A Toolchain to Compute Concurrent Places of Petri Nets

Nicolas Amat[1 [0000−0002−5969−7346]], Pierre Bouvier[2], and Hubert Garavel[2]

LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP LIG, Grenoble, France

Abstract. The concurrent places of a Petri net are all pairs of places that may simultaneously have a token in some reachable marking. Concurrent places generalize the usual notion of dead places and are particularly useful for decomposing a Petri net into synchronized automata executing in parallel. We present a state-of-the-art toolchain to compute the concurrent places of a Petri net. This is achieved by a rich combination of various techniques, including: state-space exploration using BDDs, structural rules for concurrent places, quadratic over- and under-approximation of reachable markings, and polyhedral abstraction of the state space based on structural reductions and linear arithmetic constraints on markings. We assess the performance of our toolchain on a large collection of 850 nets originating from the 2022 edition of the Model Checking Contest.

Keywords: Petri nets · Nested-unit Petri nets · Model checking · Reachability problems · Concurrency theory · Abstraction techniques · Structural reductions · State-space exploration

1 Introduction

There is a rich corpus of scientific literature on the analysis of concurrent systems, which is a difficult topic, as most algorithms have a high complexity that increases with the size of the systems under study. Besides the usual properties expressing safety and liveness features of concurrent systems, the present article focuses on a less known property that fundamentally characterizes where parallelism is present in such systems. To present this property, a few preliminary definitions are necessary.

1.1 Petri Nets

Petri nets [32,31] are one of the oldest techniques for modelling concurrent systems. In this article, the full generality of Petri nets is not required and we can merely consider nets that are *ordinary* (i.e., all arcs have multiplicity one) and *one-safe* (i.e., all reachable markings contain at most one token per place).

Formally, we define here a Petri net as a 4-tuple (P, T, F, M_0), where P is a finite, non-empty set (the elements of P are called *places*); T is a finite set such that $P \cap T = \varnothing$ (the elements of T are called *transitions*); F is a subset of $P \times T \cup T \times P$ (the elements of F are called *arcs*); M_0 is a subset of P (M_0 is called the *initial marking*). Figure 1 gives an example of a Petri net having 13 places (two of which being initial places), 11 transitions, and 26 arcs.

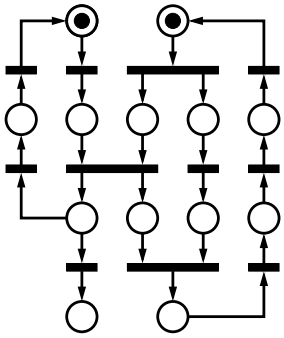

Fig. 1: An ordinary, one-safe Petri Net.

Given a transition $t \in T$, the *pre-set* of t (noted ${}^\bullet t$) and the *post-set* of t (noted $t^\bullet$) are the two sets of places defined as follows: ${}^\bullet t = \{p \in P \mid (p, t) \in F\}$ and $t^\bullet = \{p \in P \mid (t, p) \in F\}$.

A *marking* is a subset of P. A transition t can fire from some marking M_1 to some other marking M_2 (noted $M_1 \xrightarrow{t} M_2$) iff ${}^\bullet t \subseteq M_1$ and $M_2 = (M_1 \setminus {}^\bullet t) \cup t^\bullet$. A marking M is *reachable* from the initial marking M_0 iff $M = M_0$ or there exist $n \geq 1$ transitions $t_1, t_2, ..., t_n$ and $(n-1)$ markings $M_1, M_2, ..., M_{n-1}$ such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ ... \ M_{n-1} \xrightarrow{t_n} M$. To simplify the presentation, we denote by (N, M_0) a marked net, which is a pair composed of a net $N \triangleq (P, T, F)$ and an initial marking M_0.

1.2 Nested-Unit Petri Nets

Nested-Unit Petri Nets (NUPNs, for short) [12,13] are a widespread extension of Petri nets for expressing *locality* and *hierarchy* properties of concurrent systems. The concept of NUPN is not recent (see, e.g., [18]), but it has been adopted by many recent Petri-net analysis tools, which significantly increase their performance by exploiting NUPN information about locality and hierarchy.

Formally, a NUPN is defined as a 8-tuple $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unit})$, where: (P, T, F, M_0) is a Petri net (as defined in Sect. 1.1); U is a finite, non-empty set such that $U \cap T = U \cap P = \varnothing$ (the elements of U are called *units*); u_0 is an element of U (u_0 is called the *root unit*); $\sqsubseteq$ is a binary relation over U such that $(U, \sqsupseteq)$ is a tree with a single root u_0, where $(\forall u_1, u_2 \in U)$ $u_1 \sqsupseteq u_2$ is defined as $u_2 \sqsubseteq u_1$ ($\sqsubseteq$ is thus a reflexive, antisymmetric, and transitive relation that

expresses that a unit is transitively included in another unit, the root unit u_0 being the maximal element for $\sqsubseteq$, i.e., the unit that transitively contains all other units); unit is a function $P \to U$ such that $(\forall u \in U \setminus \{u_0\})\ (\exists p \in P)\ \mathsf{unit}\,(p) = u$ (intuitively, $\mathsf{unit}\,(p) = u$ expresses that unit u directly contains place p). The *height* of a NUPN is the height of its unit tree, not counting the root unit if it contains no place directly (i.e., for each $p \in P$, $\mathsf{unit}\,(p) \neq u_0$). The *width* of a NUPN is the number of leaf units in its unit tree.

The token game for NUPNs is exactly the same as for Petri nets, meaning that the rules for firing transitions and the set of reachable markings are not modified by the introduction of units.

A key property of NUPNs is the notion of *unit safeness* [13, Sect. 3], which generalizes the one-safeness property of Petri nets. Formally, two units u_1 and u_2 are *disjoint* iff $(u_1 \not\sqsubseteq u_2)\ \wedge\ (u_2 \not\sqsubseteq u_1)$, meaning that both units are neither equal nor contained one in the other. A NUPN is *unit-safe* iff each of its reachable markings (including M_0) only contains pairs of places located into disjoint units, meaning that each unit, or two transitively nested units, may not contain two tokens at the same time. This property enables logarithmic reductions in the number of bits or Boolean variables needed to represent reachable markings [13, Sect. 6].

In practice, the unit-related information, namely $(U, u_0, \sqsubseteq, \mathsf{unit})$, is directly obtained when the NUPN is produced from a higher-level model [13, Sect. 4]. For instance, if the NUPN is generated from a process-calculus language such as LOTOS [21] or LNT [16], the unit tree can be deduced from the parallel composition operators present in the source specifications; if the NUPN is generated from a network of automata, the unit tree represents the various automata that execute concurrently; etc.

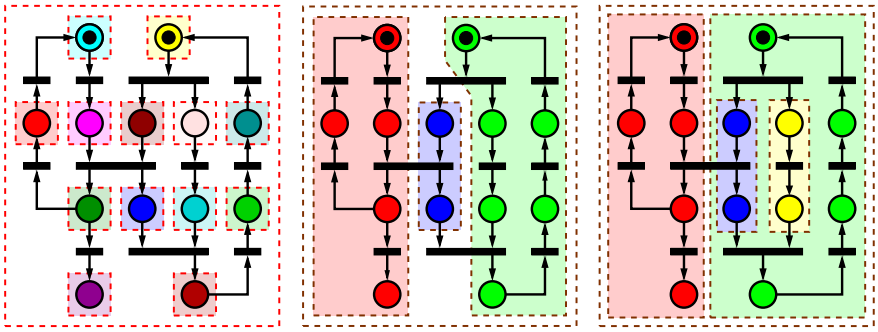

Fig. 2: Petri Net of Fig. 1 decomposed either into a trivial NUPN (left), a flat NUPN (middle) or a hierarchical NUPN (right).

