Kong: a Tool to Squash Concurrent Places

Nicolas Amat, Louis Chauvet

Kong: a Tool to Squash Concurrent Places

Nicolas Amat[1] and Louis Chauvet[1]

LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France

Abstract. Kong, the Koncurrent places Grinder, is a tool designed to compute the concurrency relation of a Petri net by taking advantage of structural reductions. The specificity of Kong is to rely on a state space abstraction, called polyhedral abstraction in previous works, that involves a combination of structural reductions and linear arithmetic constraints between the marking of places.

Keywords: Petri nets · Abstraction techniques · Reachability problems

1 Introduction

KONG, the *Koncurrent places Grinder*, is a recent formal verification tool for Petri nets that can take advantage of structural reductions to accelerate the verification of reachability properties. We made our code freely available under the GPLv3 license and all the software, scripts and data used in this paper are available on GitHub.

In a nutshell, KONG can compute a reduced Petri net, (N', m'), from an initial one, (N, m), and prove properties about the initial net by exploring only the state space of the reduced one. A difference with previous works on structural reductions [4,15], is that our approach is not tailored to a particular class of properties—such as safety or the absence of deadlocks—but could be applied to more general problems. In this paper, we focus on a particular problem supported by KONG, called the *concurrent places problem*.

The correctness of our tool relies on two main theoretical notions. First, a new state space abstraction method, that we called *polyhedral abstraction* in [1], which involves a combination of structural reductions and linear arithmetic constraints between the marking of places. Second, a new data structure, called *Token Flow Graph* (TFG) in [2], that can be used to compute properties based on a polyhedral abstraction. We give a short overview of these two notions in this paper. Nonetheless, our main objective here is to describe the features implemented in our tool.

The basic operation involved in our approach is to compute reductions of the form $(N, m) \vartriangleright_E (N', m')$ where: N is an initial Petri net (that we want to analyse); N' is a residual net (hopefully simpler than N); and E is a system of linear equations. The goal is to preserve enough information in E so that we can rebuild the reachable markings of N knowing only those of N'. We say in this case that N and N' are E-equivalent. While there are many examples of the benefits of structural reductions when model-checking Petri nets, the use of an equation system (E) for tracing back the effect of reductions is new.

In our approach, the computation of structural reductions is delegated to a separate tool. We mention two possibilities in this paper. First the tool Reduce, which is a new addition to the Tina model-checking toolbox since version 3.7 (https://projects.laas.fr/tina). We also describe, with more details, a new open-source framework called Shrink. This is a highly customizable tool, and also a library, that we hope can be reused and improved in other contexts.

A TFG is a graph-like data structure that can be built from an E-equivalence statement, $(N, m) \rhd_E (N', m')$, and that embodies the structure of the equations occurring in E. Kong can build a TFG from sequences of reductions computed using Shrink or Reduce, and use it to symbolically explore the state space of the initial net.

We describe two applications of TFGs. The main application [2] is to compute the *concurrency relation* of a Petri net; what is also known as the concurrent places problem [8]. The goal is to enumerate all pairs of places that can be marked together in some reachable marking. This problem has practical applications, for instance because of its use for decomposing a safe Petri net into the product of concurrent processes [8,9]. It also provides an interesting example of safety property that nicely extends the notion of *dead places*; meaning places that can never be marked. To illustrate the versatility of our approach, we also describe a new feature, implemented in Kong, for checking the reachability of a given marking. Our method exploits the TFG to compute a reduced, projected marking that needs to be found in the reduced net. This is a direct illustration of the philosophy behind Kong, which is solving complex problems by first reducing a Petri net; then solving the problem in a "lower dimension"; before finally transposing this solution to the initial net.

Outline. The rest of the paper is organized as follows. In Sect. 2, we detail how to install and use Kong. Section 3 describes the architecture of Kong and Shrink. We illustrate the workings of Kong on a concrete example, in Sect. 4. Finally, we validate our tool by discussing the results of experiments performed with nets used in the 2021 edition of the Model Checking Contest (MCC).

2 Commands, Basic Usage and Installation

Kong is an open-source tool made freely available on GitHub (https://github.com/nicolasAmat/Kong). The code repository also provides all the material to reproduce the experiments described in Sect. 5.

