



Petri Net Model Checking with LoLA 2

Karsten Wolf^(✉)

Institut für Informatik, Universität Rostock, Rostock, Germany
karsten.wolf@uni-rostock.de

Abstract. LoLA 2 offers a suite of algorithms for verifying place/transition Petri nets. It combines structural with state space methods and general purpose with Petri net-specific techniques. The methods are easily accessible to people with little knowledge of Petri nets since there is a uniform query language based on temporal logic, and the tool takes care of sound application of its methods. Unlike its predecessor LoLA 1, LoLA 2 is based on a strict modularisation and integration of various standard tools. A careful software engineering approach has been used for coding. Through its code quality and its frequent comparison to other tools in the yearly model checking contests, LoLA 2 has become one of the most reliable verification tools for distributed systems.

1 Introduction

Work on LoLA started in 1997. Originally, the intention was to have just enough code for experimental validation of new state space reduction methods [20, 32–34]. This code collection was presented as LoLA 1.0 in [35].

LoLA has been applied in various case studies stemming from different domains. It proved to be useful for finding hazards in asynchronous circuits [37], for validating and comparing Petri net semantics of web service modeling languages [25], for analysing interacting web services [23, 24], for ontology-based service composition [28], in the automotive area [27], and for self-adaptive systems [10].

Several researchers used LoLA as a reference representing the state of the art in Petri net verification. They compared the performance of LoLA with their own, domain specific tool to justify their algorithms. Examples for this approach include the verification of parameterised Boolean programs [16], verification of multiprocessor code [2], or new ideas for CTL model checking [6].

Several tools integrated LoLA or offer an export to the LoLA input format. This way, the technology implemented in LoLA is available in their particular frameworks. An incomplete list includes tools for design, analysis and simulation such as Snoopy [13] from Cottbus and Renew [22] from Hamburg, packages that try to integrate analysis and synthesis of nets such as APT [1] from Oldenburg and Travis [26] from Hagen, tools in the area of business process management such as ProM [42] or Oryx [3], or for the exploration of biochemical reaction chains such as the Pathalyzer tool [7].

As far as these activities did not directly include members of our group, they typically did not require any consulting from our side. This underpins the ease of use and integration of LoLA. However, as the code grew further, several initial design decisions turned out to be less than optimal. In particular, tool control via a configuration file (with the necessity to re-compile LoLA for every use) was good enough for the initial purpose of LoLA but not very comfortable for being integrated into other tools. Additionally, the poor code structure inhibited the further development via student projects.

Consequently, we decided to completely re-implement LoLA. LoLA 2.x was designed from the very beginning as a tool targeting two use cases. First, we understand LoLA as a community service making state-of-the-art verification technology publicly available thus taken the burden off other scientists to implement Petri net verification methods themselves. Thus, LoLA is designed such that integration into other tools is as simple as possible. Second, LoLA remains our experimental platform for new verification algorithms. This way, we hope that we can keep LoLA competitive in the future.

Code quality was a core issue for LoLA 2. Consequently, we completely re-implemented the tool and did not re-use any piece of code of LoLA 1. First, LoLA 2 got a clear modular structure, benefitting from the experience with LoLA 1. Second, we used a systematic software engineering approach that includes continuous integration management, frequent code reviews and broad discussions on major design decisions. Test case coverage is close to 100% in the core parts of the tool (some code, e.g. reaction to exceptions in the interaction with the operating system, cannot be covered by test cases). In addition, participation in the model checking contest [8] (LoLA is the only tool that participated every year so far) is an excellent platform for adding trust into the produced result. In 2017, results of LoLA were overwhelmingly consistent with the results of other tools, in some categories even 100%. Remaining issues typically concern fresh algorithms and the (in some cases quite involved) translation of PNML and XML input into the input format of LoLA.

