Efficient Algorithms for Three Reachability Problems in Safe Petri Nets

Pierre Bouvier, Hubert Garavel

Efficient Algorithms for Three Reachability Problems in Safe Petri Nets

Pierre Bouvier Hubert Garavel

Univ. Grenoble Alpes, Inria, Cnrs, Grenoble Inp, Lig, France
E-mail: {pierre.bouvier,hubert.garavel}@inria.fr

Abstract

We investigate three particular instances of the marking coverability problem in ordinary, safe Petri nets: the Dead Places Problem, the Dead Transitions Problem, and the Concurrent Places Problem. To address these three problems, which are of practical interest, although not yet supported by mainstream Petri net tools, we propose a combination of static and dynamic algorithms. We implemented these algorithms and applied them to a large collection of 13,000+ Petri nets obtained from realistic systems — including all the safe benchmarks of the Model Checking Contest. Experimental results show that 95% of the problems can be solved in a few minutes using the proposed approaches.

1 Introduction

The present article focuses, in the framework of ordinary, safe Petri nets, on three related problems: the *Dead Place Problem*, which searches for all places that are never marked, the *Dead Transition Problem*, which searches for all transitions that can never fire, and the *Concurrent Places Problem*, which searches for all pairs of places that get a token in some reachable marking.

The two former problems characterize those parts of a net that are never active, and are thus relevant for simplifying complex Petri nets, especially those generated automatically from higher-level formalisms such as process calculi. The latter problem characterizes those parts of a net that can be simultaneously active, and plays a crucial role in the conversion of an ordinary, safe Petri net into an equivalent network of automata (see, e.g., [2]), an operation that opens the way to a compositional expression of Petri nets using process calculi.

The present article is organized as follows. Section 2 introduces the three problems, discussing their practical usefulness and theoretical complexity. Section 3 explains why mainstream model checkers are not optimal for these problems and presents the dedicated software tools that implement our algorithms, as well as

the comprehensive set of models used to evaluate these algorithms. Section 4 describes various algorithms for the Dead Place Problem and the Dead Transition Problem, and reports about their performance when applied to the set of models. Section 5 does the same as Sect. 4 for the Concurrent Places Problem. Finally, Sect. 6 gives concluding remarks.

2 Problem Statement

2.1 Basic Definitions

We briefly recall the usual definitions of Petri nets and refer the reader to classical surveys for a more detailed presentation of Petri nets.

Definition 1 *A (marked)* Petri Net *is a 4-tuple* (P, T, F, M_0) *where:*

1. P *is a finite, non-empty set; the elements of* P *are called* places.
2. T *is a finite set such that* $P \cap T = \varnothing$*; the elements of* T *are called* transitions.
3. F *is a subset of* $(P \times T) \cup (T \times P)$*; the elements of* F *are called* arcs.
4. M_0 *is a non-empty subset of* P*;* M_0 *is called the* initial marking.

Notice that the above definition only covers *ordinary* nets (i.e., it assumes all arc weights are equal to one). Also, it only considers *safe* nets (i.e., each place contains at most one token), which enables the initial marking to be defined as a subset of P, rather than a function $P \to \mathbb{N}$ as in the usual definition of P/T nets. We now recall the classical firing rules for ordinary safe nets.

Definition 2 *Let* (P, T, F, M_0) *be a Petri Net.*

- *A* marking M *is defined as a set of places* ($M \subseteq P$)*. Each place belonging to a marking* M *is said to be* marked *or, also, to possess a* token.
- *The* pre-set *of a transition* t *is the set of places* ${}^\bullet t \stackrel{\text{def}}{=} \{p \in P \mid (p, t) \in F\}$.
- *The* post-set *of a transition* t *is the set of places* $t^\bullet \stackrel{\text{def}}{=} \{p \in P \mid (t, p) \in F\}$.
- *A transition* t *is* enabled *in some marking* M *iff* ${}^\bullet t \subseteq M$.
- *A transition* t *can* fire *from some marking* M_1 *to another marking* M_2 *iff* t *is enabled in* M_1 *and* $M_2 = (M_1 \setminus {}^\bullet t) \cup t^\bullet$*, which we write as* $M_1 \xrightarrow{t} M_2$.
- *A marking* M *is* reachable *from the initial marking* M_0 *iff* $M = M_0$ *or there exist* $n \geq 1$ *transitions* $t_1, ..., t_n$ *and* $(n-1)$ *markings* $M_1, ..., M_{n-1}$ *such that* $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ ... \ M_{n-1} \xrightarrow{t_n} M$*, which we write as* $M_0 \xrightarrow{*} M$.