Dependencies. Kong is written in Python and requires a version 3.5 or higher. It also requires the `graphviz` Python library in order to output a graphical description of Token Flow Graphs (optional). Scripts and models included in the repository are used for benchmarking and for continuous testing. Kong is intended to be as understandable as possible; the code is heavily documented and we provide many tracing and debugging options that can help understand its inner workings.

We support two different tools to compute polyhedral abstractions, Reduce and Shrink, that both use the same input and output formats. Reduce is a tool developed inside the Tina toolbox [14], since version 3.7. It is currently used by the Tina.Tedd and SMPT model-checkers, that both compete in the Model Checking Contest (MCC) [3,13], albeit on different examinations. Shrink is an open-source alternative, on which we focus in Sect. 3. Kong runs Reduce if the executable is in the current `PATH` environment variable, but automatically switches to Shrink otherwise. It is still possible to enforce the use of Shrink by using the `--shrink` option. It is also possible to directly provide a precomputed result of structural reductions with the option `--reduced-net`.

Concurrent and Dead Places. Kong is a CLI tool organized around subcommands to expose its different features. The tool provides several options that are described in the documentation using `--help`. We give a brief description of some of them in the following sections.

The main subcommands of Kong are `conc` and `dead` for, respectively, computing the concurrent relation and the list of dead places in a net. When computing a concurrency matrix, Kong relies on an external tool to compute the concurrency matrix of the reduced net. This is currently done using cæsar.bdd, part of the CADP toolbox [7,12], which is the state-of-the-art tool for the concurrent places problem [7,12].

Kong takes as inputs ordinary, safe Petri nets defined using either the Petri Net Markup Language (PNML) [11], or the Nest-Unit Petri Net (NUPN) format [9]. (The file format is automatically detected from the file extension.) The use of a NUPN decomposition, which provides information about the concurrent structure of the net, can bring a significant performance improvement. The tool was designed to be fully compatible with Petri net instances used in the MCC. For instance, we can make use of NUPN information added to a PNML model using its tool-specific extension mechanism.

Kong can be executed as a Python script or converted into a standalone executable using `cx_Freeze`. Each subcommand only requires the path to the input Petri net (with a `.pnml` or `.nupn` extension). Hence a typical call to Kong is of the form ‘`./kong.py conc model.pnml`‘. We also provide two main options to limit the exploration performed by cæsar.bdd: `--bdd-timeout` to set a time limit and `--bdd-iterations` to limit the number of iterations. Debugging options are described in Sect. 4.

The concurrency relation of a Petri net, denoted C, is encoded as a symmetric matrix of dimension $|P|$, where $|P|$ is the number of places in the net. We also use the name *concurrency matrix*. We use the notation $\mathrm{C}[p, q] = 1$ when places p, q can be marked together in a reachable state, and 0 otherwise. In some cases, we may need to work with “partial relations”; for example when we impose a time limit. We say that the concurrency matrix is *incomplete* in this case and use the value '·' (a dot) for pairs of places where the relation is undecided.

Our output format for the concurrency matrix is taken from CÆSAR.BDD. We can output our results using a compressed format, based on a run-length encoding (RLE) of the rows of C. For the sake of readability, it is possible to disable this encoding using option `--no-rle`. It is also possible to print the place ordering with option `--place-names`.

A call to `‘kong.py conc‘` delegates the computation of the concurrent relation on the reduced net to the tool CÆSAR.BDD. It can also take as input a precomputed concurrency matrix of the reduced net, using option `--reduced-matrix`. Likewise, the `dead` subcommand provides option `--reduced-vector` if we have a precomputed list of dead places for the reduced net.

Marking Reachability. The `reach` subcommand provides a procedures to check if a given marking is reachable. Like previously, this command relies on an external tool to check if a marking is reachable in the reduced net. To this end, we use SIFT, which is an explicit-state model-checker for Petri nets from the Tina toolbox, that can check reachability properties on the fly.

The tool takes as input a Petri net—not necessarily safe, ordinary or bounded— described either in the PNML or the NET format. (NET is the specification format of the Tina toolbox). The target marking is defined using a simple textual format, as a space-separated list of place identifiers with their multiplicities, of the form `p*k`, where `p` is a place and `k` is a positive integer. By default, places that are not listed contain no tokens. The path to the file describing the target marking is given using option `--marking`.