In the sequel we shall give an overview on the offered functionality and the concepts for integration. Then we survey the architecture of LoLA 2, briefly surveying the core modules. Finally, we report on use cases and success stories.

2 Installation and Usage

LoLA 2 can be downloaded from www.service-technology.org and is installed using the `automake` procedures. Petri net input can be generated using an ASCII editor, or by modeling tools that offer an export to LoLA. Then LoLA is called on the command line of any UNIX (LINUX or MACOS) terminal where command line options control the property to be verified as well as the technology to be applied. Results appear on the screen or in a file. The package includes a user manual that describes in detail the installation procedure, the file formats, the output, and the options of LoLA 2.

3 Supported Properties

Generally, LoLA 2 can operate in *full*, *none*, or *model checking* mode. In *full* mode, it just computes a (complete or reduced) state space, without investigation of a property. This might be useful for analysing the impact of a verification technique. In *none* mode, it just pre-processes the net, without any state space generation. This way, the user might gather structural information on the net that is calculated in LoLA 2 prior to state space exploration. The most important mode for using LoLA 2, however, is the *model checking* mode. The user formulates a query (on the command line or in a file). The query language permits the specification of a bound expression or a formula in the temporal logic CTL* [5].

A bound expression is a formal sum using places as variables. LoLA 2 computes the maximum value that this expression can get in any reachable marking. A CTL* formula is first pre-processed based on an integrated term rewriting system. Processing aims at

- Removing syntactic sugar (e.g. replacing implication by disjunction);
- Detecting logical tautologies and contradictions in sub-formulas;
- Separating the formula into as many as possible subproblems that are connected via Boolean operations only;
- Pushing the subproblems into any of the fragments LTL or CTL of CTL*.

Looking for tautologies may appear to be odd, but actually we believe that both place/transition nets and their properties are typically the result of a systematic and partly automatic translation process from other kinds of specification. The translation procedures are not necessarily optimised for getting rid of tautologies.

For the resulting Boolean combination, each subproblem is then categorised into one of the following classes:

- Initial satisfaction (can be decided just by inspecting the initial marking);
- Reachability of deadlocks or deadlock freedom;
- Reachability or invariance of a state predicate;
- TSCC based property ($AG EF \phi$, $EF AG \phi$, $AG EF AG \phi$, $EF AG EF \phi$)
- LTL property;
- CTL property;
- CTL* property.

If a property falls into the CTL* category, LoLA cannot verify it and terminates with result *unknown*. For each category, LoLA offers specific verification techniques and specific variants of general techniques. For all classes, state space exploration is available with a category-specific version of the stubborn set method (deadlock: [40], reachability: [32], LTL: [41], CTL: [12]) and a property preserving version of the symmetry method [33,34] being available. For the two reachability categories, we further offer the sweep-line method [19] (with automatic calculation of the progress measure [21]), a random search [31], and

specific structural techniques based on Commoner's theorem [29] and the state equation [44]. For TSCC based properties, a specific search algorithm to find the terminal strongly connected components (TSCC) of the net is used. This category contains Petri net standard properties such as liveness and reversibility.

The atomic propositions in LoLA 2 comprise place based properties such as $p_1 + 2 \cdot p_2 \geq 13$, transition based properties such as $\text{FIREABLE}(t_3)$, and the global properties INITIAL and DEADLOCK. For being able to capture boundedness, we introduced a constant ∞ representing ∞ . So, unboundedness of place p_7 can be specified as $EF\ p_7 \geq \infty$. For such properties, LoLA can construct the coverability graph instead of the reachability graph.

The set of properties that can be specified and analysed in LoLA 2 is significantly larger than in LoLA 1. LoLA 1 did not support formal sums of places, nor DEADLOCK, nor INITIAL, nor the boundedness properties. Categorisation was left to the user thus requiring more knowledge on Petri nets.

If more than one verification technique is available, LoLA 2 runs a portfolio approach. That is, the user can choose to run several methods sequentially or in parallel. The first algorithm that returns a value different from *unknown* determines result and run-time.