We then recall the basic definition of a NUPN, referring the interested reader to [5] for a complete presentation of this model of computation.

Definition 3 *A (marked)* Nested-Unit Petri Net *(acronym: NUPN) is a 8-tuple* $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unit})$ *where* (P, T, F, M_0) *is a Petri net, and where:*

5. *U is a finite, non-empty set such that $U \cap T = U \cap P = \varnothing$; the elements of U are called* units.
6. *u_0 is an element of U; u_0 is called the* root unit.
7. *$\sqsubseteq$ is a binary relation over U such that $(U, \sqsupseteq)$ is a tree with a single root u_0, where $(\forall u_1, u_2 \in U)\ u_1 \sqsupseteq u_2 \stackrel{\text{def}}{=} u_2 \sqsubseteq u_1$; intuitively[1], $u_1 \sqsubseteq u_2$ expresses that unit u_1 is transitively nested in or equal to unit u_2.*
8. *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.*

Because NUPNs merely extend Petri nets by grouping places into units, they do not modify the Petri-net firing rules for transitions: all the concepts of Def. 2 for Petri nets also apply to NUPNs, so that Petri-net properties are preserved when NUPN information is added. Finally, we recall a few useful NUPN concepts [5]:

Definition 4 *Let $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unit})$ be a NUPN.*

- *The predicate $\mathsf{disjoint}(u_1, u_2) \stackrel{\text{def}}{=} (u_1 \not\sqsubseteq u_2) \wedge (u_2 \not\sqsubseteq u_1)$ characterizes pairs of units neither equal nor nested one in the other.*
- *A marking $M \subseteq P$ is said to be* unit safe *iff $(\forall p_1, p_2 \in M)\ (p_1 \neq p_2) \Rightarrow \mathsf{disjoint}(\mathsf{unit}(p_1), \mathsf{unit}(p_2))$, i.e., all the places of a unit-safe marking are contained in disjoint units.*
- *The NUPN N is said to be* unit safe *iff its underlying Petri net (P, T, F, M_0) is safe and all its reachable markings are unit safe.*
- *N is said to be* trivial *iff its number of leaf units equals its number of places, meaning that N carries no more NUPN information than a Petri net.*

2.2 The Dead Places Problem

Definition 5 *Let (P, T, F, M_0) be a Petri Net. A place $p \in P$ is* dead *if there exists no reachable marking containing p.*

A more general definition is given in [4, Def. 4.16], where a place p is said to be dead at a marking M if it is not marked at any marking reachable from M. Our definition only considers the case where the net is safe and p is dead at the initial marking M_0.

In the present article, we will carefully avoid using the word *live*, because it is not the negation of *dead* according to the standard definition given in Petri-net literature, namely: a place p is live if, for any reachable marking M, there exists a marking M' that contains p and is reachable from M. Thus, any live place is not dead, but a non-dead place is not necessary live.

Definition 6 *Given a Petri net (P, T, F, M_0), the* Dead Places Problem *consists in finding all the dead places. Equivalently, this problem consists in computing*

[1] $\sqsubseteq$ is reflexive, antisymmetric, transitive, and u_0 is the greatest element of U for $\sqsubseteq$.

a vector of $|P|$ bits such that, for each place p, the bit corresponding to p in the vector is equal to one iff p is dead.

2.3 The Dead Transitions Problem

Definition 7 *Let (P, T, F, M_0) be a Petri Net. A transition $t \in T$ is* dead *if there exists no reachable marking in which t is enabled.*

The above definition is a particular case, for the initial marking M_0, of the definition given in [4, Def. 4.16], where a transition t is dead at a marking M if t is not enabled in any marking reachable from M.