3 Architecture of KONG

Our tool is basically composed of three modules: `kong.py` the front-end program in charge of parsing command-line options; `pt.py` a Petri net parser; and `tfg.py` the data structure and computational module based on Token Flow Graphs. We illustrate the architecture of KONG in Fig. 1, where we describe the different steps involved during a typical computation. The first step is to reduce the input Petri net, say (N, m), using the SHRINK tool. SHRINK outputs a reduced net (N', m') and a system of linear equations E. We display in Fig. 2 a sequence of structural reductions, with their equations, computed using SHRINK. By construction, the result of this first stage is guaranteed to be a polyhedral abstraction.

Then we build a Token Flow Graph, $[\![E]\!]$, from the set of linear equations in E. The TFG is a Directed Acyclic Graph (DAG), capturing the specific structure of the equations in E, that allows us to reason about the reachable markings by playing a token game on this graph.

At this stage, we must distinguish two possible cases. First, the net could be fully reduced, meaning the resulting net is “empty”; it has no remaining places. In this case, the set of markings of (N, m) is exactly the solutions of the linear system E. Hence the TFG is enough to compute the concurrency matrix using an algorithm that we call *dimensionality reduction*, or to decide if a given marking

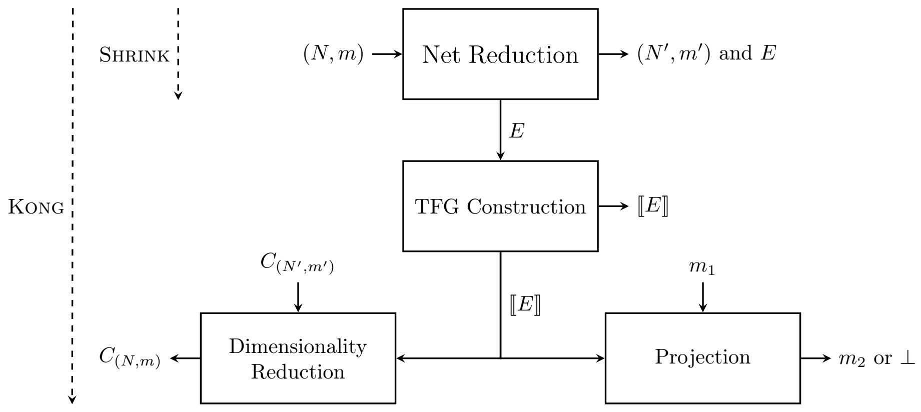

Fig. 1: Kong's architecture.

is reachable. Otherwise, we have a non-trivial reduced net, in which case we need to compute the concurrency matrix of (N', m') or to check the reachability of the *projection* of our marking of interest.

The first module of our pipeline relies on two Rust libraries, based on a common crate called `pnets`, that defines functions for parsing, manipulating and reducing Petri nets. This code is freely available on GitHub (https://github.com/Fomys/pnets), under the MIT license.

Petri Nets Library. The `pnets` library is the core for parsing and manipulating Petri nets. It supports both standard and timed Petri nets. Internally, Petri nets are stored using adjacency lists, ensuring a low memory footprint and fast iterations over connected places and transitions. The toolbox includes two sublibraries for parsing nets: `pnets_pnml` and `pnets_tina`, respectively for the PNML and NET formats.

Structural Reduction Library. The `pnets_shrink` library implements reductions rules described in [5,6]. It implements a large subset of the reductions included in Reduce, such as (definitions refers to the ones in [6]): `T` - Redundant transitions (def. 1), `P` - Redundant places (def. 2), `SCA` - Simple chain agglomeration (def. 5), `SLA` - Simple loop agglomeration (def. 6), and `SSP` - Source-sink pair (def. 10).

Standalone Reduction Tool. Shrink is a standalone program, integrated with Kong, and built with the `pnets` crate. It can be installed using `cargo`, the Rust package manager, by running `‘cargo install pnets_shrink‘`, or built from sources available in the GitHub repository.

Shrink can parse nets defined in the PNML or NET formats, and use the NET format for its output. Use option `-i` to indicate the path to the input net, and `-o` to redirect the reduced net. It is possible to use `-` for replacing paths by the standard input or output. Another option, `--equation`, can be used to print the reduction equations as comments in the output net (lines starting by `#`).

Shrink is quite modular, different options permit to enable subsets of reduction rules from the `pnets_shrink` library. For instance, `--redundant` enables the `T`, `P` and `SSP` rules, and `compact` the `SCA` and `SLA` ones. Furthermore, a loop iteration limit over the net can be set using the option `--max-iter <MAX_ITER>`.

