LTL Model Checking with Neco

Łukasz Fronc[1] and Alexandre Duret-Lutz[2]

[1] IBISC, Université d'Évry/Paris-Saclay
fronc@ibisc.univ-evry.fr
[2] LRDE, EPITA, Kremlin-Bicêtre, France
adl@lrde.epita.fr

Abstract. We introduce neco-spot, an LTL model checker for Petri net models. It builds upon Neco, a compiler turning Petri nets into native shared libraries that allows fast on-the-fly exploration of the state-space, and upon Spot, a C++ library of model-checking algorithms. We show the architecture of Neco and explain how it was combined with Spot to build an LTL model checker.

1 Introduction

Neco is a suite of Unix tools to compile high-level Petri net models into shared libraries that can then be used to check reachability properties (building only the set of reachable states), or check any LTL property (synchronizing the reachability graph with a property automaton). It is based on SNAKES, a general Petri net Python library [12], which key feature is the use of arbitrary Python objects as tokens and Python expressions as net annotations. This allows a great amount of expressivity at the cost of slow execution times, Python being an interpreted language. Neco uses this library as a frontend allowing this high degree of expressivity but also notably speeds up the execution, efficiently compiling the models to native libraries. This compilation step allows Neco to compete with state-of-the-art tools [7,10].

Originally, Neco did only reachability analysis. In this paper, we explain how we connected it with the Spot library to perform LTL model checking. Beside presenting Neco, this paper can therefore be seen as presenting a use-case of Spot, showing how to build an LTL model checker for a custom formalism.

2 Architecture of Neco

To perform model-checking, Neco provides three tools: `neco-compile`, `neco-check`, and `neco-spot`. Each of these tools handle a specific task and the whole tool set allows for a simple workflow as presented in Figure 1.

First, `neco-compile` builds an exploration engine (`net.so`) from a high-level Petri net model. The model can be programmatically specified in Python using the SNAKES toolkit [12], specified in the ABCD formalism [11], or provided in PNML format [9]. This step uses model specific information (inferred or provided by the user) to generate optimized data structures and exploration functions on a per-model basis [7,8].

Next, we set up an atomic proposition checker. Because Spot is a general model-checking library, it does not provide a language for atomic propositions. So each tool

D. Van Hung and M. Ogawa (Eds.): ATVA 2013, LNCS 8172, pp. 451–454, 2013.
© Springer International Publishing Switzerland 2013

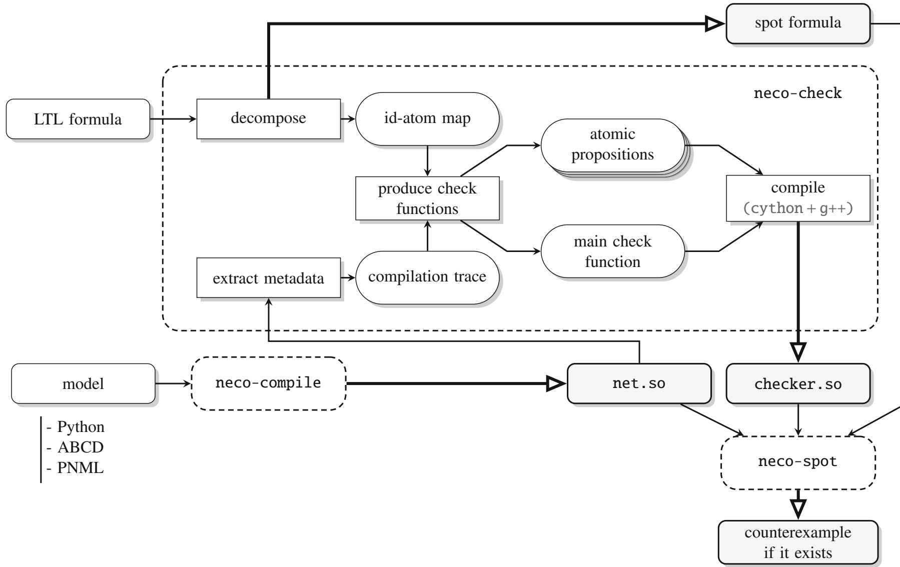

Fig. 1. The architecture of Neco

using Spot has to provide its atomic proposition language, but also functions to check these. This is the role of `neco-check` tool. It takes a LTL formula as an input, then decomposes it in order to extract atomic propositions. During this step a simplified formula where all atomic propositions were replaced by simple identifiers is produced (`spot formula`). The tool keeps the track of these atoms using an identifier-atomic proposition map, which can also be used to understand the simplified formula. The exploration engine being model-specific, `neco-check` cannot make any assumption about the Petri net marking structure or memory layout. Fortunately `net.so` exports some metadata (`compilation trace`) about the marking structure that is used by `neco-check` to generate check functions for each atomic proposition. The last function we produce is a main check function that serves as an interface for the whole module. It returns the value of atomic propositions based on their identifiers and a provided states. All functions generated, we can compile the code producing the shared library `checker.so`.