4 Integrating LoLA 2

As its predecessor, LoLA 2 is purely command-line oriented. It can process inputs from the UNIX standard input stream and produce results on the standard output. Alternatively, appropriate files can be used. Petri nets are specified in a language that is machine readable and permits, (to our own taste better than PNML [9]) the manual generation of LoLA input for small models. Translation from PNML to the LoLA input is available as a helper tool that is shipped with the LoLA 2 distribution. Results are presented in human readable form on the terminal (together with data collected during computation and status information). Additionally or alternatively, output can be organised according to the Javascript Object Notation (JSON) format which is very convenient for further computer aided post-processing. For supporting the execution of LoLA 2 on remote servers, the output can be broadcasted via the UDP protocol. A listener tool that is part of the distribution can then catch the broadcasted messages and display them. It can also send messages to LoLA 2 forcing it to terminate. With all these features, LoLA 2 supports being run in complex scripts, or being remotely called from other tools.

5 Architecture of LoLA 2

LoLA 2 is strictly modularised thus making it easy to locate the right place for adding code. In the sequel, we shall briefly survey the most important modules.

Parsing. Using the `bison` and `flex` standard tools, input files are transformed into a syntax tree. We use the term processor `kimwitu++` for post-processing

the syntax tree. Kimwitu is able apply rewriting rules and to systematically traverse the tree, for instance for final generation of the internal data structures. We experienced that managing the rewriting and traversing rules of Kimwitu is much more convenient, less error-prone, and easier extensible than the manual implementations we used in LoLA 1.

Net. Places, transitions, and arcs used to be objects in LoLA 1. In LoLA 2, places and transitions are just indices. Information on a place or transition is found under that index in big arrays. This way, retrieving information on the actual Petri net boils down to the traversal of an array rather than traversal of a linked list. This leads to a more “cache friendly” access to the net. Additionally, on a 64 bit architecture, we may still work with 32 bit numbers for representing the net while pointers and object references would consume 64 bits each.

Preprocessing. The data computed here help us to speed up subsequent state space exploration. We explicitly store, for each transition, arrays containing the pre-set and post-set, respectively. This way, we can implement enabledness check as well as transition occurrence very efficiently. We further store, for every transition t , the set of conflicting transitions $(\bullet t)\bullet$ which is frequently needed in stubborn set calculations. Also for stubborn set calculations, we compute the set of conflict clusters using Tarjan’s union/find algorithm [39]. For a net $[P, T, F]$ with set of places P , set of transitions T , and set of arcs $F \subseteq (P \times T) \cup (T \times P)$, a conflict cluster is a class of the partition generated by the symmetric, reflexive, and transitive closure of $F \cap (P \times T)$. Last but not least, we gather information on place and transition invariants to be used for saving memory.

Formula. The formula is internally stored as a tree. The module contains procedures for evaluating and updating state predicates (the temporal parts of the formula are evaluated during the actual state space exploration). Evaluation determines the value in the initial state. Updating computes the impact of a fired transition t occurrence to the formula value. Here, we exploit locality (only subformulas related to t , $\bullet t$, and $t\bullet$ are re-evaluated) and linearity (for instance, a predicate “ $p > 1$ ” is only re-considered if it is false and $p \in t\bullet$, or it is true and $p \in \bullet t$). Necessary dependencies between formula and net structure are calculated during pre-processing. This way, we again save a lot of run-time (given that we need to update a formula millions of times during state space exploration). The module further contains procedures for transforming a state predicate into disjunctive normal form. This is a prerequisite for applying the state equation to reachability problems. Since the formula may explode during this construction (something that frequently occurred in the model checking contests), we have taken this transformation out of the Kimwitu term processor. Additionally, we have implemented an abstract interpretation approach to the sub-formulas. Using that approach, we are able to detect duplicate formulas and can thus alleviate the formula explosion during normalisation.