Again, to avoid confusion, we will not use the word *live* throughout the present paper, as there exist multiple notions of liveness for transitions, e.g., *L1-live*, *L2-live*, *L3-live*, and *L4-live* [13, Sect. IV.C]; in our context, the negation of *dead* is not *live*, which means *L4-live* and is a too strong property, but *L1-live*, also known as *quasi-live*.

Definition 8 *Given a Petri net (P, T, F, M_0), the* Dead Transitions Problem *consists in finding all the dead transitions. Equivalently, this problem consists in computing a vector of $|T|$ bits such that, for each transition t, the bit corresponding to t in the vector is equal to one iff t is dead.*

Notice that this problem is different from the *deadlock freeness* problem: a net has a deadlock if there exists a reachable marking in which no transition is enabled, whereas a net has a dead transition if there exists a transition that is not enabled in any reachable marking.

Notice that this problem is also different from the *net quasi-liveness* problem: asking whether a net is quasi-live only calls for a Boolean answer, whereas the Dead Transitions Problem calls for a vector of Booleans. The latter problem is more general, as the net is quasi-live iff the solution of the Dead Transitions Problem is a vector of zeros. In practice, when studying a complex net, quasi-liveness is not sufficient, as one needs to know the exact set of dead transitions.

In practice, the Dead Places and Dead Transition Problems are relevant for several reasons. These problems are instances, with respect to Petri nets, of the more general *dead code* problem in software engineering. Dead code is generally considered as a nuisance for readability and maintenance, so that most software methodologies recommend to get rid of dead code. This is also the case for Petri nets in industrial automation, as the Grafcet specification prohibits Sequential Function Charts containing "unreachable" branches (i.e., Petri nets with dead places or dead transitions).

In model-checking verification, dead places and dead transitions are likely to increase the cost (in memory and time) of verification. Moreover, many global properties of a net can be changed to true or to false just by adding dead places

and/or dead transitions. This may disturb structural analyses or net transformations, by invalidating certain "good" properties (e.g., free choice) and thus lead to incorrect transformations or prevent the application of efficient algorithms relying on such properties. It is therefore important to detect and eliminate dead places and dead transitions to only consider a truly "minimal" Petri net.

2.4 The Concurrent Places Problem

Definition 9 *Let* (P, T, F, M_0) *be a Petri Net. Two places* p_1 *and* p_2 *are* concurrent*, which we write as* $p_1 \parallel p_2$*, if there exists a reachable marking* M *containing both* p_1 *and* p_2*.*

Proposition 1 *The relation* $\parallel$ *is symmetric and quasi-reflexive. It is reflexive iff the net has no dead place.*

Proof. Symmetry and quasi-reflexivity follow from Def. 9. As for reflexivity, a place is concurrent with itself iff it is not dead. Notice that both the relation $\parallel$ and its negation $\nparallel$ are neither transitive nor intransitive (since they are not irreflexive). □

The relation $\parallel$ can be found in the literature under various names: *coexistency defined by markings* [9, Sect. 9], *concurrency relation* [10] [15] [12] [11] [7], *concurrency graph* [16], etc. These definitions differ by details, such as the kind of Petri nets considered or the handling of reflexivity, i.e., whether and when a place is concurrent with itself or not. For instance, [9] defines that $p \parallel p$ is always false, while [10] defines that $p \parallel p$ is true iff there exists a reachable marking in which place p has at least two tokens.

Although the concurrency relation can easily be defined on non-ordinary, non-safe nets (see, e.g., [10]), this is of limited interest. For instance, consider a Petri net that is a state machine, with an initial marking M_0 containing a single place p_0, and all the places reachable from p_0: if there is initially a single token in p_0, each place is non-concurrent with each other, but as soon as there is initially more than one token in p_0, all the places become pairwise concurrent.