4 Concrete Example

The simplest way to illustrate the usage of Kong is to look at a concrete example. This is also a good opportunity to show the debugging options provided by our tool. Assume (N, m) is the net in top left position in Fig 2.

Net Reduction. Structural reduction is performed iteratively, until no new reductions are possible. We display, Fig. 2, a sequence of four reductions that leads to the result computed with Shrink; the marked net at the bottom-right. Each row is an example of reduction, and its associated equation. First, it is always safe to remove a *redundant place*, e.g. a place with the same pre and post conditions than another one. This is the case with places p_4, p_5. Redundant places can sometimes be found by looking at the structure of the net, but we can use more elaborate methods to find redundant places by solving an integer linear programming problem [16]. After the removal of p_5, we obtain the equation $p_4 = p_5$, and we are left with the residual net at the left part of row 2. In this case, we can use an agglomeration rule, which states that we can fuse places inside a "deterministic sequence" of transitions. For instance to simplify places p_1 and p_2 into a new place, a_{12}. Similar situations, where we can aggregate several places together, can be found by searching patterns in the net. After this step, we find a new opportunity to reduce a redundant place, based on the structural invariant $a_{12} = p_3 + p_4$. We conclude by agglomerating places p_3 and p_4 into a new place, a_{13}.

At the end of these reductions, we obtain the reduced net, (N', m'), with only 3 places instead of 6. We also obtain a system of four linear equations $E \triangleq (p_5 = p_4), (a_{12} = p_1 + p_2), (a_{12} = p_3 + p_4), (a_{13} = p_3 + p_4)$.

Kong provides an option, `--save-reduced-net`, to save the reduced net into a specific file. Additionally, we can print the reduction equations with the option `--show-equations`.

TFG Construction. Kong can build the TFG associated with the linear system E; see Fig. 3. It is possible to output a graphical version of the TFG using option `--draw-graph`. The TFG is a DAG where the vertices are the places of the input and reduced net, in addition to the free variables from E. The set of roots (nodes with no predecessor) is exactly the set of places of the reduced net N'. Arcs in the TFG are used to depict the relation induced by equations in E.

A TFG includes two different kinds of arcs. Arcs for *redundancy equations*, $q \rightarrow\!\bullet\, p$, to represent equations of the form $p = q$ (or $p = q + r + \ldots$), corresponding

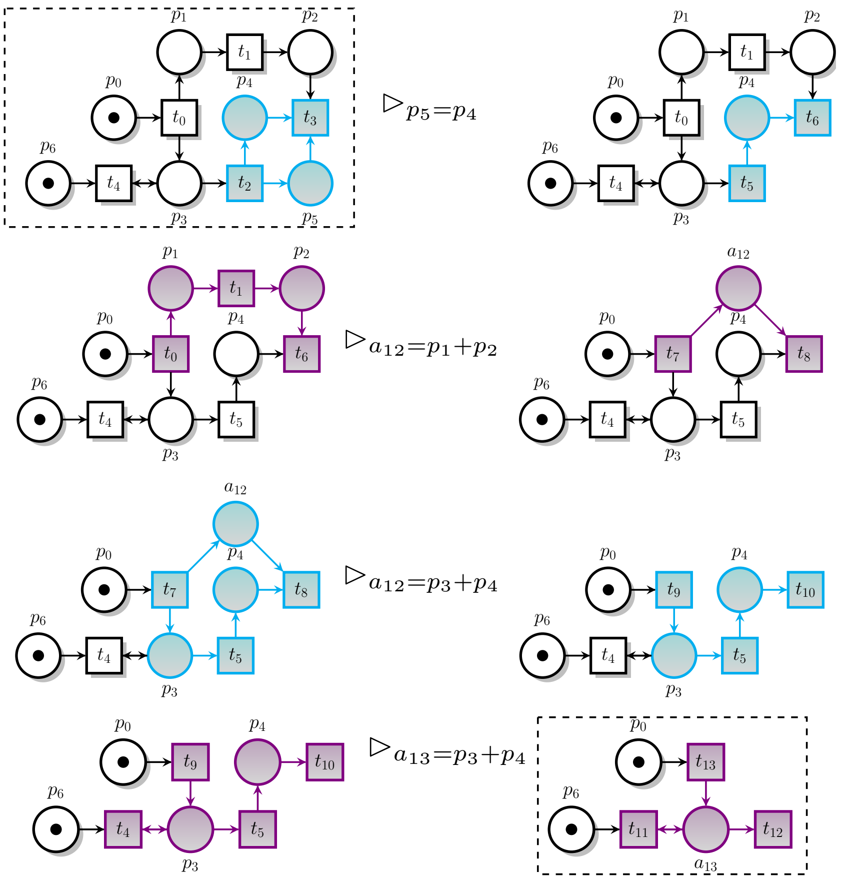
$\rhd p_5 = p_4$
$\rhd a_{12} = p_1 + p_2$
$\rhd a_{12} = p_3 + p_4$
$\rhd a_{13} = p_3 + p_4$