Planning. This component gathers the information from the categorisation of the property and the command line options. It determines the verification workflow and configures several modules (exploration, firelist, encoding, store) such

that the property under investigation is preserved. For a reachability property, the workflow may consist of a sequential or parallel execution of state space exploration, random walk exploration, and structural verification (siphon/trap property or state equation). For parallel execution of several methods, we use threads. The siphon/trap property is coded as a Boolean formula and shipped to the Minisat SAT checker [4]. The state equation is explored through a call to the tool Sara that extends the evaluation of the state equation with an abstraction refinement approach [44]. The planning component also schedules the verification of more than one property. This may happen if the user specifies more than one property in the command line, or if the property is a Boolean combination of sub-problems which we evaluate separately. As every sub-problem may require state space exploration, we do not support parallel execution in this case as we want to avoid competition for available memory. For supporting more than one state space exploration, we had to solve a severe problem. Since a state space exploration may generate millions of states, we would need to release millions of data objects which may consume a measurable amount of run-time. To avoid this, we clone the process that runs LoLA using the UNIX `fork` command which generates a copy of the process and its whole memory image. Thus, the cloned child process already has all the parsed and pre-processed data and can proceed as if it would execute the first state space exploration. In the end, it communicates its result to the parent process and terminates. Using this mechanism, switching from one subproblem to another takes virtually no time.

Exploration. This is actually a collection of different state space exploration methods. These include

- Simple depth first search for deadlock and reachability properties;
- Coverability graph generation [17] for boundedness properties;
- The sweep-line method [19], including the automated calculation of a progress measure [21];
- A random walk algorithm for deadlock and reachability properties;
- An LTL model checker based on [11] that computes the product system of the reachability graph and a Büchi automaton that is generated from the formula using the `ltl2ba` tool [30];
- A CTL model checker based on [43];
- A depth first search algorithm for TSCC properties that uses a simplification of Tarjan’s algorithm for finding the strongly connected components [38];
- A depth first search algorithm for computing bounds.

The actual algorithm is selected by the planning component (see above). Explorations are parameterised concerning their firelist generation, the particular property to be verified, and the way states are encoded and stored (see below). All algorithms implement the on-the-fly principle. That is, they terminate as soon as the result of the verification is determined. For positive queries (target state turns out to be reachable), the on-the-fly approach is crucial for the excellent performance of explicit state space methods [45].

Fire List Generation. This task has been organised as a separate module as it implements the essence of the stubborn set method. From a base class that implements brute force exploration (all enabled transitions form the fire list), we derive classes for stubborn set methods preserving deadlock, reachability, TSCC, boundedness, bounds, LTL, and CTL. Encapsulation in an own module helps us to keep the code for the actual search algorithms reasonably small.

Encoding. The exploration modules basically work on an integer vector representing the current marking of the search. Before actually storing such a vector, we transform it into a bit vector, aiming at less memory consumption. Our compression techniques include

- No compression at all: 32 bits are used per marking and place (useful for unbounded nets);
- Bit compression: Based on place bounds given in the model, as many bits as necessary for representing the numbers between 0 and the place bound are used (useful for bounded nets with bounds that are known by construction);
- Huffman encoding [14] (new in LoLA 2): this is suitable for nets with no knowledge of token bounds;
- Place invariant compression (combinable with the other methods): we exempt places from being stored if their value can be computed from the remaining places using a place invariant [36].

As a specific way of encoding, the calculation of canonical representatives of the symmetry method [15, 34] is placed here. It is implemented as an encoder that can be parameterised with any other encoder. It canonises a marking and then encodes it using the other encoder. This way, the symmetry method is completely encapsulated for the remaining state space exploration.