Any unit-safe NUPN can be converted to an ordinary, one-safe Petri net by erasing unit-related information. Reciprocally, any ordinary, one-safe Petri net

can be easily converted to a unit-safe NUPN by putting each of its places into a separate unit and having a root unit u_0 encapsulating all the other units; such a NUPN (whose width is equal to its number of places) is called *trivial* (see Fig. 2 (left)). Unfortunately, this easy transformation brings no gain for state-space exploration. However, an ordinary, one-safe Petri net may have various corresponding unit-safe NUPNs, which may be either *flat* if their height is one (see Fig. 2 (middle)) or *hierarchical* if their height is greater than one (see Fig. 2 (right)). Converting an ordinary, one-safe Petri net into a non-trivial unit-safe NUPN is an involved task (see [11] for insights on the decomposition into flat NUPNs), but may bring significant benefits for further analyses conducted with tools that take advantage of NUPN information.

1.3 Concurrent Places

We now introduce the notion of concurrent places, which is central to the present article. Given a net (Petri net or NUPN), two places p and p' are *concurrent* iff there exists a reachable marking M such that both p and p' have a token in M. This relation is symmetric and quasi-reflexive; it is reflexive iff the net has no dead place (i.e., no place that has no token in any reachable marking) [10, Sect. 2.4].

This relation characterizes those parts of the net that can be simultaneously active. It is mentioned in many publications under various names, such as: *coexistency defined by markings* [23, Sect. 9], *concurrency graph* [24] [35], or *concurrency relation* [29] [33] [28] [27] [17], etc. These definitions slightly differ by minor details, such as the kind of Petri nets considered, or the handling of reflexivity, i.e., whether and when a place is concurrent or not with itself.

In general, this relation is relevant only for one-safe, ordinary Petri nets, since the presence of multiple tokens in the same places often implies that most pairs of places are concurrent [10, Sect. 2.4]. This retrospectively justifies our choice to consider ordinary, one-safe nets, rather than full-fledged P/T nets.

Given a net, the problem of computing all its pairs of concurrent places is PSPACE-complete [10, Sect. 2.5]. This problem is practically useful [14] for, at least, two reasons:

- Most approaches for decomposing a net into a set of concurrent automata or into a NUPN [11] require knowledge about concurrent places.
- The notion of concurrent places nicely generalizes the notion of dead places, since a place is dead iff it is not concurrent with itself. Determining dead places is a relevant problem, equivalent, for Petri nets, to dead-code removal in software engineering. Indeed, many global properties of a net can be changed to true or to false just by adding or removing dead places; also, the memory cost of verification is likely to be increased by the presence of dead places. For instance, the Grafcet specification [20] used in industrial automation prohibits Sequential Function Charts containing "unreachable" branches (i.e., Petri nets with dead places or dead transitions).

1.4 Outline

This article presents a toolchain that efficiently computes the concurrent places of a given net (Petri net or NUPN). Although such computation could be done by reusing some existing Petri-net model checker, this approach would not be efficient, as the number of temporal-logic formulas to be evaluated would be quadratic in the number of places: a more "global" algorithm should be preferred. To this aim, our toolchain integrates various tools implementing a combination of complementary analysis techniques, such as: state-space exploration using Binary Decision Diagrams (BDDs), structural rules for concurrent places, quadratic over- and under-approximation of reachable markings, and polyhedral abstraction of the state space based on structural reductions and linear arithmetic constraints on markings.

The remainder of the present article is organized as follows. Section 2 gives an overview of our toolchain from the user's point of view, by describing the software components as well as the supported formats for input and output data. Sections 3 and 4 present in detail the two main software components of the toolchain, CÆSAR.BDD and KONG, respectively. Section 5 provides experimental results obtained by applying the toolchain to a large collection of 850 nets used in the 2022 edition of the Model Checking Contest; the validation of the toolchain outputs is also discussed. Finally, Section 6 gives concluding remarks.

2 Overview of the Toolchain

This section presents our toolchain for computing the concurrent places of a given net. We adopt the point of view of an end user, by first introducing the main software components of the toolchain (Sect. 2.1), and then defining the format of its input data (Sect. 2.2) and output data (Sect. 2.3).

2.1 Software Components of the Toolchain

Our toolchain consists of five different tools:

1. CÆSAR.BDD (developed in Grenoble, France) is one of the many components of the CADP toolbox [15] and can be obtained as part of this toolbox[1]. CÆSAR.BDD is written in C and its principles are detailed below in Sect. 3. For our experiments, we used version 3.7 of CÆSAR.BDD, available with CADP version 2022-j *"Kista"* of October 2022. CÆSAR.BDD internally uses the most recent version 3.1.0 of Fabio Somenzi's CUDD library for BDDs.
2. CONCNUPN (developed in Grenoble, France) is a 830-line Python 3.7 program for checking one-safeness and unit-safeness, and cross-checking the results provided by CÆSAR.BDD. The command-line options of CONCNUPN are compatible with those of CÆSAR.BDD. Information about the use of CONCNUPN is given in Sect. 5.4.

[1] https://cadp.inria.fr

3. Kong (developed in Toulouse, France) is a verification tool for Petri nets. Written in Python, it is available on GitHub[2] under the GPLv3 license. The principles and software architecture of Kong are presented in Sect. 4. For our experiments, we used version 3.0 of this tool.
4. Pnml2Nupn (developed in Paris, France) is a translator that converts Petri nets to NUPNs. This tool can be downloaded from the Web[3]. We used version 4.0 of Pnml2Nupn (February 2022).
5. Reduce (developed in Toulouse, France) is a tool for computing polyhedral reductions. This tool is invoked by Kong, and is also used by the Tina.tedd [9] and Smpt [4] model-checkers, which participate in the Model Checking Contest [7,25,26]. We use version 3.7 of Reduce (January 2022), which has been recently added to the Tina model-checking toolbox[4].

For the end user, Cæsar.bdd and Kong are the two main entry points of our toolchain. Both tools can be invoked separately on a net to compute the pairs of concurrent places. Yet, Kong uses Cæsar.bdd and Reduce as auxiliary tools, meaning that, if Kong is used, it will automatically invoke Cæsar.bdd under the hood, thus delivering results always equal or better than those provided by Cæsar.bdd. The user can also invoke Cæsar.bdd directly but, in such case, will not benefit from the enhancements brought by the reduction techniques implemented in Kong.

2.2 Input Formats for Petri Nets

Our toolchain takes as input nets (Petri nets or NUPNs) that are expected to be ordinary and one-safe (or even unit-safe, in the case of NUPNs). Concretely, these models can be provided in two different formats:

- The PNML (Petri Net Markup Language) format [22], which is a standard, XML-based representation adopted by most Petri-net tools; PNML can also describe NUPNs, as it is equipped with a "tool-specific" extension[5] for encoding all unit-related information present in NUPNs.
- The NUPN format[6], which is a concise, human-readable representation of NUPNs; this format supports a "`!unit_safe`" pragma certifying that the NUPN is unit-safe[7], a "`!multiple_arcs`" pragma indicating that the NUPN was obtained from a non-ordinary P/T net, and a "`!multiple_initial_tokens`" pragma indicating that the NUPN was obtained from a non-safe P/T net, the initial marking of which contains places with several tokens.

[2] https://github.com/nicolasAmat/Kong
[3] https://pnml.lip6.fr/pnml2nupn
[4] https://projects.laas.fr/tina
[5] https://mcc.lip6.fr/2022/nupn.php
[6] https://cadp.inria.fr/man/nupn.html
[7] When unit-safeness is known by construction, or if it has been proven later.

Depending on their format, the input files given to the toolchain should end with a suffix "`.pnml`" or "`.nupn`". It is worth noticing that CÆSAR.BDD and KONG are able to exploit the unit-related information present in their input files.

Conversion between both formats is easy: NUPN files can be translated to PNML files by invoking CÆSAR.BDD with its "`-pnml`" option, while PNML files can be translated to NUPN files by invoking either PNML2NUPN or the NDRIO tool[8] from the TINA toolbox.

The KONG tool supports both input formats, whereas CÆSAR.BDD only accepts the NUPN format. PNML files given to CÆSAR.BDD should therefore be pre-processed by PNML2NUPN. In practice, we observed that the depth-first-search order in which PNML2NUPN encodes the unit tree gives good results, while attempts at using other orders statistically degrade the performance of BDD calculations performed by CÆSAR.BDD.

