PeCAn: Compositional Verification of Petri Nets Made Easy*

Dinh-Thuan Le[1], Huu-Vu Nguyen[1], Van-Tinh Nguyen[1], Phuong-Nam Mai[1], Bao-Trung Pham-Duy[1], Thanh-Tho Quan[1], Étienne André[2], Laure Petrucci[2], and Yang Liu[3]

[1] HoChiMinh City University of Technology, Vietnam
[2] Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, Villetaneuse, France
[3] Nanyang Technological University, Singapore

Abstract. This paper introduces PeCAn, a tool supporting compositional verification of Petri nets. Beyond classical features (such as on-the-fly analysis and synchronisation between multiple Petri nets), PeCAn generates Symbolic Observation Graphs (SOG), and uses their composition to support modular abstractions of multiple Petri nets for more efficient verification. Furthermore, PeCAn implements an incremental strategy based on counter-examples for model-checking, thus improving significantly the cost of execution time and memory space. PeCAn also provides users with the visualisation of the input Petri nets and their corresponding SOGs. We experimented PeCAn with benchmark datasets from the Petri Nets' model checking contests, showing promising results.

Keywords: Compositional verification, Petri nets, SOG.

1 Introduction

A Petri net (PN) [7] is a graphical mathematical language which efficiently supports the modelling and verification of distributed systems. Basically, a Petri net is a directed bipartite graph, featuring transitions and places. As Petri nets are widely used in research and industry communities, there are several tools developed to help users specify and verify Petri nets, in particularly LoLa [10], Snoopy [5], TAPAAL [3], CosyVerif [1], CPN Tools [12] or JPetriNet[1]. Although most of the tools work with basic place/transitions PNs, some of them cater for some advanced forms of PNs such as timed, coloured, or stochastic PNs.

In this paper, we present PeCAn (Petri net Compositional Analyser), a tool supporting verification of Petri nets in a compositional manner. PeCAn can take as input Petri Net models described in PNML, one of the most popular languages

* This work is partially supported by the STIC-Asie project CATS ("Compositional Analysis of Timed Systems").

[1] `http://jpetrinet.sourceforge.net`

F. Cassez and J.-F. Raskin (Eds.): ATVA 2014, LNCS 8837, pp. 242–247, 2014.

to describe Petri Nets nowadays. The properties to be checked are expressed as LTL formulae. PeCAn offers the following features:

- PeCAn allows users to compose a complex PN from multiple concurrent PNs and then verify the composed PN against a given property.
- PeCAn is able to generate Symbolic Observation Graphs (SOG) [4] from the actual PNs. Therefore, PeCAn supports verification of modular PNs by composing SOGs of separate components.
- PeCAn implements the incremental strategy based on counter-examples when verifying the generated SOG [2]. Thus, the cost of execution time and memory space is significantly reduced.

2 Modular Verification

In this section, we take the example presented in [8] to demonstrate how to use PeCAn to verify Petri nets. Even though PeCAn can verify a single Petri net as other existing tools do, in this paper we only focus on compositional verification of PeCAn, i.e. verifying a Petri net composed by multiple synchronised modules.

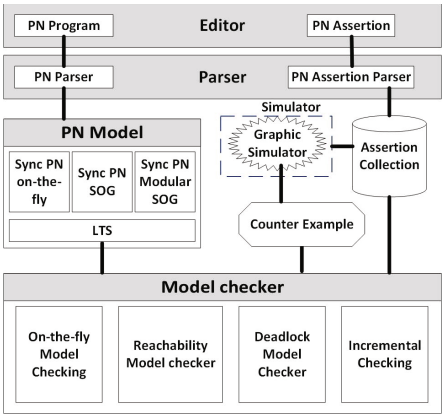
We assume that the original Petri net is already decomposed by users into modules. PeCAn allows users to verify an arbitrary composition of predefined modules. In order to do so, they must define synchronised transitions by the same name between modules. Figure 1b gives an example of a system decomposed into three modules through synchronised transitions. This system can be described easily in a modular style by PeCAn. In this example, modules A and B have two transitions with the same name (F1, F3) meaning that these two transitions must be synchronised. Similarly, a synchronised transition, F2, is shared by modules B and C, also declared by the same name in PeCAn.

When the module composition and the LTL property are defined, users can choose to perform the verification using one of the following methods:

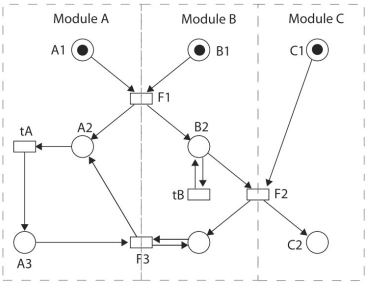
Basic LTL Verification. The modules are synchronised together based on the user specification. Then the synchronised modules are converted into an LTS model and verified on-the-fly by the PAT model checking library [11].