Storing. The store maintains information about the already seen states. It is crucial for termination but also for memory consumption and run-time. According to our own profiling, about 90% of the run time of a state exploration is consumed for searching and inserting states. LoLA supports prefix trees and Bloom filters (new in LoLA 2). A Bloom filter just records hash values of visited markings, so we may miss states thus producing an under approximation of the state space. This is taken care of in the result presentation. Instead of *not reachable*, we would return *unknown* while the result *reachable* is preserved by Bloom filtering. We can reduce the probability of hash collision by adding more hash tables with independent hash functions. The user may choose the number of hash tables. We can further calculate the likelihood of remaining hash collisions.

Symmetry. While the *application* of symmetries is integrated into encoding, their calculation remains a separate task. It is executed prior to state space exploration but after starting alternative parallel verification technique (so they do not need to wait for termination of the sometimes lengthy symmetry calculation). Symmetries are calculated based on graph automorphisms. This graph is composed of the Petri net under investigation and the property (the formula tree). The two graphs are linked at the atomic propositions of the property.

This way, symmetries are computed such that the given property is preserved. This general approach is new in LoLA 2. In LoLA 1, the symmetry method could only be applied to global properties that were known not to break symmetry at all. Although LoLA 2 just computes a polynomial size generating set of the graph automorphism group [18], number of generators and run-time may be prohibitive. We counter this problem by the opportunity to compute symmetries in parallel (exploiting the multicore nature of today's computers) and by user definable bounds for run-time and number of generators. We then continue with a subgroup of the symmetry group of the net.

Siphon/Trap Property. If every siphon contains a marked trap, a Petri net (under mild restrictions concerning the arc multiplicities) cannot have deadlocks. If the given property asks for reachability of deadlocks, evaluation of this siphon/trap property permits early abortion of an otherwise time-consuming state space exploration. We evaluate the siphon/trap property by coding it as a Boolean formula [29] which we ship to the Minisat SAT solver [4]. This approach appears to outperform by orders of magnitude other known approaches for evaluating the siphon/trap property. In the model checking contest, this approach is responsible for about 30% of the negative answers (no target state reachable) that LoLA produces to deadlock queries. This takes quite some burden off state space exploration since the on-the-fly principle does not work in the negative case, so negative queries consume much more memory and time.

State Equation. We transform the reachability problem to a list of convex sub-problems (only conjunctions of place-based atomic propositions) and ship this list to the Sara tool [44]. Sara generates a linear program for a convex problem that is based on the state equation and uses the `lp_solve` tool. It checks whether the resulting firing count vector can be arranged to an executable firing sequence. If so, it has solved the original problem. If not, it modifies the linear program based on missing tokens that have been detected in the fireability check. If no modification yields a result, it has a negative answer to the original problem. As the siphon/trap property, the state equation approach takes a lot of burden off state space verification, this time for reachability queries other than deadlock checks. Sara is even more supportive in the model checking contest than the siphon/trap check.

For plain reachability problems, the offered methods perfectly complement each other. While state space exploration performs excellent on positive queries (target state reachable), the structural methods have their merit especially (but in case of Sara not limited to) negative queries. Altogether, the portfolio approach of LoLA 2 yields excellent performance. For more complex properties, it is the rewriting and categorisation that propels the performance of LoLA 2. In the model checking contest, about 15% of CTL properties can be characterised as reachability problems and solved using the powerful portfolio described above. Other CTL problems could be categorised as LTL problems, so the more efficient LTL model checker would be available (recently, LoLA solved more than 90% of the LTL queries but only about 70% of the CTL queries). We are not using this opportunity in the model checking contest since there are subtle semantic

deviances in case of deadlocks between LTL and CTL, regarding the semantics fixed for the contest.

With already a few years experience with LoLA 2, we experienced no major problems concerning its modular structure. The most convincing observation is the fact, that all major state space reduction methods have their natural place: symmetries in the encoder, stubborn sets in firelist generation (only the *ignorance problem* [41] must be taken care of in the exploration), Bloom filtering in the store, the sweep-line method and coverability graph generation in the exploration, and the structural methods as part of our portfolio. Several student project showed that the code base of LoLA 2 is much better extensible than LoLA 1. We believe that this fact contributes to the stability of the tool.