2.3 Output Format for Concurrent Places

Given a net with n places, our toolchain displays information about concurrent places using a dedicated file format that was carefully designed:

- We opt for a unique format, excluding the coexistence of two distinct formats, namely a compressed binary format to minimize disk space, and a textual format intended for humans. Indeed, the problem addressed by our toolchain does not justify the definition of two separate formats, the development of conversion tools between these formats, and the tedious manipulations to perform such conversions.
- The format should be concise and readable by humans; it is therefore a textual format, not based on XML. Given that the concurrent-place relation is symmetric, it can be represented as a lower triangular matrix (named *concurrency matrix*) containing $n(n+1)/2$ characters. The (i, j)-th element of this matrix is equal to "1" if the corresponding places are concurrent, or to "0" if they are not. Each diagonal element of this matrix is "0" if the corresponding place is dead, or "1" otherwise.
  As a side note, CÆSAR.BDD may use "synonymous" characters for "0", in order to explain why two places are not concurrent. For instance, if the net is known to be unit-safe, any pair of places directly contained in the same unit cannot be concurrent, which is noted "=" rather than "0"; similarly, any pair of places contained in two nested units cannot be concurrent, which is noted "<" or ">" — see here[9] for details.
- Since the determination of concurrent places is PSPACE-complete, it may fail on large nets, by lack of memory or upon timeout, leaving a concurrency matrix that is not entirely computed. Instead of aborting the computation with no output at all, it is practically better to deliver a result that is a concurrency matrix with unknown values, which can later be replaced by

[8] https://projects.laas.fr/tina/manuals/ndrio.html
[9] https://cadp.inria.fr/man/caesar.bdd.html

either "0" or "1", based upon pessimistic assumptions (see, e.g., [11]). Such a matrix is called *partial* or *incomplete* and the file format uses the notation "." (a dot) for those elements corresponding to pairs of places where the concurrency relation is undecided. A concurrency matrix is *complete* iff it contains no "." element.

– Being quadratic in the number of places, the size of the concurrency matrix may get large. For instance, the nets used as benchmarks in Sect. 5 have an average number of places equal to 2665, leading to 3.4-Mbyte matrices, and the largest of these nets has 78,643 places, leading to a 2.9-Gbyte matrix. To ensure that large matrices can be stored in computer files of manageable sizes, our file format introduces a simple, yet effective run-length compression [14] on the lines of the concurrency matrix. Measured on 12,600+ examples, this compression reduces file sizes by a factor of 214 (mean value) up to 4270 (maximal value). The compression and decompression algorithms, together with an example of compressed matrix, are given in Appendix A.

3 Presentation of Caesar.bdd

3.1 Overview of Caesar.bdd

CÆSAR.BDD has been part of the CADP toolbox since 2004. Originally, it was introduced as an auxiliary tool for detecting dead transitions in the interpreted Petri nets generated by the LOTOS compiler [18] present in CADP; to this aim, symbolic methods (based on BDDs) were found to be more effective than explicit-state methods, and thus implemented in CÆSAR.BDD.

The tool was also capable of computing *concurrent units* in NUPNs (i.e., pairs of units that may simultaneously have a token in some reachable marking), a notion that is required to perform data-flow analyses on interpreted Petri nets [17].

As from 2013, CÆSAR.BDD has progressively been extended with new functionalities, such as the conversion of the NUPN file format to PNML (see Sect. 2.2) and the computation of 20 structural and behavioural properties of Petri nets (liveness, reversibility, etc.); the latter feature is routinely used by the organization team of the Model Checking Contest to check the properties of the models used during the competition.

The tool was further modified to enrich the NUPN file format with pragmas, place labels, transition labels, unit labels, and more stringent syntax and static-semantics constraints. Many new options were added to CÆSAR.BDD to query NUPN models: number of places, number of transitions, arc density, unit-tree height, etc.

CÆSAR.BDD was then extended with new algorithms for computing dead places, dead transitions, and concurrent places [10], which are useful notions when decomposing Petri nets into flat NUPNs (i.e., automata networks) [11] or hierarchical NUPNs.

Recently, the tool was enriched with new options that help detecting isomorphic Petri nets and NUPNs, a major issue when building and managing large

benchmarks with tens or hundreds of thousands of nets, which are generated automatically and potentially contain many "duplicates".

3.2 Command-Line Invocation of Caesar.bdd

CÆSAR.BDD is a command-line tool with many (currently, 54) options[10]. Computing the concurrent places is done by invoking CÆSAR.BDD with its "`-concurrent-places`" option. The name of the input NUPN file is given on the command line and, if the file is correct, the concurrency matrix is displayed on the standard output. Environment variables (in the POSIX style) can be set to control the state-space exploration performed by CÆSAR.BDD; they will be presented in the next section.

3.3 Principles of Caesar.bdd

To compute the concurrent places, CÆSAR.BDD uses dedicated data structures (which also serve for its other options) and implements four methods, which are detailed in [10, Sect. 5] and used in combination:

1. *Marking graph exploration* performs a forward traversal of the state space, starting from the initial marking. The visited markings are stored symbolically using BDDs, as implemented in the CUDD library. The user can bound the exploration either by setting the environment variable `CAESAR_BDD_TIMEOUT` to a maximal number of seconds, or by setting the environment variable `CAESAR_BDD_ITERATIONS` to a maximal depth. Once the exploration terminates, the BDD containing all visited markings is queried repeatedly to decide whether a given pair of places belongs or not to at least one visited marking. If the exploration was fully done, the concurrency matrix is complete; otherwise, only a subset of concurrent pairs of places can be inferred from the visited markings.
2. *Structural rules* are a collection of 7 theorems that enable one to conclude that certain pairs of places are concurrent (or not concurrent) by examining only their local context. In particular, if the net is a unit-safe NUPN, this information is exploited to conclude that two places belonging to the same unit or to two nested units are not concurrent. Structural rules are applied repeatedly until saturation.
3. *Quadratic under-approximation* explores an abstraction of the marking graph by approximating a reachable marking M by the set of all pairs[11] of places having a token in M. This is an under-approximation because the algorithm may miss exploring certain pairs of places that are actually reachable and concurrent. The exploration progresses forwards, starting from the initial marking (or, better, from all pairs of places already known to be concurrent), and produces a subset of concurrent pairs of places.

[10] https://cadp.inria.fr/man/caesar.bdd.html
[11] In this method, singletons are also considered as pairs $\{p, p\}$.

4. *Quadratic over-approximation* also does a forward exploration of the marking graph, again abstracted away using a set of pairs of places, but performs (improving the prior approach of [28]) an over-approximation instead of an under-approximation. Indeed, the algorithm explores all markings that it assumes to be potentially reachable because all the pairs of places in each of these markings are potentially concurrent. If the exploration completes, it produces a subset of non-concurrent pairs of places.

Cæsar.bdd applies these four complementary methods in sequence, in the specified order 1-2-3-4. The execution may terminate earlier, as soon as the concurrency matrix does not contain unknown values any more.

4 Presentation of Kong

4.1 Overview of Kong

Kong, the *Koncurrent places Grinder*, is an open-source[12] formal verification tool for Petri nets. It can take advantage of structural reductions to accelerate the verification of reachability properties.

Kong is written in Python and requires version 3.5 or higher. Scripts and models included in the GitHub 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 a user understand its inner workings.

The main application [5,6] of Kong is to accelerate the computation of the *concurrency relation* of a Petri net using polyhedral reductions, that is computing the concurrency relation on a reduced version of the input net, and then tracing back the result to the original net (more details in Sect. 4.4). But Kong is not only designed for this problem, as well as for one-safe and ordinary Petri nets. It also provides procedures to check if a given marking is reachable, without any constraints on the bound of places.

4.2 Command-Line Invocation of Kong

Kong offers a command-line interface with various subcommands to expose its different features. The tool provides several options, which are described in the documentation using "`--help`".

The main subcommands of Kong are "`conc`" and "`dead`" for, respectively, computing the concurrency relation and the list of dead places in a net.