SOG-based Verification. In this method, we do not directly verify the synchronised modules. Instead, we produce a corresponding SOG and use it for the verification. If a counter-example is found, it is verified again on the original Petri net to check whether it is an actual counter-example.

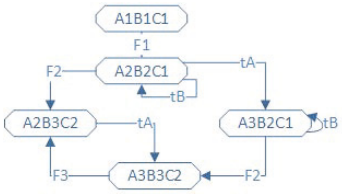
Incremental SOG-based Verification. It is similar to the SOG-based Verification method. However, we do not generate the SOG for the whole synchronisation of modules. Instead, we incrementally synchronise two modules first and verify the corresponding SOG. If no counter-example is found, we incrementally synchronise one more module and repeat the SOG-based verification step, until a counter-example is found or all modules are synchronised and verified (see [2]).



(a) Architecture of PeCAn



(b) A modular PN example



(c) Full state space of the model

Fig. 1. Architecture of PeCAn and example with state space

3 Architecture

The architecture of PeCAn, given in Figure 1a, is described as follows:

Editor Layer. Allows users to describe PNs by a (i) PNML specification or (ii) graph-based visualisation. Users can design an arbitrary number of modules as well as any composition between them.

Parser Layer. Parses the architectures of the PNs from the Editor Layer and converts the Petri net models as well as properties to check into an internal representation for the Semantic Layer.

Semantic Layer. Responsible for generating the corresponding LTS of the input Petri nets, in order to be model checked by the next layer. The three approaches of Basic LTL Verification, SOG-based Verification and Incremental SOG-based Verification are then implemented as three sub-modules: *Sync PN on-the-fly*, *Sync PN SOG* and *Sync PN Modular SOG*.

Model Checker Layer. We make use of the PAT model checking library [11] for this layer. This library takes an LTS as input, and verifies the properties.

4 Functionality Comparison and Experiments

We finally present some comparative discussion and experiments of our tool with other similar approaches. Since PeCAn takes PNML as input, we collected

Table 1. Available tools that support PNML models

No	Tool	PNML format supported	GUI editor	Deadlock checking	User-defined LTL checking	Simulation
1	PeCAn	✓	✓	✓	✓	✓
2	PNEditor[2]	✓	✓	×	×	
3	Snoopy[3]	×[4]	✓	×	×	✓
4	PNML Framework[5]	✓	✓	?[6]		
5	ProM framework[7]	✓	✓	×	?[8]	×
6	P3[9]	✓		×		
7	ePNK[10]	✓	✓	?[11]		
8	Tina[12]	✓	×	×	✓	✓

other PN verification tools also supporting PNML. We selected the tools listed at `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/` and supporting PNML. As shown in Table 1, very few tools can support PNML specification and perform full LTL verification.

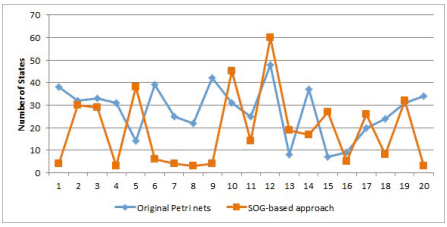
We then experimented PeCAn with benchmark datasets downloaded from the Model Checking contest [6][13]. Results, as display in Table 2 showed that PeCAn can endure some remarkably large model sizes. When a counter-example is found, PeCAn can terminate quickly with significantly less resources usage.

Lastly, we also compared the performance of PeCAn in terms of the (symbolic) states and transitions generated by the SOG-based approach. The results are presented in Figures 2a and 2b respectively. Results show that the SOG-based approach of PeCAn usually reduces the number of states, and always significantly reduces the number of transitions when compared to the standard approach. In fact, the number of generated transitions is always significantly reduced, leading to a substantial gain of time when applying a model checking algorithm. The tool and all experiments can be downloaded from [9].

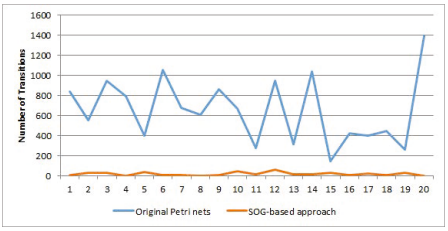
[2] `http://www.pneditor.org/download/pneditor-0.64.jar`
[3] `http://www-dssz.informatik.tu-cottbus.de/track/download.php?id=136`
[4] Claimed as coming soon
[5] `http://pnml.lip6.fr`
[6] Depends on analysis tool using PNML Framework
[7] `http://www.promtools.org/prom6/`
[8] Claimed to be done via plugins, but we could not find where.
[9] `http://www.sfu.ca/~dgasevic/projects/P3net/Download.htm`
[10] `http://www.imm.dtu.dk/~ekki/projects/ePNK/`
[11] It could not load Eclipse after installation
[12] `http://projects.laas.fr/tina//download.php`
[13] `http://mcc.lip6.fr/`