Fig. 2: Example of sequence of four reductions leading from the net N to N'.

to redundant places. In this case, we say that place p is *removed* by arc $q \to\!\!\bullet\, p$, because the marking of q may influence the marking of p, but not necessarily the other way round.

The second kind of arcs, $a \circ\!\!\rightarrow p$, is for *agglomeration equations*. It represents equations of the form $a = p + q$, generated when we agglomerate several places into a new one. In this case, we expect that if we can reach a marking with k tokens in a, then we can certainly reach a marking with k_1 tokens in p and k_2 tokens in q when $k = k_1 + k_2$. Hence information flows in reverse order compared to the case of redundancy equations. This is why, in this case, we say that places p and q are removed. We also say that node a is *inserted*; it does not appear in N but may appear as a new place in N'. We can have more than two places in an agglomeration.

We can use the TFG to reason about the reachable markings of a net by playing a "token game" on this DAG. Basically, we can put tokens on the roots

```
# generated equations
# R |- p5 = p4
# A |- a12 = p1 + p2
# R |- a12 = p3 + p4
# A |- a13 = p3 + p4
```

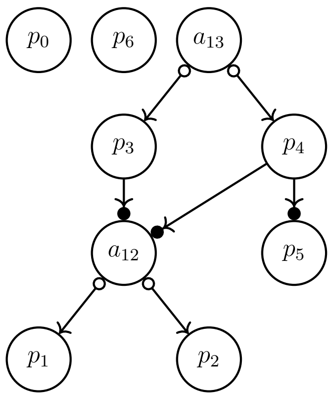

Fig. 3: Equations generated from net N, in Fig.2, and associated TFG $[\![E]\!]$.

of the graph (given a marking of N') then propagate them downwards while respecting the constraints dictated by the $\rightarrow\!\bullet$ and $\circ\!\!\rightarrow$ arcs. The result observed on the $\circ\!\!\rightarrow$-leaf nodes (the places of N) is guaranteed to be reachable in (N, m).

Concurrent Places Algorithm. With subcommand `conc`, the final stage is to compute the concurrency matrix of the input net, $C_{(N,m)}$, from the one of the reduced net, $C_{(N',m')}$. Currently, Kong uses CÆSAR.BDD to compute $C_{(N',m')}$. But we could adapt Kong to use any other tool that can compute the concurrency relation, such as [17]. It is possible to output this matrix with option `--show-reduced-matrix` (resp. `--show-reduced-vector` if we use subcommand `dead`).

We can give an intuition for our *Dimensionality Reduction* algorithm using our example. For instance, we have that place a_{13}, in the reduced net N' of Fig. 2, is non-dead (because we can fire t_9). As a consequence, all the successors nodes of a_{13} in the TFG (that are also places in N) must also be non-dead, meaning $C[p_i, p_i] = 1$ for all i in 1..5. Also, we can deduce that p_4 is concurrent to p_5 (meaning $C[p_4, p_5] = 1$), because of the redundancy $p_5 = p_4$, and p_1, p_2 are concurrent to p_3, p_4, p_5. A detailed description of our algorithm can be found in [2].

Marking Reachability Decision. With subcommand `reach`, the final step is to project the marking of interest into a new marking defined on the reduced net, and to check its reachability in the reduced net (N', m').

We illustrate this procedure by taking two concrete examples on the marked net N given in Fig. 2 (first row, left). Assume we want to check if marking $m_1 \triangleq (p_0 = 0\,, p_1 = 1\,, p_2 = 1\,, p_3 = 1\,, p_4 = 1\,, p_5 = 1\,, p_6 = 0)$ is reachable in (N, m). This marking can be mapped to a unique marking of N', namely $m_2 \triangleq (p_0 = 0\,, p_6 = 0\,, a_{13} = 2)$. (Use option `--show-projected-marking` to output this marking.) Deciding if marking m_1 is reachable in (N, m) is equivalent to deciding if $m_2 \triangleq (p_0 = 0\,, a_2 = 2\,, p_6 = 0)$ is reachable in (N', m') (which it is not). Observe that m_1 would be reachable if the initial marking m was $(p_0 = 2, p_6 = 1)$ and the other places empty.