Definition 10 *Given a Petri net* (P, T, F, M_0)*, the* Concurrent Places Problem *consists in finding all pairs of concurrent places. Equivalently, since the concurrency relation is symmetric, this problem consists in computing a half matrix of* $|P|(|P|+1)/2$ *bits such that, for each two places* p_1 *and* p_2*, the corresponding bit in the half matrix is equal to one iff* $p_1 \parallel p_2$*.*

As mentioned above, solving the Concurrent Places Problem is practically useful, e.g., for decomposing a Petri net into a network of automata [2].

2.5 Complexity

Proposition 2 *The Dead Places Problem is a subproblem of the Dead Transition Problem.*

Proof. Given the set of dead transitions, one can easily compute the set of dead places. Indeed, each non-dead place belongs to the initial marking and/or the post-set of at least one non-dead transition. Notice that the converse does not hold, as the set of dead transitions cannot be directly inferred from the set of dead places. □

Proposition 3 *The Dead Places Problem is a subproblem of the Concurrent Places Problem.*

Proof. The diagonal of the Concurrent Places half matrix is the negation of the Dead Places vector. □

Proposition 4 *The Dead Places Problem, the Dead Transition Problem, and the Concurrent Places Problem are subproblems of the marking coverability problem, which is the problem of deciding whether a given marking is included in some reachable marking of a given Petri net.*

Proof. Given a set of places M, let $R(M)$ be the predicate: *does it exist a reachable marking containing all the places of M?* The Dead Places Problem can be expressed as: for each place p of the net, decide $R(\{p\})$. The Dead Transitions Problem can be expressed as: for each transition t, decide $R({}^\bullet t)$. The Concurrent Places Problem can be expressed as: for each two places p_1 and p_2, decide $R(\{p_1, p_2\})$. □

Proposition 5 *On safe nets, the Dead Places Problem, the Dead Transition Problem, and the Concurrent Places Problem are PSPACE-complete.*

Proof. One knows from [3, Th. 15] that the marking coverability problem is PSPACE-complete on safe nets. Although the Dead Places Problem is a subproblem of the marking coverability problem (see Prop. 4), its complexity is not lower: given a net N, let N' be the net consisting of N to which one adds a new place p and a new transition t such that ${}^\bullet t = M$ and $t^\bullet = \{p\}$; if N is safe, then N' is also safe; deciding $R(p)$ in N' requires to decide whether M is coverable in N. Given that the Dead Place Problem is a subproblem of the Dead Transition Problem (see Prop. 2) and of the Concurrent Places Problem (see Prop. 3), the two latter problems are also PSPACE-complete on safe nets.

2.6 Complete vs Incomplete Solutions

Because the three aforementioned problems are PSPACE-complete, there will always be Petri nets large enough to prevent the computation of exhaustive solutions on a given computer. Rather than an "all or nothing" approach (in which

an algorithm is considered to fail if it cannot compute all the dead places, dead transitions, or concurrent places of a given net), one should consider more pragmatic approaches in which an algorithm may stop or be halted after computing only a part of the solution, still leaving some results unknown.

Concretely, this means that the vectors of dead places and dead transitions, and the half matrices of concurrent places should contain three-valued logical results (zero, one, or unknown) rather than mere Boolean results. A solution is said to be *incomplete* if it contains unknown values, or *complete* otherwise. An efficient algorithm should produce as few unknown results as possible in a given lapse of time.

For the Dead Places Problem and Dead Transitions Problems, an incomplete solution can be turned into a complete one by replacing all unknown values by zeros, meaning that places and transitions are considered to be dead only if this has been positively proven.

For the Concurrent Places Problem, the elimination of unknown values depends on the context. For instance, the decomposition of safe Petri nets into automata networks [2] can work with incomplete half matrices, in which it replaces unknown values by ones — meaning that two places are assumed by default to be concurrent unless proven otherwise. Under such a pessimistic assumption, the algorithms that produce zeros in the half matrix are clearly more useful than the algorithms that produce ones. However, they may exist other applications based upon optimistic assumptions about concurrency, for which the latter kind of algorithms would be preferable.