Table 2. Experiments with deadlock models: PeCAn does not need to explore the whole state space

No	Model	Parameter	State space	Number of Reached Markings	Number of Transition Firings	Time (s)	Memory (KB)
1	CSRepetitions	2	7.424	32	31	0.015	8,962
2	CSRepetitions	3	1.341×10^8	117	116	0.078	10,465
3	CSRepetitions	4	unknown	291	290	0.202	16,033
4	CSRepetitions	5	unknown	1,274	1,283	0.642	43,289
5	CSRepetitions	7	unknown	2,148	2,147	2.367	135,809
6	CSRepetitions	8	unknown	7,242	7,241	20.836	1,056,194
7	Eratosthenes	10	32	12	19	0.191	8,635
8	Eratosthenes	20	2,048	28	60	0.015	8,929
9	Eratosthenes	50	1.718×10^{10}	287	821	0.071	11,451
10	Eratosthenes	100	1.899×10^{22}	1,236	4,099	0.539	23,446
11	Eratosthenes	200	1.142×10^{46}	3,614	13,007	4.794	91,365
12	Eratosthenes	500	4.13×10^{121}	24,236	88,363	76.082	899,525
13	HouseContruction	2	1,501	74	73	0.119	9,100
14	HouseContruction	5	unknown	209	208	0.031	10,095
15	HouseContruction	10	$1,664 \times 10^9$	434	433	0.018	10,354
16	HouseContruction	20	1.367×10^{13}	884	883	0.052	13,481
17	HouseContruction	50	unknown	2,234	2,233	0.121	17,747
18	HouseContruction	100	unknown	4,484	4,483	0.294	20,059
19	HouseContruction	200	unknown	8,984	8,983	0.471	32,975
20	HouseContruction	500	unknown	22,484	22,483	1.48	63,711
21	PermAdmissibility	1	52,537	41	40	0.183	10,437
22	PermAdmissibility	2	unknown	253	252	0.098	13,243
23	PermAdmissibility	5	unknown	1,025	1,024	0.363	25,011
24	PermAdmissibility	10	unknown	2,372	2,371	0.869	45,072
25	PermAdmissibility	20	unknown	5,027	5,026	2.021	87,138
26	PermAdmissibility	50	unknown	12,912	12,911	4.901	201,224
27	Philosopher	5	243	68	84	0.007	9,274
28	Philosopher	10	59,049	7,242	10,576	1.057	42,935
29	Philosopher	20	3.487×10^9	Time out after 7200s			



(a) Number of states



(b) Number of transitions

Fig. 2. Experimental results on a set of Petri nets

Acknowledgments. All our thanks to the PAT team [11] for their help in interfacing our tool with the PAT library.

References

1. André, É., Hillah, L.-M., Hulin-Hubard, F., Kordon, F., Lembachar, Y., Linard, A., Petrucci, L.: CosyVerif: An open source extensible verification environment. In: ICECCS, pp. 33–36. IEEE Computer Society (2013)
2. André, É., Klai, K., Ochi, H., Petrucci, L.: A counterexample-based incremental and modular verification approach. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 283–302. Springer, Heidelberg (2012)
3. Byg, J., Jørgensen, K.Y., Srba, J.: TAPAAL: Editor, simulator and verifier of timed-arc Petri nets. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 84–89. Springer, Heidelberg (2009)
4. Haddad, S., Ilié, J.-M., Klai, K.: Design and evaluation of a symbolic and abstraction-based model checker. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 196–210. Springer, Heidelberg (2004)
5. Heiner, M., Richter, R., Schwarick, M.: Snoopy: A tool to design and animate/simulate graph-based formalisms. In: SimuTools, vol. 15 (2008)
6. Kordon, F., Linard, A., Beccuti, M., Buchs, D., Fronc, L., Hillah, L.-M., Hulin-Hubard, F., Legond-Aubry, F., Lohmann, N., Marechal, A., Paviot-Adet, E., Pommereau, F., Rodríguez, C., Rohr, C., Thierry-Mieg, Y., Wimmel, H., Wolf, K.: Model checking contest @ Petri nets, report on the 2013 edition. CoRR, abs/1309.2485 (2013)
7. Kozura, V.E., Nepomniaschy, V.A., Novikov, R.M.: Verification of distributed systems modelled by high-level Petri nets. In: PARELEC, pp. 61–66 (2002)
8. Lakos, C., Petrucci, L.: Modular state spaces for prioritised Petri nets. In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 136–156. Springer, Heidelberg (2011)
9. Le, D.-T.: PeCAn Web page (2014), `http://cse.hcmut.edu.vn/~save/project/pn-ver/start`
10. Schmidt, K.: Distributed verification with LoLA. Fund. Inf. 54(2-3), 253–262 (2003)
11. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
12. Westergaard, M.: CPN Tools 4: Multi-formalism and extensibility. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 400–409. Springer, Heidelberg (2013)