The "marking projection" algorithm can also directly return with a contradiction ($\perp$), meaning that the target marking cannot be reached. Assume we want to check the reachability of a marking m'_1 such that $m'_1(p_4) = 2$ and $m'_1(p_1) = m'_1(p_2) = 0$. It is not possible to project this marking into N' while respecting the constraint given in the TFG. In this case, we directly obtain that m'_1 is not reachable in (N, m).

5 Performance

We used the database of models provided by the Model Checking Contest [3,13] to study the performances of Kong. For the computation of concurrent matrices, among the 562 safe and ordinary instances used in the MCC'2021, we kept only the ones with reduction opportunities; which amount to 424 nets in total. And we selected 426 instances (among 1 411) to evaluate the marking reachability procedure, for which we generated 5 queries that are markings found using a "random walk" on the state space of the net. We used Reduce to compute net reductions, we computed the concurrency matrices on the reduced net with cæsar.bdd, version 3.6, part of CADP version 2022-b "Kista", published in February 2022 and and used Sift to check the reachability of the projected marking.

To understand the impact of reductions on the computation time, we compare cæsar.bdd and Sift alone, on the initial net, and Kong + Reduce + cæsar.bdd or Sift on the reduced net. We display our results in the charts of Fig. 4, which gives the number of feasible instances, for each tool, when we change the timeout value. (To reproduce the experiments follow the instructions from the README file in the `benchmark/` directory of the repository.)

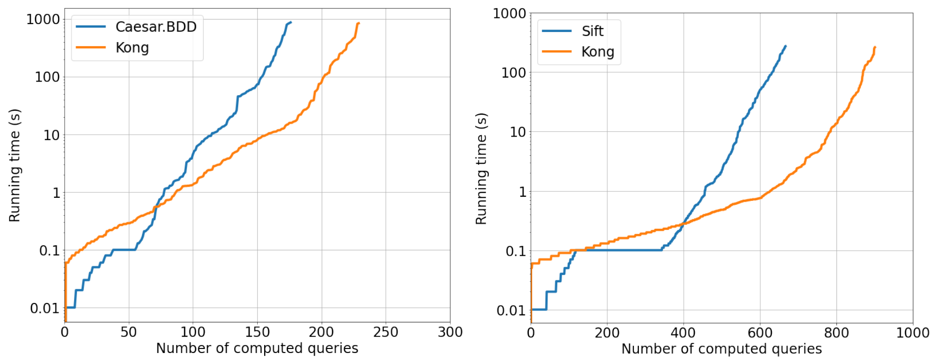

Fig. 4: Minimal timeout to compute a given number of instances: (left) concurrency matrices, (right) reachable markings.

We observe that net reductions have a clear impact on the speed-up and that we can compute more instances with reductions than without: 229 matrices for KONG against 176 for CÆSAR.BDD alone, with a timeout of 15 min. The same observations holds for the reachability procedure: 901 queries solved for KONG against 667 for SIFT alone, with a timeout of 5 min. Furthermore concerning the tool REDUCE, we obtained on safe instances a mean reduction ratio—that is the quotient between how many places can be removed and the number of places in the initial net—of 40% (median of 26%), computed in an average time of 0.7 s (median of 0.2 s).

6 Future Work

Both KONG and SHRINK are destined to evolve. For instance, we want to experiment with more challenging problems using KONG and the TFG data-structure. We are particularly interested in answering reachability queries expressed using a boolean combination of constraints over place markings. Another interesting problem would be to support the verification of Generalized Mutual Exclusion Constraints, like in [10], that requires checking invariants involving a weighted sums over the marking of places, of the form $\sum_{p \in P} w_p.m(p) \leqslant k$, with $w_1, \dots, w_n, k$ constants in $\mathbb{Z}$.

We also want to explore new reduction rules using our polyhedral abstraction framework SHRINK. We already developed new reduction rules for specific models from the MCC, such as `Election2020` and `ViralEpidemic`, and plan to look at more specific examples of reduction rules.