3 Implementation and Experimentation

3.1 Relation to Temporal Logic

Given that our three reachability problems are particular instances of the marking coverability problem, one way to solve these problems is to encode them in temporal logic (e.g., CTL or LTL formulas) and to submit them to an existing model checker for Petri nets, e.g., one of those competing every year at the Model Checking Contest [1]. However, such an approach is not practical:

- The number of temporal-logic formulas required for each problem is linear or quadratic in the size of the Petri net — respectively, $|P|$, $|T|$, and $|P|(|P|+1)/2$. This is generally too large to be done manually, so one has to develop ad hoc tools that build these formulas (generating a huge file or a large number of small files), invoke a model checker on each of these formulas, then collect and aggregate the results of these invocations.
- Invoking a model checker repeatedly to evaluate hundreds or thousands of formulas on the same Petri net is inefficient. At each invocation, the formula and the net are parsed, checked for correctness, and translated from concrete

syntax to internal abstract representation: most of these steps are redundant given that all formulas are alike and correct by construction.

We thus believe that, although the three aforementioned problems can be reduced to the evaluation of temporal-logic formulas, it is better to express these problems at a higher level, namely by equipping Petri-net tools with built-in options (e.g., `-dead-places`, `-dead-transitions`, and `-concurrent-places`) dedicated to these problems. Not only such options would be easier for tool users, but they would also give tool developers more freedom to choose the most efficient approach(es).

For a tool based on some temporal logic, such options would allow a major boost in performance: (i) they could produce the formulas in the most appropriate temporal logic supported by the tool; (ii) they could generate the formulas directly in the internal abstract representation, thus saving the cost of writing intermediate files and later parsing these files; (iii) they would also save the cost of correctness checking, since the generated formulas would be known to be correct by construction; (iv) the Petri-net model would be read and analyzed only once; (v) because the tool knows in advance how many formulas are to be evaluated on this Petri net, it can try applying preliminary simplifications (e.g., structural reductions) and sophisticated optimizations to this model; (vi) the tool may profitably consider using *global model checking* (i.e., building the set of reachable markings first, then evaluating all the formulas on this state space) whereas, for a single formula, *local model checking* (i.e., on-the-fly evaluation that only explores a fragment of the state space relevant to the formula) is often the default choice.

If temporal logics are one possible way to address the three aforementioned problems, they are not the only way. The remainder of this article presents alternative approaches, whose algorithms are implemented in software tools.

3.2 Software Tools and File Formats

We implemented our proposed algorithms in two different tools:

- CÆSAR.BDD[2] is a verification tool for Petri nets and NUPNs. It is written in C (10,400 lines of code), uses the CUDD library for Binary Decision Diagrams, and is available as part of the CADP toolbox. CÆSAR.BDD provides many functionalities, among which solutions for the three aforementioned problems. For instance, it is routinely used to remove dead transitions from large interpreted Petri nets automatically generated from specifications written in higher-level languages such as LOTOS, LNT, AADL, etc.
- ConcNUPN is a prototype written in Python (730 lines of code) that implements algorithms for the three aforementioned problems and a few other

[2] `http://cadp.inria.fr/man/caesar.bdd.html`

functionalities. It is used to quickly prototype new ideas and to cross-check the results produced by CÆSAR.BDD.

Both tools take as input a file in the NUPN format[3] [5, Annex A], which provides a concise, human-readable representation for ordinary, safe Petri nets. Two translators[4] automatically convert the NUPN format to the standard PNML format [8] and vice versa.

Depending on the option given (`-dead-places`, `-dead-transitions`, or `-concurrent-places`), the tools produce as output a vector or a half matrix. Both are encoded in the same textual format [6, Sect. 8 and A] made up of one or more lines containing the characters "`0`", "`1`", and "`.`", the latter representing unknown values. If the input net is an non-trivial NUPN, some values "`0`" and "`.`" in the half matrix may be replaced by other characters giving additional information about the NUPN structure. Because half matrices can get large, a run-length compression scheme[5] is used, which, on average, divides by 2.4 the size of vectors and by 214 the size of half matrices — a reduction factor of 4270 was even observed on very large half matrices.

3.3 Data Sets

To perform our experiments, we used a collection of 13,116 models in NUPN format. Most of these models are derived from "realistic" specifications (i.e., nets obtained from industrial problems described by humans in high-level languages, rather than randomly generated Petri nets). Our collection contains all the ordinary, safe models from the former PetriWeb collection and from the 2020 edition of Model Checking Context[6].