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

[12] https://github.com/nicolasAmat/Kong

time limit and "`--bdd-iterations`" to limit the number of iterations. Debugging options are described in Sect. 4.6.

A call to "`kong.py conc`" delegates the computation of the concurrency 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 such option if we have a precomputed list of dead places for the reduced net.

For the sake of readability, it is possible to disable the run-length encoding of the concurrency matrix produced by KONG or to print the place ordering.

4.3 Auxiliary Tools Invoked by Kong

When computing a concurrency matrix, KONG relies on an external tool, such as CÆSAR.BDD, to compute the concurrency matrix of the reduced net.

In our approach, the computation of polyhedral reductions is delegated to the tool REDUCE. We can also reuse a precomputed reduced net with the option "`--reduced-net`".

4.4 Principles of Kong

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 [8,30], 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.

The correctness of our tool relies on two main theoretical notions. First, a new state space abstraction method, that we called *polyhedral abstraction* in [1,2], 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 [5,6], 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 $(N, M) \rhd_E (N', M')$ 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.

A TFG is a Directed Acyclic Graph (DAG) that can be built from an E-equivalence statement, $(N, M) \rhd_E (N', M')$, capturing the specific structure of the equations in E, that allows us to reason about the reachable markings by

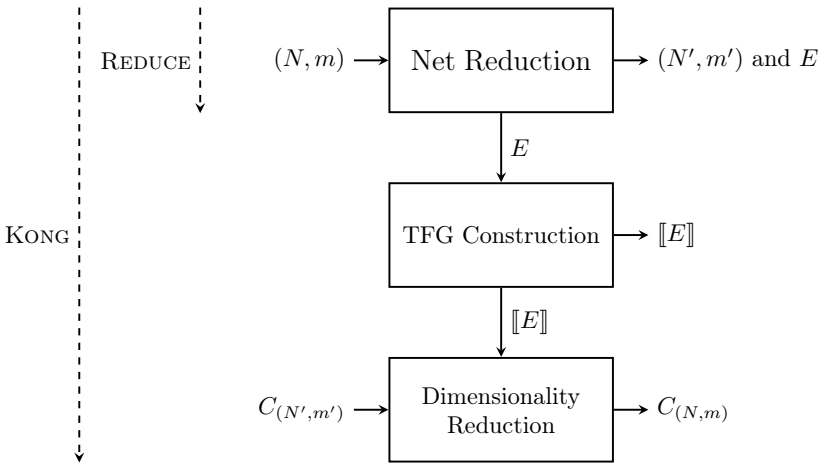

Fig. 3: Kong's architecture.

playing a token game on this graph. Kong can build a TFG from sequences of reductions computed using Reduce, and use it to symbolically explore the state space of the initial net.

4.5 Software 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. 3, 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 Reduce tool. Reduce outputs a reduced net (N', M') and a system of linear equations E. We display in Fig. 4 a sequence of structural reductions, with their equations, computed using Reduce. 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.

If the initial net has some non-trivial NUPN information, we are able to project the decomposition on the reduced net. This can be done using the graph structure of the TFG.

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) gives 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*. Otherwise, we have a non-trivial reduced net, in which case we need to compute the concurrency matrix of (N', M').

4.6 Net Reduction on a 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 4.

Structural reduction is performed iteratively, until no new reductions are possible. We display, Fig. 4, a sequence of three reductions that leads to the result computed with REDUCE; 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 [34]. 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, places p_1 and p_2 can be fused into a new place, a_1, and p_3, p_4 can be fused into a_2. Similar situations, where we can aggregate several places together, can be found by searching patterns in the net. After this step, we conclude with a new opportunity to reduce a redundant place, based on the structural invariant $a_1 = a_2$.

At the end of this process, we obtain the reduced net, (N', M'), with only 3 places instead of 6. We also obtain a system of four linear equations $E \triangleq (p_5 = p_4), (a_1 = p_1 + p_2), (a_2 = p_3 + p_4), (a_1 = a_2)$.

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`".

4.7 Token Flow Graph Construction

KONG can build the TFG associated with the linear system E; see Fig. 5. It is possible to output a graphical version of the TFG using option "`--draw-graph`", which requires the `graphviz` Python library. 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*, $q \twoheadrightarrow\!\!\bullet\, p$, represent equations of the form $p = q$ (or $p = q + r + \dots$ in which case we also have $r \twoheadrightarrow\!\!\bullet\, p, \dots$), corresponding to redundant places. In this case, we say that place p is *removed* by arc $q \twoheadrightarrow\!\!\bullet\, p$, 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

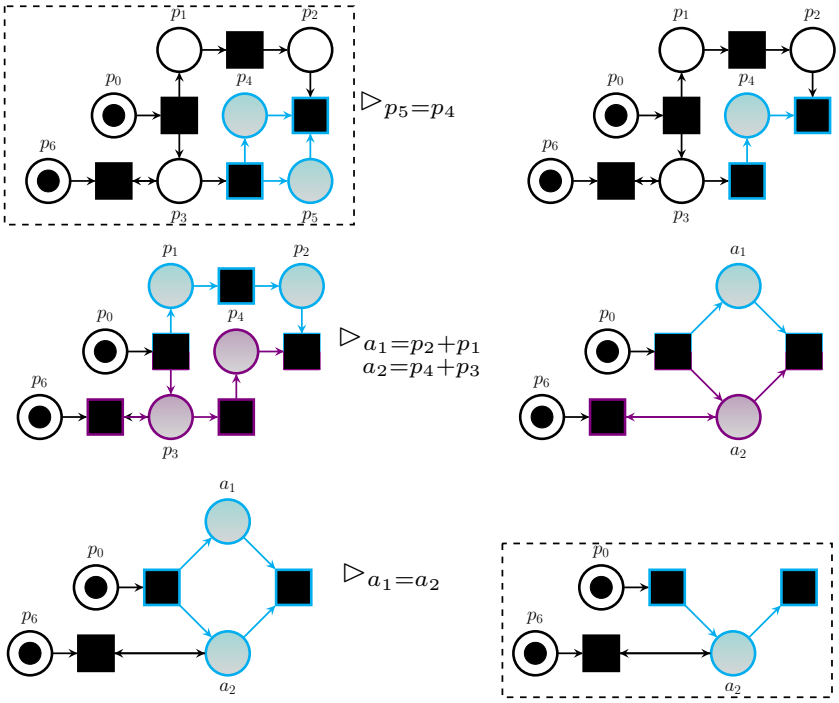

Fig. 4: Example of sequence of three reductions leading from the net N to N'.

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.

The idea is that each relation $X \twoheadrightarrow v$ or $v \circ\!\!\rightarrow X$ corresponds to one equation $v = \sum_{v_i \in X} v_i$ in E, and that all the equations in E should be reflected in the TFG. We also want to avoid situations where the same place is removed more than once, or where some place occurs in the TFG but is never mentioned in N, N' or E. All these constraints can be expressed using a suitable notion of well-formed graph built from E in [5,6].

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 of the graph (given a marking of N') then propagate them downwards while respecting the constraints dictated by the $\twoheadrightarrow$ 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).

```
# generated equations
# R |- p5 = p4
# A |- a1 = p2 + p1
# A |- a2 = p4 + p3
# R |- a1 = a2
```

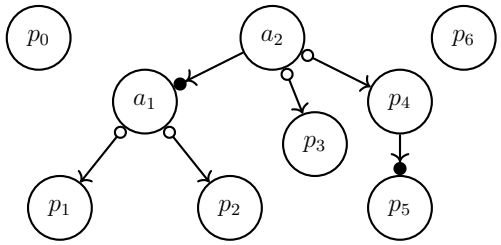

Fig. 5: Equations generated from net N, in Fig.4, and associated TFG $[\![E]\!]$.

4.8 Dimensionality Reduction Algorithm

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 [36]. It is possible to output this matrix with option "`--show-reduced-matrix`".

We can give an intuition for our *Dimensionality Reduction* algorithm using our example. For instance, we have that place a_2, in the reduced net N' of Fig. 4, is non-dead (because we can fire both input transitions). As a consequence, all the successors nodes of a_2 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 [5,6].