The model-checking procedure is performed by the third and last tool: `neco-spot`. This tool takes as inputs: the LTL formula to check (`spot formula`), the exploration engine library to build the reachability graph on demand (`net.so`), and the atomic proposition checker module to check atomic proposition values (`checker.so`). Then using the Spot library outputs a counterexample if one exists, and builds the whole state space otherwise.

3 Bridge between Neco and Spot

We now describe how we built our LTL model-checking tool, `neco-spot`, combining Neco's exploration engine with the model-checking algorithms of Spot [4].

Spot handles Transition-based Generalized Büchi Automata (TGBA), which, as the name suggests are Büchi automata with transition-based generalized acceptance conditions. TGBA allows for more compact representation of LTL properties [3], and can be checked for emptiness efficiently [2]. The TGBA is also an abstract C++ class, `tgba`, with an interface that allows on-the-fly exploration. Kripke structures are viewed as a subclass of `tgba` without acceptance sets.

The automata-theoretic approach is implemented by `neco-spot` as follows:

1. A wrapper of `net.so` and `checker.so` that presents the reachability graph of the model as a subclass of Spot's `kripke` class. The interface boils down to three functions: `get_init_state()` returns the initial state, `succ_iter(s)` returns an iterator on the successors of the state `s`, and `state_condition(s)` returns the valuation of the atomic propositions for the state `s`. Note that this interface allows an on-the-fly exploration of the state space, computing the results of `succ_iter(s)` and `state_condition(s)` on demand, by simply calling the relevant functions compiled in `net.so` and `checker.so`.
2. The LTL formula is simplified, converted into a TGBA, which is in turn also simplified. All these operations are functions offered by Spot [3].
3. The previous two automata are synchronized using the class `tgba_product` of Spot (another subclass of `tgba`). This synchronous product object is actually constructed in constant time, and delays its computation until it is actually explored.
4. The synchronous product is checked for emptiness using any of the emptiness check algorithms implemented by Spot [4]. It is this emptiness check procedure that will trigger the on-the-fly computation of the product, which will in turn construct the part of the reachability graph that need to be explored.
5. If the product was empty, a counterexample is computed and displayed.

The most important part of the work for building `neco-spot` therefore consisted in implementing the interface for Spot's `kripke` class; the rest is just chaining calls to various algorithms of Spot.

4 Possible Evolutions

There are a couple features of Spot that we do not use in `neco-spot`, and that will constitute some easy extensions.

A first one is the support of the linear fragment of the Property Specification Language [1] (PSL), a superset of LTL. Spot has built-in support for PSL, and all it would require is an extension of Neco's parser of formulas.

A second extension would be to support for weak fairness properties [5] in the model. Currently, `neco-spot` presents its model as an instance of the `kripke` class, which is just a TGBA without acceptance conditions, but it could present the model as a `fair_kripke` where states can be associated to acceptance sets representing weak fairness constraints.

We also plan to add reductions by symmetries [6] which have been already prototyped in Python, but are not available for LTL model checking yet. This would improve both exploration times and state-space sizes, leading to smaller product automata when performing model checking with Spot.

Furthermore, in order to easily debug models, we would like to implement fast simulation within Neco. This would also allow to replay couterexamples provided by `neco-spot`.

5 Availability

Neco is free software. Documentation and installation instructions can be found at

`http://code.google.com/p/neco-net-compiler/`.

A test-suite is also supplied.

References

1. Property Specification Language Reference Manual v1.1. Accellera (June 2004), http://www.eda.org/vfv/
2. Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized büchi automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)
3. 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 2011). Electronic Workshops in Computing, British Computer Society, Tunis (2011), http://ewic.bcs.org/category/15853
4. Duret-Lutz, A., Poitrenaud, D.: SPOT: An Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In: Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), pp. 76–83. IEEE Computer Society Press, Volendam (2004)
5. Francez, N.: Fairness. Springer (1986)
6. Fronc, Ł.: Effective marking equivalence checking in systems with dynamic process creation. In: Proceedings of the 14th International Workshop on Verification of Infinite-State Systems (Infinity 2012), Paris. EPTCS (August 2012)
7. Fronc, Ł., Pommereau, F.: Optimizing the compilation of Petri Nets models. In: Proceedings of the Second International Workshop on Scalable and Usable Model Checking for Petri Net and other Models of Concurrency (SUMO 2011), vol. 726. CEUR (2011)
8. Fronc, Ł., Pommereau, F.: Building Petri Nets tools around Neco compiler. In: Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE 2013), Milano, vol. 989. CEUR (June 2013)
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: Proceedings of the 10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools, CPN 2009 (October 2009)
10. Kordon, F., et al.: Raw Report on the Model Checking Contest at Petri Nets 2012. CoRR abs/1209.2382 (2012)
11. Pommereau, F.: Algebras of coloured Petri Nets. LAP LAMBERT Academic Publishing (2010)
12. Pommereau, F.: Quickly prototyping Petri Nets tools with SNAKES. Petri Net Newsletter (October 2008)