A statistical survey confirms the diversity of our collection. The upper part of Table 1 gives the percentage of models that satisfy various (structural and behavioural) net properties, including the percentage of nets that are non-trivial NUPNs known to be unit safe (and thus, safe) by construction. The lower part of the table gives information about the size of the models: number of places, transitions, and arcs, as well as *arc density*, which we define as the number of arcs divided by twice the product of the number of places and the number of transitions, i.e., the amount of memory needed to store the arc relation as a pair of place×transition matrices.

[3] `http://cadp.inria.fr/man/nupn.html`
[4] `http://cadp.inria.fr/man/caesar.bdd.html` (when invoked with "`-pnml`" option) and `http://pnml.lip6.fr/pnml2nupn`
[5] `http://cadp.inria.fr/man/caesar.bdd.html` (see compression/decompression)
[6] `http://mcc.lip6.fr/models.php`

property	yes	no
pure	62.9%	37.1%
free choice	41.3%	58.7%
extended free choice	42.7%	57.3%
marked graph	3.5%	96.5%
state machine	12.1%	87.9%

property	yes	no
connected	94.0%	6.0%
strongly connected	14.3%	85.7%
conservative	16.5%	83.5%
sub-conservative	29.7%	70.3%
non trivial and unit safe	67.7%	32.3%

feature	min value	max value	average	median	std deviation
#places	1	131,216	282.4	15	2690
#transitions	0	16,967,720	9232.8	20	270,287
#arcs	0	146,528,584	72,848	55	2,141,591
arc density	0.0%	100.0%	14.5%	9.4%	0.2

Table 1: Structural, behavioural, and numerical properties of our models

4 Algorithms for Dead Places and Dead Transitions

This section discusses various algorithms for the Dead Places and Dead Transition Problems, both of which are addressed together as they are largely similar.

4.1 Marking Graph Exploration

The easiest (at least, conceptually) way of computing the dead places and dead transitions of a safe Petri net is to follow the global model checking approach mentioned in Sect. 3.1 by first exploring all the reachable markings, and then examining whether, during this exploration, some places have never been marked or some transitions have never been fired.

If the state space can be explored entirely, one obtains the complete vectors for dead places and dead transitions. If the state space is too large for being exhaustively generated, these vectors only contain zeros and unknown values, but no ones.

Actually, one can often avoid generating the state space entirely, still obtaining complete vectors. This can be achieved using algorithmic *shortcuts*, which stop the exploration as soon as all the information needed has been determined, following the idea of *on-the-fly* verification. For dead transitions, there is one shortcut: the exploration can stop if each transition has been fired at least once, meaning that the net is quasi-live. For dead places, there are two shortcuts: the exploration can stop if each place has been marked, meaning that there are no dead places, or if each transition has been fired at least once, meaning that a place never marked so far cannot receive a token from further transition firings.

These ideas have been implemented as follows in the CÆSAR.BDD tool. Reachable markings are represented using Binary Decision Diagrams, as symbolic exploration proved, in the case of safe Petri nets, to be much more efficient than

explicit-state exploration. If the input net is a non-trivial NUPN and is known to be unit-safe by construction, CÆSAR.BDD takes advantage of this information to significantly reduce the number of Boolean variables needed to encode reachable markings [5, Sect. 6]. The user can limit state-space construction by specifying a timeout or giving an upper bound on the depth of exploration. Observing which transitions have been fired is easy, as CÆSAR.BDD fires each transition separately (like in explicit-state exploration) rather than building a single BDD that encodes all the transition relation. Shortcuts are simply implemented using decreasing counters that store the number of remaining unknown values in the solution vectors.

4.2 Structural Rules

Marking-graph exploration is a brute-force approach, which may fail on large nets, yielding incomplete vectors. We now examine complementary algorithms of a lower complexity, which take as input a vector containing unknown values and produce as output the same vector in which some unknown values have been replaced by zeros or ones — meaning that all previously known values are preserved. In this section, we present such an algorithm based on eight simple "structural" rules (Prop. 6–13) that can decide whether certain places or transitions are dead or not.