5 Experiments with the Toolchain

We now report about the assessment of our integrated toolchain to large benchmarks of significant complexity. We first describe our benchmarks, and how they were produced (Sect. 5.1), then present the results of experiments on these benchmarks (Sects. 5.2 and 5.3), and finally discuss the validation of these results (Sect. 5.4).

5.1 Benchmarks for Experiments

We have chosen to base our assessment on the collection of Petri nets provided by the Model Checking Contest (MCC) [7,26], a yearly international competition devoted to the evaluation of formal verification tools. This collection grows every year and has been constructed by gathering complex models provided by the Petri-net community. Given that the challenge of computing concurrent places is not covered by the MCC, we are confident that this collection provides an unbiased ground for our experiments.

The 2022 edition of the MCC[13] provides 128 (potentially parameterized) models, which amount to a total of 1628 individual nets. After excluding colored nets, this total drops down to 1387 nets, available in both PNML and NUPN formats. We opted for the NUPN format, taking as is any model natively provided in this format, while converting all other models from PNML to NUPN using the Pnml2Nupn translator. Since our experiments require unit-safe NUPNs (keeping in mind that one-safe, ordinary nets are trivial, unit-safe NUPNs), we proceeded in two (independent) steps:

- Among these 1387 nets, we kept those that were ordinary and whose initial marking was one-safe; this was done by discarding all NUPN files containing "`!multiple_arcs`" or "`!multiple_initial_tokens`" pragmas. This resulted in 777 nets, from which we selected those that were provably unit-safe, either because they contained a "`!unit_safe`" pragma, or by invoking Cæsar.bdd or ConcNupn to check unit-safeness[14]. This led to 705 nets, from which we further excluded 3 nets that were found to be (potentially) isomorphic to 3 other nets according to structural signatures computed using Cæsar.bdd with its "`-signature`" option. We therefore obtained 702 unit-safe, non-isomorphic NUPNs.
- Among these 1387 nets, we also considered the 610 ones containing "`!multiple_arcs`" and/or "`!multiple_initial_tokens`" pragmas. By deleting these pragmas, we obtained "new" nets, in which each arc has multiplicity one and each place contains at most one token initially. Again, we retained only those nets that were provably unit-safe, using Cæsar.bdd and/or ConcNupn, followed by a manual verification of the safeness indications given by the authors of these models. This resulted in 439 nets, from which we eliminated all (potentially) isomorphic nets, still using structural signatures; doing so, many nets were rejected, because a fraction of MCC nets are just derived from each other by changing the number of tokens in the initial marking. This left us with 148 unit-safe, non-isomorphic NUPNs.

By gathering both sets of 702 and 148 nets, we obtained a collection of 850 nets. We made sure that all of them have distinct structural signatures, so that the collection contains no isomorphic duplicates. Notice that this collection is twice as large as that used in our earlier work [3], which used 424 nets taken from the 2021 edition of the MCC. Table 1 summarizes statistical properties about our collection, highlighting its diversity of models.

To assess the performance of our toolchain on "traditional" Petri nets (as in our earlier work [3]), we built a second collection of 850 nets obtained by removing all unit-related information from the NUPNs of the collection described in Table 1. Notice that 18.6% of the first collection (i.e., 158 trivial NUPNs) are also present in the second collection, and that the second collection contains a

[13] https://mcc.lip6.fr/2022/models.php

[14] This succeeded for all the nets considered here, although, in general, unit-safeness may be difficult to determine for large nets.

property	yes	no
pure	57.3%	42.7%
free-choice	2.4%	97.6%
extended free-choice	2.4%	97.6%
marked graph	0.0%	100.0%
state machine	0.7%	99.3%

property	yes	no
connected	94.5%	5.5%
strongly connected	20.2%	79.8%
conservative	9.3%	90.7%
sub-conservative	16.6%	83.4%
trivial	18.6%	81.4%

feature	min value	max value	average	median	std deviation
#places	4	78 643	2 665.2	403	7 712
#transitions	1	1 070 836	10 479.5	677	48 460
#arcs	4	25 615 632	106 034	2 760	1 023 467
arc density	0	50	1.7	0.5	3.6
#units	4	78 644	1 317	67	5 800
height	1	2 891	24.4	2	164.6
width	3	78 643	1 273.7	56	5 799

Table 1: Structural, behavioural, and numerical properties of the 850 NUPNs.

pair of isomorphic duplicates, originating from two NUPNs that only differ by their unit trees.

5.2 Experiments on Nested-Unit Petri Nets

We assessed the performance of our toolchain by separately running Cæsar.bdd and Kong (which itself invokes Reduce to compute net reductions, and Cæsar.bdd to compute the concurrency matrices of reduced nets) on each of the 850 NUPNs of our first collection. Each run was made using two different wallclock timeout values: 10 minutes (corresponding to the patience of a human user waiting for the result computed by a tool) and one hour (corresponding to the duration granted by the MCC to each of its examinations). In order to vary the computation time allocated to the marking-graph-exploration phase of Cæsar.bdd, we set the environment variable `CAESAR_BDD_TIMEOUT` to three possible values: 25%, 50%, and 75% of the wallclock timeout. To perform our experiments, we used the machine clusters (Intel x64 processors) of the French Grid'5000 testbed[15]; further details about the experimental setting are given in Sect. 5.4.

Table 2 summarizes the outcome of these experiments, focussing on three types of results: (i) the percentage of complete matrices generated by a tool during a specified timeout; (ii) the average percentage of known values (i.e., "0" or "1") in all the matrices generated from the 850 NUPNs; and (iii) the average computation time taken by the tool to generate these matrices.

Finally, we present supplementary data to address potential biases that could affect the average computation time in our performance evaluation due to the

[15] https://www.grid5000.fr

observed results	tool	timeout 10 min	timeout 1 hour
complete matrices	CÆSAR.BDD	62.7%	66.6%
	KONG	64.5%	67.6%
average matrix completion	CÆSAR.BDD	72.1%	76.2%
	KONG	74.4%	77.4%
average computation time	CÆSAR.BDD	3 min 36 sec	17 min 44 sec
	KONG	3 min 14 sec	16 min 40 sec
average computation time, considering only those matrices fully computed by both tools	CÆSAR.BDD	46 sec	3 min 26 sec
	KONG	33 sec	2 min 26 sec

Table 2: Performance results of the toolchain on the first collection.

significant difference between general computation time and the timeout value. We calculate the average time based solely on matrices that are successfully computed by both tools (525 and 527 respectively, with timeouts of 10 min and 1 hour). By excluding partial matrices, we ensure a more accurate representation of the acceleration achieved when KONGis used in conjunction with CÆSAR.BDD.

In this table, `CAESAR_BDD_TIMEOUT` is set to 50% of the wallclock timeout (i.e., 5 and 30 minutes); our experiments show that, when `CAESAR_BDD_TIMEOUT` increases, the number of complete matrices slightly increases (by 1.7% maximum) but matrix completion decreases (by -4.8% maximum), as BDD calculations take most of the time, preventing later approaches (structural rules, quadratic under- and over-approximations) from being applied.

The key finding of Table 2 is that our toolchain can entirely solve 64.5% of our first collection in less than ten minutes.

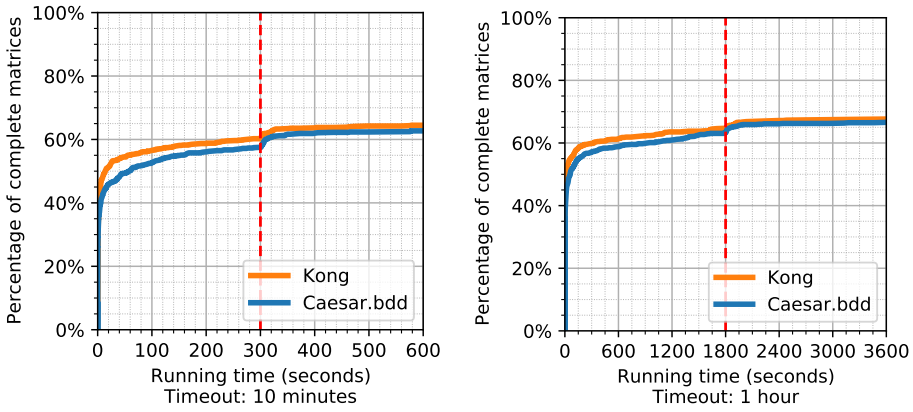

Fig. 6: Percentage of complete matrices on the first collection.

