a side effect, this will lead us to develop new case studies for Neco (*i.e.*, those models that are included in the contest), which will be extended later with more case studies.

Based on case studies, we would like to perform extensive benchmarks of the Cython backend by comparing it to a combination of the Python backend with various Python compilers (in particular [13] and [14]) as well as with PyPy implementation of Python that features efficient just-in-time compilation [15]. This should allow either to drop Cython backend if it happens that it is outperformed by other approaches, or, more probably, to define typical situations where Cython should not be used. In particular, we expect PyPy to be more efficient on Petri nets that embed a lot of Python objects that cannot be converted to efficient C++ code.

Finally, we are working on an additional Java backend, allowing to compile Petri nets and LTL formulae to Java code. This will require some internal reorganisation of Neco so its core will become language-agnostic while only the backends will have to deal with language-specific aspects. Thanks to this work, we expect that more backends will be implemented in the future to handle Petri nets annotated with a wider variety of languages.

References

1. Jensen, K., Kristensen, L.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer (2009)
2. Fronc, Ł., Pommereau, F.: Optimizing the compilation of Petri nets models. In: Proc. of SUMo'11. Volume 726., CEUR (2011)
3. Kordon, F., Fronc, Ł., Pommereau, F., et al.: Raw Report on the Model Checking Contest at Petri Nets 2012. Technical report (2012)
4. Duret-Lutz, A.: LTL translation improvements in Spot. In: Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'11). Electronic Workshops in Computing, Tunis, Tunisia, British Computer Society (2011)
5. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., Smith, K.: Cython: The best of both worlds. Computing in Science Engineering **13** (2011) 31 –39
6. Fronc, Ł.: Effective Marking Equivalence Checking in Systems with Dynamic Process Creation. In: Infinity'12. Electronic Proceedings in Theoretical Computer Science, Paris (2012)
7. Pommereau, F.: Quickly prototyping Petri nets tools with SNAKES. Petri net newsletter (2008)
8. Pommereau, F.: Algebras of coloured Petri nets. LAP LAMBERT Academic Publishing (2010)
9. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In: 10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools (CPN'09). (2009)

10. Fronc, Ł.: Neco net compiler wiki. `goo.gl/CXrry` (2012)
11. Khomenko, V.: Model checking based on Petri net unfolding prefixes. PhD thesis, PhD thesis, School of Computer Science, University of Newcastle upon Tyne (2002)
12. Tismer, C.: Continuations and stackless Python. In: Proceedings of the 8th International Python Conference. (2000)
13. Dufour, M.: Shed skin. `http://code.google.com/p/shedskin` (2012)
14. Hayen, K.: Nuitka. `http://nuitka.net` (2012)
15. Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: PyPy's tracing JIT compiler. In: Proc. ICOOOLPS '09, ACM (2009)