Proposition 6 *Any place belonging to the initial marking M_0 is not dead.*

Proposition 7 *Any transition having no input place and no output place is not dead.*

The two next rules exploit the properties of safeness (we only consider Petri nets that are expected to be safe) and unit safeness (a large proportion of our models are known to be unit safe by construction — see Sect. 3.3) to characterize certain classes of dead transitions.

Proposition 8 *If the net is safe, any transition whose input places form a strict subset of the output places is dead.*

Notice that, if a transition has no input place and at least one output place, the net is not safe, as this transition can fire indefinitely often, accumulating tokens in its output places.

Proposition 9 (from [5, Prop. 8]) *If the net is unit safe, any transition having at least two input (resp. two output) places located in two non-disjoint NUPN units is dead.*

The four last rules allow to propagate, in certain cases, the fact that a place (resp. a transition) is dead or not dead to its adjacent transitions (resp. places).

Proposition 10 (from [4, Prop. 4.17(3)]) *If a place p is dead, all the transitions of ${}^\bullet p \cup p^\bullet$ are also dead.*

Proposition 11 (contrapositive of Prop. 10) *If a transition t is not dead, all the places of ${}^\bullet t \cup t^\bullet$ are also not dead.*

Proposition 12 *If a transition t is dead, any place p such that ${}^\bullet t = \{p\}$ is also dead.*

Proposition 13 (contrapositive of Prop. 12) *If a place p is not dead, any transition t such that ${}^\bullet t = \{p\}$ is also not dead.*

The algorithm below applies Prop. 6–13 iteratively until saturation. The vector of dead places is represented by P_1 and P_0, which are, respectively, the sets of places known to be dead and not dead, the places of $P \setminus (P_0 \cup P_1)$ being unknown. Similarly, the vector of dead transitions is represented by two sets T_1 and T_0. Before the algorithm starts, these four sets have been either initialized to $\varnothing$ or filled in by some other algorithm(s), such as the one of Sect. 4.1.

```
1  P_0 := P_0 ∪ M_0                                                  — from Prop. 6
2  T_0 := T_0 ∪ {t ∈ T | •t = t• = ∅}                                — from Prop. 7
3  T_1 := T_1 ∪ {t ∈ T | (•t ⊆ t•) ∧ (•t ≠ t•)} ∪                    — from Prop. 8 and 9
4     {t ∈ T | (∃(p_1, p_2) ∈ (•t × •t) ∪ (t• × t•)) ¬ disjoint (unit (p_1), unit (p_2))}
5  P' := ∅ ; T' := ∅ ; assert P_0 ≠ ∅
6  while P' ≠ P_0 ∪ P_1 loop
7        assert P' ⊊ P_0 ∪ P_1
8        for p ∈ (P_0 ∪ P_1) \ P' loop
9              if p ∈ P_1 then
10                   T_1 := T_1 ∪ •p ∪ p•                           — from Prop. 10
11             else if p ∈ P_0 then
12                   T_0 := T_0 ∪ {t ∈ T | •t = {p}}                — from Prop. 13
13       end loop
14       P' := P_0 ∪ P_1
15       assert T' ⊆ T_0 ∪ T_1
16       for t ∈ (T_0 ∪ T_1) \ T' loop
17             if t ∈ T_0 then
18                   P_0 := P_0 ∪ •t ∪ t•                           — from Prop. 11
19             else if t ∈ T_1 ∧ |•t| = 1 then
20                   P_1 := P_1 ∪ •t                                — from Prop. 12
21       end loop
22       T' := T_0 ∪ T_1
23 end loop
```

4.3 Linear Over-Approximation

Definition 11 *Let M_1 and M_2 be two markings, and t a transition. We write $M_1 \stackrel{t}{\Longrightarrow} M_2$ iff t is enabled in M_1 (i.e., ${}^\bullet t \subseteq M$) and $M_2 = M_1 \cup t^\bullet$.*

The relation differs from the usual firing relation $M_1 \xrightarrow{t} M_2$ (cf. Def. 2) in that the latter uses $(M_1 \setminus {}^\bullet t)$ instead of M_1. Thus, when a transition t fires according to Def. 11, each input place of t keeps its token, while each output place of t gets a token. Said otherwise, $\stackrel{t}{\Longrightarrow}$ behaves exactly as $\xrightarrow{t}$ if one assumes that each marked place holds an infinite number of tokens (hence, $M_1 \setminus {}^\bullet t = M_1$).