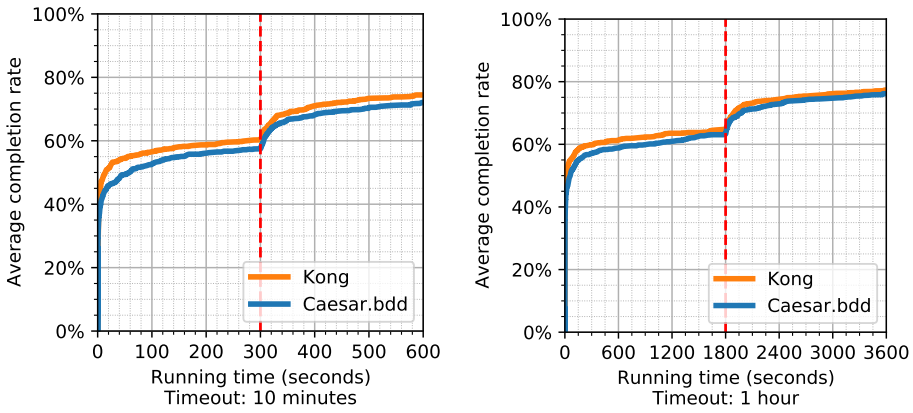

Fig. 7: Average matrix completion on the first collection.

Figures 6 and 7 provide additional information about the speed of the tools by showing the growth, as time elapses, of the number of complete matrices generated and the matrix completion rate. The red dotted vertical line corresponds to the 50% value given to `CAESAR_BDD_TIMEOUT` (5 and 30 minutes, respectively). These figures show very fast progress during the first minutes, which progressively slows down, followed by a rebound upon expiration of the BDD timeout, as other techniques (structural rules, under- and over-approximations) are executed and show their effectiveness. Despite the impression given by these figures, there is no horizontal asymptote, as infinite CPU time should allow the results to reach 100%.

5.3 Experimental Results on Petri Nets

We did the same experiments as in Sect. 5.2 on our second collection, which contains Petri nets without unit-related information. Table 3 presents the results of these experiments. Compared to Table 2, all percentages are significantly lower, since the symbolic exploration of reachable markings becomes more demanding once unit-related information has been dropped, thereby increasing the number of BDD variables: this confirms the practical importance of the NUPN concept. In Table 3, the value of `CAESAR_BDD_TIMEOUT` is set to 50%; our experiments show that, when `CAESAR_BDD_TIMEOUT` increases, the number of complete matrices slightly increases (by 3.8% maximum) but matrix completion decreases (by -6.3% maximum); these observations are in line with those of Sect. 5.2, the influence of `CAESAR_BDD_TIMEOUT` being somewhat greater when unit-related information is not present.

Figures 8 and 9 show the effectiveness of our toolchain as time elapses. Together with Table 3, these figures illustrate the added value of the structural

observed results	tool	timeout	
		10 min	1 hour
complete matrices	CÆSAR.BDD	52.5%	59.9%
	KONG	59.6%	63.4%
average matrix completion	CÆSAR.BDD	62.1%	68.1%
	KONG	69.8%	73.7%
average computation time	CÆSAR.BDD	4 min 30 sec	23 min 28 sec
	KONG	3 min 39 sec	19 min 16 sec
average computation time, considering only those matrices fully computed by both tools	CÆSAR.BDD	50 sec	5 min 18 sec
	KONG	29 sec	2 min 34 sec

Table 3: Performance results of the toolchain on the second collection.

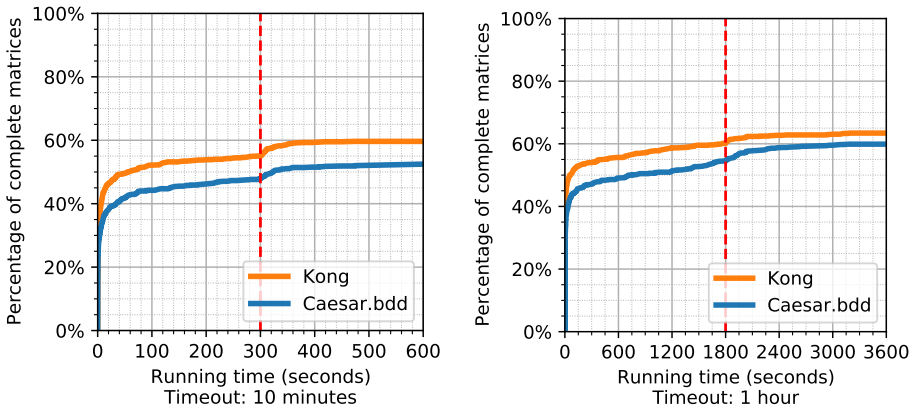

Fig. 8: Percentage of complete matrices on the second collection.

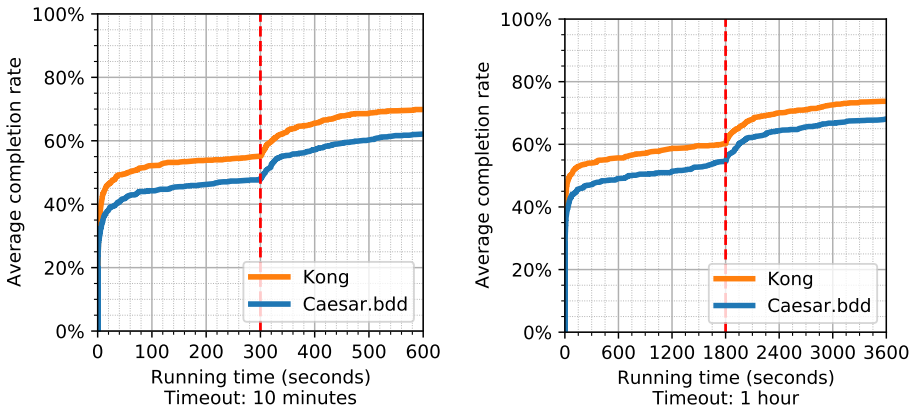

Fig. 9: Average matrix completion on the second collection.

reductions performed by Kong, especially when state-space exploration is time-intensive.

5.4 Validation of Results

Beforehand, the algorithms of Cæsar.bdd for computing concurrent places had been intensively validated on a test suite of 13,116 nets [10, Sect. 5.6]. When bringing together the software components of our toolchain, we also did a preliminary validation of Kong on roughly 7 200 "small" nets (having less than 20 places each), and brought enhancements to ensure plain interoperability between the components of the toolchain.

For the validation of results, we also used ConcNupn, which implements the quadratic under- and over-approximations of Sect. 3.3. This tool can be invoked exactly in the same way as Cæsar.bdd, but ignores the `CAESAR_BDD_TIMEOUT` and `CAESAR_BDD_ITERATIONS` environment variables, as it does not perform any BDD-based marking graph exploration.

On each x64 machine of the Grid'5000 testbed, we decided to run only two experiments in parallel, in order to benefit from the highest clock frequency of the processor, and to avoid dividing the 96-Gbyte RAM between too many tasks. The exit status and the standard error stream of each tool invoked to compute concurrency matrices was systematically recorded and analysed. This enabled us to proactively detect crashes, unexpected interrupts, and other run-time errors (e.g., memory shortage arising from two memory-intensive benchmarks concurrently running on the same machine). When there was a doubt about an execution, it was systematically restarted as a single task on a machine. All experiments were repeated to ensure that their results, including execution times and matrix completion rates, were consistent across executions.

We checked that each concurrency matrix generated by our experiments was syntactically correct, i.e., that the matrix after decompression was indeed triangular, only contained valid characters, and was not truncated (having as many rows as places of the corresponding net).