To conclude, we are convinced that there is still a lot of work to be done to compute polyhedral abstractions, and to apply them on useful and complex problems.

Acknowledgements. We would like to thank Bernard Berthomieu and Silvano Dal Zilio for their help on the development of our reduction library, and Pierre Bouvier for his remarks that helped improve the quality of KONG.

References

1. Amat, N., Berthomieu, B., Dal Zilio, S.: On the combination of polyhedral abstraction and SMT-based model checking for Petri nets. In: International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets). LNCS, vol. 12734. Springer (2021). https://doi.org/10.1007/978-3-030-76983-3_9
2. Amat, N., Dal Zilio, S., Le Botlan, D.: Accelerating the Computation of Dead and Concurrent Places using Reductions. In: 27th International SPIN Symposium on Model Checking of Software. LNCS, vol. 12864. Springer, Aarhus, Denmark (2021). https://doi.org/10.1007/978-3-030-84629-9_3
3. Amparore, E., Berthomieu, B., Ciardo, G., Dal Zilio, S., Gallà, F., Hillah, L.M., Hulin-Hubard, F., Jensen, P.G., Jezequel, L., Kordon, F., Le Botlan, D., Liebke, T., Meijer, J., Miner, A., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y., van Dijk, T., Wolf, K.: Presentation of the 9th edition of the model checking contest. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 11429. Springer (2019). https://doi.org/10.1007/978-3-662-58381-4_9
4. Berthelot, G.: Transformations and Decompositions of Nets. In: Petri Nets: Central Models and their Properties. LNCS, vol. 254. Springer (1987). https://doi.org/10.1007/978-3-540-47919-2_13
5. Berthomieu, B., Le Botlan, D., Dal Zilio, S.: Petri net reductions for counting markings. In: International Symposium on Model Checking Software (SPIN). LNCS, vol. 10869. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_4
6. Berthomieu, B., Le Botlan, D., Dal Zilio, S.: Counting Petri net markings from reduction equations. International Journal on Software Tools for Technology Transfer **22** (2019). https://doi.org/10.1007/s10009-019-00519-1
7. Bouvier, P., Garavel, H.: Efficient algorithms for three reachability problems in safe Petri nets. In: International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets). LNCS, vol. 12734. Springer (2021). https://doi.org/10.1007/978-3-030-76983-3_17
8. Bouvier, P., Garavel, H., Ponce-de León, H.: Automatic decomposition of Petri nets into automata networks – a synthetic account. In: International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets). LNCS, vol. 12152. Springer (2020). https://doi.org/10.1007/978-3-030-51831-8_1
9. Garavel, H.: Nested-unit Petri nets. Journal of Logical and Algebraic Methods in Programming **104** (2019). https://doi.org/10.1016/j.jlamp.2018.11.005
10. Giua, A., DiCesare, F., Silva, M.: Generalized mutual exclusion contraints on nets with uncontrollable transitions. In: IEEE International Conference on Systems, Man, and Cybernetics. IEEE (1992). https://doi.org/10.1109/ICSMC.1992.271666
11. Hillah, L.M., Kordon, F., Petrucci, L., Treves, N.: PNML framework: an extendable reference implementation of the Petri Net Markup Language. In: International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets). LNCS, Springer (2010). https://doi.org/10.1007/978-3-642-13675-7_20
12. INRIA: CADP. https://cadp.inria.fr/ (2020)
13. Kordon, F., Bouvier, P., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., , Dal Zilio, S., Jensen, P., He, C., Le Botlan, D., Li, S., , Srba, J., Thierry-Mieg, ., Walner, A., Wolf, K.: Complete Results for the 2020 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php (2021)
14. LAAS-CNRS: Tina Toolbox. http://projects.laas.fr/tina (2020)

15. Murata, T., Koh, J.: Reduction and expansion of live and safe marked graphs. IEEE Transactions on Circuits and Systems **27**(1) (1980). https://doi.org/10.1109/TCS.1980.1084711
16. Silva, M., Terue, E., Colom, J.M.: Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In: Advanced Course on Petri Nets. Springer (1996). https://doi.org/10.1007/3-540-65306-6_19
17. Wiśniewski, R., Wiśniewska, M., Jarnut, M.: C-exact hypergraphs in concurrency and sequentiality analyses of cyber-physical systems specified by safe Petri nets. IEEE Access **7** (2019). https://doi.org/10.1109/ACCESS.2019.2893284