6 Conclusion

The title *Petri net model checking* for this article was chosen with two thoughts in mind. First, LoLA 2 reads place/transition nets and is thus a model checker for Petri nets. Second, much of the performance of LoLA comes from Petri net theory and from exploiting the defining features of Petri nets, monotonicity, linearity, and locality. The monotonicity of the firing rule is used for optimising many basic routines, and enables coverability graphs and the siphon/trap property. Linearity, that is the view of Petri nets being vector addition systems, helps for state compression as well as the state equation approach. Locality is useful for the stubborn set method which is the most powerful state space reduction method in LoLA 2. Consequently, the performance of LoLA 2 can hardly be transferred to other modelling formalisms.

LoLA is focussed on verification technology. Instead of offering our own graphical user interface, we designed LoLA for easy integration into other frameworks. These design goals were confirmed in several case studies and actual integration examples. In the future, we shall continue to work on powerful verification technology. Additionally, we shall include the last remaining features of LoLA 1 that have not yet made it into LoLA 2 (such as printing the state space, or searching home states).

References

1. Best, E., Schlachter, U.: Analysis of Petri nets and transition systems. In: Proceedings ICE. EPTCS, vol. 189, pp. 53–67 (2015)
2. Das, D., Chakrabarti, P.P., Kumar, R.: Functional verification of task partitioning for multiprocessor embedded systems. *ACM Trans. Des. Autom. Electron. Syst.* **12**(4), 44 (2007)
3. Decker, G., Overdick, H., Weske, M.: Oryx – an open modeling platform for the BPM community. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) *BPM 2008*. LNCS, vol. 5240, pp. 382–385. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85758-7_29

4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
5. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.* **2**(3), 241–266 (1982)
6. Dalsgaard, A.E., et al.: Extended dependency graphs and efficient distributed fixed-point computation. In: van der Aalst, W., Best, E. (eds.) PETRI NETS 2017. LNCS, vol. 10258, pp. 139–158. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57861-3_10
7. Dill, D.L., Knapp, M.A., Gage, P., Talcott, C., Laderoute, K., Lincoln, P.: The pathalyzer: a tool for analysis of signal transduction pathways. In: Eskin, E., Ideker, T., Raphael, B., Workman, C. (eds.) RRG/RSB-2005. LNCS, vol. 4023, pp. 11–22. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-48540-7_2
8. Kordon, F., et al.: Homepage of the Model Checking Contest, June 2017. <http://mcc.lip6.fr/>
9. Billington, J., et al.: The Petri net markup language: concepts, technology, and tools. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 483–505. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44919-1_31
10. Cardozo, N., et al.: Modeling and analyzing self-adaptive systems with context Petri nets. In: Proceedings of the TASE, pp. 191–198. IEEE (2013)
11. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan’s algorithm. *Theoret. Comput. Sci.* **345**(1), 60–82 (2005)
12. Gerth, R., Kuiper, R., Peled, D.A., Penczek, W.: A partial order approach to branching time logic model checking. In: Proceedings of the International Symposium on Theory of Computing and Systems, ISTCS 1995, Tel Aviv, Israel, 4–6 January 1995, pp. 130–139. IEEE Computer Society (1995)
13. Heiner, M., Richter, R., Schwarick, M.: Snoopy - a tool to design and animate/simulate graph-based formalisms. In: Proceedings of the PNTAP (2008)
14. Huffman, D.A.: A method for the construction of minimum-redundancy codes. *Proc. IRE* **40**, 1098–1101 (1952)
15. Junttila, T.A.: Computational complexity of the place/transition-net symmetry reduction method. *J. UCS* **7**(4), 307–326 (2001)
16. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_55
17. Karp, R.M., Miller, R.E.: Parallel program schemata. *J. Comput. Syst. Sci.* **3**(2), 147–195 (1969)
18. Knuth, D.E.: Efficient representation of perm groups. *Combinatorica* **11**(1), 33–43 (1991)
19. Kristensen, L.M., Mailund, T.: A generalised sweep-line method for safety properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45614-7_31
20. Kristensen, L.M., Schmidt, K., Valmari, A.: Question-guided stubborn set methods for state properties. *Form. Methods Syst. Des.* **29**(3), 215–251 (2006)
21. Schmidt, K.: Automated generation of a progress measure for the sweep-line method. *STTT* **8**(3), 195–203 (2006)
22. Kummer, O., Wienberg, F.: Renew - the reference net workshop. In: Petri Net Newsletter, pp. 12–16 (2000)

23. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: verification and participant synthesis. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79230-7_4
24. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. *Data Knowl. Eng.* **64**(1), 38–54 (2008)
25. Lohmann, N., Verbeek, E., Ouyang, C., Stahl, C.: Comparing and evaluating Petri net semantics for BPEL. *IJBPM* **4**(1), 60–73 (2009)
26. Meis, B., Bergenthum, R., Desel, J.: travis - an online tool for the synthesis and analysis of Petri nets with final states. In: van der Aalst, W., Best, E. (eds.) PETRI NETS 2017. LNCS, vol. 10258, pp. 101–111. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57861-3_7
27. Mrasek, R., Mülleand, J., Böhm, K., Becker, M., Allmann, C.: Property specification, process verification, and reporting - a case study with vehicle-commissioning processes. *Inf. Syst.* **56**(C), 326–346 (2016)
28. Niewiadomski, A., Wolf, K.: LoLA as abstract planning engine of PlanICS. In: Proceedings of the PNSEi. CEUR, vol. 1160, pp. 349–350 (2014)
29. Oanea, O., Wimmel, H., Wolf, K.: New algorithms for deciding the Siphon-Trap property. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 267–286. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13675-7_16
30. Oddoux, D., Gastin, P.: LTL 2 BA: fast translation from LTL formulae to Büchi automata. <http://www.lsv.fr/~gastin/ltl2ba/>
31. Schmidt, K.: LoLA wird Pfadfinder. In: Proceedings of the AWP, CEUR Workshop Proceedings, p. 26 (1999)
32. Schmidt, K.: Stubborn sets for standard properties. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 46–65. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48745-X_4
33. Schmidt, K.: How to calculate symmetries of Petri nets. *Acta Inf.* **36**(7), 545–590 (2000)
34. Schmidt, K.: Integrating low level symmetries into reachability analysis. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 315–330. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_22
35. Schmidt, K.: LoLA: a low level analyser. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44988-4_27
36. Schmidt, K.: Using Petri net invariants in state space construction. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 473–488. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_35
37. Stahl, C., Reisig, W., Krstic, M.: Hazard detection in a GALS wrapper: a case study. In: Proceedings of the ACS, pp. 234–243. IEEE (2005)
38. Tarjan, R.E.: Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1**(2), 146–160 (1972)
39. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. *J. ACM* **22**(2), 215–225 (1975)
40. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) ICATPN 1989. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53863-1_36

41. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) ACPN 1996. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-65306-6_21
42. van der Aalst, W.M.P., et al.: ProM: the process mining toolkit. In: Proceedings of the BPMDemos. CEUR, vol. 489 (2009)
43. Vergauwen, B., Lewi, J.: A linear local model checking algorithm for CTL. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 447–461. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_31
44. Wimmel, H., Wolf, K.: Applying CEGAR to the Petri net state equation. *Log. Methods Comput. Sci.* **8**(3) (2012)
45. Wolf, K.: Running LoLA 2.0 in a model checking competition. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 274–285. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_13