We developed a script that checks whether two matrices generated from the same net are *compatible*, which is defined as follows: for each "0" element in one matrix, the corresponding element in the other matrix should be "0" or "·", and for each "1" element in one matrix, the corresponding element in the other matrix should be "1" or "·". In particular, two complete matrices are compatible iff they are identical.

For each of the 850 benchmarks of Sect. 5.1, we generated many matrices by invoking our tools with various options: (i) Cæsar.bdd was invoked with several BDD-timeout values, including 0, which means that the marking graph exploration (see Sect. 3.3) is turned off, and $+\infty$ (actually, the value of the global timeout), which means that structural rules and the quadratic under- and over-approximations are never invoked; (ii) Kong was invoked with three different back-ends: Cæsar.bdd (invoked with several timeout values), ConcNupn, and a "dummy" tool that always gives Kong a reduced matrix containing only unknown values; (iii) ConcNupn was invoked too. For each non-trivial NUPN, we also generated these matrices for the Petri net obtained by removing all unit-related information from the NUPN. In total, this gave 17 matrices for each trivial NUPN and 34 matrices for each non-trivial NUPN; we cross-checked all these matrices two by two to make sure that they are compatible, and did not find any problem.

6 Conclusion

The concurrent-place problem is an old issue, which can be traced back at least to the 80s [23, Sect. 9]. It is of practical importance, as it subsumes the dead-place problem and seems unavoidable for decomposing a net into a flat or hierarchical NUPN. The formulation of this problem is extremely simple, but its complexity (PSPACE-complete) makes it difficult, especially on large nets, for which the solution may require too much memory or time, unless one is ready to accept partial solutions only (namely, incomplete concurrency matrices).

The present article proposes a toolchain that addresses this fascinating problem. The toolchain combines different tools that implement a wealth of complementary techniques: state-space exploration using BDDs, structural rules for concurrent places, quadratic over- and under-approximation of reachable markings, and polyhedral abstraction of the state space based on structural reductions and linear arithmetic constraints on markings.

When applied to a collection of 850 nets from the Model Checking Contest, our toolchain was able to determine all concurrent places for roughly 55% of the nets in less than one minute, and 65% in less than ten minutes. Building this toolchain enabled us to improve the individual tools, e.g., by making sure that

Kong introduces no performance overhead when directly invoking Cæsar.bdd on input nets that cannot be reduced.

Concerning future work, the clearest objective would be to improve the toolchain and its components to better address the challenging nets of our benchmark. For instance:

- Cæsar.bdd could be enhanced with heuristics to keep the dynamic reordering of BDDs under control (avoiding situations where, on large nets, most of the CPU time is spent in reordering) and by implementing more compact BDD encodings for the markings of unit-safe, hierarchical NUPNs (so as to reduce the number of BDD variables).
- Kong is also destined to evolve. We plan to explore new reduction rules, and we are particularly interested in reachability queries expressed using a Boolean combination of constraints over place markings. Another interesting problem would be the verification of generalized mutual exclusion constraints, as in [19], that amount to invariants $\sum_{p \in P} w_p.m(p) \leqslant k$ involving weighted sums over the marking of places, with $w_1, \ldots, w_n$, and k constants in $\mathbb{Z}$.

Finally, we believe that the concurrent-place problem should become part of the Model Checking Contest, so as to foster the development of new algorithms and tools for this problem. Such an evolution could take place after a few preliminary changes, such as replacing Boolean queries (e.g., *does the net contain dead places?*) by more precise ones (e.g., *what are the dead places of the net?*).

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including Cnrs, Renater, and several Universities as well as other organizations. We are grateful to Lom Messan Hillah for his Pnml2Nupn translator, to Fabio Somenzi for providing us with the latest version 3.1.0 of Cudd, to Bernard Berthomieu for providing the tool Reduce, and to Silvano Dal Zilio and Didier Le Botlan for their contributions to Kong.

A Run-Length-Encoding Compression

As mentioned in Sect. 2.3, each line of the concurrency matrix is compressed using a simple run-length-encoding scheme: any sequence of $n > 3$ consecutive identical characters is replaced by a single character followed by the value of n enclosed between parentheses. For instance, the following sequence of 18 characters: "`10000.........001`" is replaced by a sequence of 13 characters: "`10(4).(10)001`". Table 4 illustrates the compression of an entire matrix, and Table 5 gives an implementation in C of the compression and decompression algorithms (which we also implemented in Awk, Python, and Bourne shell).

This scheme enjoys three nice properties: (i) the size (in characters) of the compressed output is always less or equal to the size of the input; in practice, we observed a reduction factor of 214 (mean value) up to 4270 (maximal value) measured on 12,671 NUPNs; (ii) compressing an already compressed input has no effect; (iii) compression and decompression can operate on the fly (e.g., using coroutines, pipes, or data streams), meaning that it is not mandatory to generate a matrix entirely before starting to compress it, and that one can compare two (or more) compressed matrices without having to decompress them entirely in advance.

`1`	`1`
`01`	`01`
`001`	`001`
`0001`	`0001`
`0000.`	`0(4).`
`00000.`	`0(5).`
`0000001`	`0(6)1`
`0.0000..`	`0.0(4)..`
`010000.01`	`010(4).01`
`010000.001`	`010(4).001`
`010000.0001`	`010(4).0001`
`0010.0000001`	`0010.0(6)1`
`00010.0000001`	`00010.0(6)1`
`010000.0000001`	`010(4).0(6)1`
`01..000......11`	`01..000.(6)11`
`0.00000.0000000.`	`0.0(5).0(7).`
`0000..........001`	`0(4).(10)001`

Table 4: Sample uncompressed matrix (left) and its compressed version (right).

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>

int uncompress () {
    char c, previous = '\0';
    int repeat = 0;
    while (true) {
        if (repeat > 0) {
            assert (previous != '\0');
            assert (previous != '\n');
            putchar (previous);
            -- repeat;
        } else {
            c = getchar ();
            if (c == EOF)
                return 0;
            if (c != '(') {
                putchar (c);
                previous = c;
            } else {
                scanf ("%d)", &repeat);
                assert (repeat > 3);
                -- repeat;
            }
        }
    }
}
```

```
int compress () {
    char c, previous = '\0';
    int n, repeat = 0;
    while (true) {
        c = getchar ();
        if (c == previous) {
            assert (c != '\0');
            assert (c != EOF);
            assert (c != '\n');
            ++ repeat;
        } else {
            // flush repetition buffer, if any
            if (repeat > 3) {
                printf ("(%d)", repeat);
            } else if (repeat > 0) {
                for (n = 1; n < repeat; ++ n)
                    putchar (previous);
            }
            if (c == EOF)
                return 0;
            putchar (c);
            if (c == '0') {
                previous = '\0';
                repeat = 0;
            } else {
                previous = c;
                repeat = 1;
            }
        }
    }
}
```

Table 5: Decompression (left) and compression (right) algorithms.

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). Lecture Notes in Computer Science, vol. 12734, pp. 164–185. Springer (2021). https://doi.org/10.1007/978-3-030-76983-3_9
2. Amat, N., Berthomieu, B., Dal Zilio, S.: A Polyhedral Abstraction for Petri Nets and its Application to SMT-Based Model Checking. Fundamenta Informaticae **187**(2–4), 103–138 (2022). https://doi.org/10.3233/FI-222134, publisher: IOS Press
3. Amat, N., Chauvet, L.: Kong: A tool to squash concurrent places. In: Bernardinello, L., Petrucci, L. (eds.) Proceedings of the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'22). Lecture Notes in Computer Science, vol. 13288, pp. 115–126. Springer (2022). https://doi.org/10.1007/978-3-031-06653-5_6
4. Amat, N., Dal Zilio, S.: SMPT: A Testbed for Reachabilty Methods in Generalized Petri Nets. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods (FM'23). Lecture Notes in Computer Science, vol. 14000, pp. 445–453. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_25
5. Amat, N., Dal Zilio, S., Le Botlan, D.: Accelerating the Computation of Dead and Concurrent Places using Reductions. In: Laarman, A., Sokolova, A. (eds.) Proceedings of the 27th International SPIN Symposium on Model Checking of Software. Lecture Notes in Computer Science, vol. 12864, pp. 45–62. Springer, Aarhus, Denmark (2021). https://doi.org/10.1007/978-3-030-84629-9_3
6. Amat, N., Dal Zilio, S., Le Botlan, D.: Leveraging polyhedral reductions for solving Petri net reachability problems. International Journal on Software Tools for Technology Transfer (Dec 2022). https://doi.org/10.1007/s10009-022-00694-8
7. Amparore, E.G., Berthomieu, B., Ciardo, G., Dal-Zilio, S., Gallà, F., Hillah, L., Hulin-Hubard, F., Jensen, P.G., Jezequel, L., Kordon, F., Botlan, D.L., Liebke, T., Meijer, J., Miner, A.S., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y., van Dijk, T., Wolf, K.: Presentation of the 9th Edition of the Model Checking Contest. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Proceedings (Part III) of 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19, TOOLympics), Prague, Czech Republic. Lecture Notes in Computer Science, vol. 11429, pp. 50–68. Springer (Apr 2019). https://doi.org/10.1007/978-3-030-17502-3_4
8. Berthelot, G.: Transformations and Decompositions of Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and their Properties (ACPN'86). Lecture Notes in Computer Science, vol. 254, pp. 359–376. Springer (1987). https://doi.org/10.1007/978-3-540-47919-2_13
9. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA - Construction of abstract state spaces for petri nets and time petri nets. International Journal of Production Research **42**(14) (2004). https://doi.org/10.1080/00207540412331312688
10. Bouvier, P., Garavel, H.: Efficient Algorithms for Three Reachability Problems in Safe Petri Nets. In: Buchs, D., Carmona, J. (eds.) Proceedings of the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'21), Paris, France. Lecture Notes in Computer Science, vol. 12734, pp. 339–359. Springer (Jun 2021). https://doi.org/10.1007/978-3-030-76983-3_17

11. Bouvier, P., Garavel, H., Ponce de León, H.: Automatic Decomposition of Petri Nets into Automata Networks – A Synthetic Account. In: Janicki, R., Sidorova, N., Chatain, T. (eds.) Proceedings of the 41st International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'20), Paris, France. Lecture Notes in Computer Science, vol. 12152, pp. 3–23. Springer (Jun 2020). https://doi.org/10.1007/978-3-030-51831-8_1
12. Garavel, H.: Nested-Unit Petri Nets: A Structural Means to Increase Efficiency and Scalability of Verification on Elementary Nets. In: Devillers, R.R., Valmari, A. (eds.) Proceedings of the 36th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'15), Brussels, Belgium. Lecture Notes in Computer Science, vol. 9115, pp. 179–199. Springer (Jun 2015). https://doi.org/10.1007/978-3-319-19488-2_9
13. Garavel, H.: Nested-Unit Petri Nets. Journal of Logical and Algebraic Methods in Programming **104**, 60–85 (Apr 2019). https://doi.org/10.1016/j.jlamp.2018.11.005
14. Garavel, H.: Proposal for Adding Useful Features to Petri-Net Model Checkers. Tech. Rep. abs/2101.05024, arXiv Computing Research Repository (Dec 2020), https://arxiv.org/abs/2101.05024
15. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Springer International Journal on Software Tools for Technology Transfer (STTT) **15**(2), 89–107 (Apr 2013). https://doi.org/10.1007/s10009-012-0244-z
16. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10500, pp. 3–26. Springer (Oct 2017). https://doi.org/10.1007/978-3-319-68270-9_1
17. Garavel, H., Serwe, W.: State Space Reduction for Process Algebra Specifications. Theoretical Computer Science **351**(2), 131–145 (Feb 2006)
18. Garavel, H., Sifakis, J.: Compilation and Verification of LOTOS Specifications. In: Logrippo, L., Probert, R.L., Ural, H. (eds.) Proceedings of the 10th IFIP International Symposium on Protocol Specification, Testing and Verification (PSTV'90), Ottawa, Canada. pp. 379–394. North-Holland (Jun 1990)
19. Giua, A., DiCesare, F., Silva, M.: Generalized mutual exclusion constraints on nets with uncontrollable transitions. In: IEEE International Conference on Systems, Man, and Cybernetics. IEEE (1992). https://doi.org/10.1109/ICSMC.1992.271666
20. IEC: GRAFCET specification language for sequential function charts. International Standard 60848:2013, International Electrotechnical Commission, Geneva (Feb 2013)
21. ISO/IEC: LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva (Sep 1989), https://www.iso.org/standard/16258.html
22. ISO/IEC: High-level Petri Nets – Part 2: Transfer Format. International Standard 15909-2:2011, International Organization for Standardization – Information Technology – Systems and Software Engineering, Geneva (2011)
23. Janicki, R.: Nets, Sequential Components and Concurrency Relations. Theoretical Computer Science **29**, 87–121 (1984). https://doi.org/10.1016/0304-3975(84)90014-8
24. Karatkevich, A.: Conditions of SM-Coverability of Petri Nets (Sep 2012), https://www.researchgate.net/publication/267508814_Conditions_of_SM-Coverability_of_Petri_Nets

25. 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., Jezequel, L., He, C., Le Botlan, D., Li, S., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y., Walner, A., Wolf, K.: Complete Results for the 2021 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php (Jun 2021)
26. Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Donatelli, D., Dal Zilio, S., Jensen, P., Jezequel, L., He, C., Li, S., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y.: Complete Results for the 2022 Edition of the Model Checking Contest. http://mcc.lip6.fr/2022/results.php (Jun 2022)
27. Kovalyov, A.: A Polynomial Algorithm to Compute the Concurrency Relation of a Regular STG. In: Yakovlev, A., Gomes, L., Lavagno, L. (eds.) Hardware Design and Petri Nets, chap. 6, pp. 107–126. Springer, Boston, MA, USA (Jan 2000). https://doi.org/10.1007/978-1-4757-3143-9_6
28. Kovalyov, A., Esparza, J.: A Polynomial Algorithm to Compute the Concurrency Relation of Free-choice Signal Transition Graphs. In: Proceedings of the 3rd Workshop on Discrete Event Systems (WODES'96), Edinburgh, Scotland, UK. pp. 1–6. IEEE (Jun 1996)
29. Kovalyov, A.V.: Concurrency Relations and the Safety Problem for Petri Nets. In: Jensen, K. (ed.) Proceedings of the 13th International Conference on Application and Theory of Petri Nets (ICATPN'92), Sheffield, UK. Lecture Notes in Computer Science, vol. 616, pp. 299–309. Springer (Jun 1992). https://doi.org/10.1007/3-540-55676-1_17
30. 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
31. Murata, T.: Petri Nets: Analysis and Applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
32. Peterson, J.L.: Petri Nets. ACM Computing Surveys **9**(3), 223–252 (1977)
33. Semenov, A., Yakovlev, A.: Combining Partial Orders and Symbolic Traversal for Efficient Verification of Asynchronous Circuits. In: Ohtsuki, T., Johnson, S. (eds.) Proceedings of the 12th International Conference on Computer Hardware Description Languages and their Applications (CHDL'95), Makuhari, Chiba, Japan. IEEE (Aug–Sep 1995). https://doi.org/10.1109/ASPDAC.1995.486371
34. Silva, M., Terue, E., Colom, J.M.: Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In: Reisig, W., Rozenberg, G. (eds.) Advanced Course on Petri Nets (ACPN'96). Lecture Notes in Computer Science, vol. 1491, pp. 309–373. Springer (1996). https://doi.org/10.1007/3-540-65306-6_19
35. Wiśniewski, R., Karatkevich, A., Adamski, M., Kur, D.: Application of Comparability Graphs in Decomposition of Petri Nets. In: Proceedings of the 7th International Conference on Human System Interactions (HSI'14), Costa da Caparica, Portugal. pp. 216–220. IEEE (Jun 2014). https://doi.org/10.1109/HSI.2014.6860478
36. 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