Proposition 14 *If a marking M is reachable from the initial marking M_0, i.e., $M_0 \xrightarrow{*} M$, there exists a marking M' such that $M_0 \stackrel{*}{\Longrightarrow} M'$ and $M \subseteq M'$.*

Proof. By induction on firing sequences $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots M_{n-1} \xrightarrow{t_n} M$.

The algorithm below is based on the contrapositive of Prop. 14. It performs a marking-graph exploration, starting from M_0 and using $\stackrel{*}{\Longrightarrow}$ instead of $\xrightarrow{*}$. During the exploration, each place, once marked, never loses its token, so that the state space can be simply represented using the set P' of visited places and the state P'' of explored places (with $P' \subseteq P''$). We speed up transition firings by attaching to each (non-dead) transition t a counter $c[t]$ containing the number of input places of t that have not been marked yet, so that t becomes fireable when $c[t]$ drops to zero. When the exploration completes, any place that has not been marked is dead (and thus added to P_1) and any transition that has not been fired is dead (and thus added to T_1). The converse is not true: unless each transition has a single input place, the algorithm may overlook some dead places and/or dead transitions, as it over-approximates the number of tokens and the set of fireable transitions.

```
 1  P' := ∅
 2  P'' := P_0 ∪ M_0
 3  for t ∈ T \ T_1 loop c[t] := |•t|
 4  while P' ≠ P'' loop
 5        assert (P' ⊊ P'') ∧ (P'' ∩ P_1 = ∅) ∧ (∀t ∈ T \ T_1) (c[t] = |•t \ P'|)
 6        let p = oneof (P'' \ P')
 7        P' := P' ∪ {p}
 8        for t ∈ p• \ T_1 loop
 9              c[t] := c[t] − 1
10              if c[t] = 0 then P'' := P'' ∪ t•
11        end loop
12  end loop
13  assert (P' = P'') ∧ (P' ∩ P_1 = ∅) ∧ (∀t ∈ T \ T_1) (c[t] = |•t \ P'|)
14  P_1 := P_1 ∪ (P \ P')
15  T_1 := T_1 ∪ {t ∈ T \ T_1 | c[t] > 0}
```

At line 14, the new set of dead places $(P \setminus P')$ contains the initial set of dead places P_1 if Prop. 10 has been applied before executing this algorithm.

4.4 Ordering of Algorithms

Let A_1, A_2, and A_3 denote the algorithms presented in Sections 4.1, 4.2, and 4.3, respectively. Let A_2 be split into two successive parts: $A_2 = A_2'; A_2''$, where A_2' comprises lines 1 to 4 (Prop. 6–9) and A_2'' comprises lines 5 to 23 (Prop. 10–13). It is easy to see that, after executing any of these algorithms, applying it immediately again never decreases the number of unknown values. But two successive executions of some algorithm may be fruitful if another algorithm has been successfully applied in between.

Thus, the next question is: in which order, and how many times, should these algorithms be applied? Our experiments suggest that executing them in the order $(A_2; A_3; A_1; A_2'')$ is likely to give the best results, based upon the fact that A_1, which is the most expensive algorithm, can take advantage of the information pre-computed by $(A_2; A_3)$. Namely, A_1 can avoid trying to fire those transitions known to be dead, and it can enhance the effectiveness of the algorithmic shortcuts defined in Sect. 4.1 by generalizing their triggering conditions: instead of checking if all places have been marked or all transitions fired at least once, A_1 can merely check if the solution vector contains no more unknown values, which better takes into account the existence of dead places or dead transitions, if any.

Finally, between each two algorithms, one also checks the number of remaining unknown values in the vector being computed, and the execution sequence stops if this number drops to zero. For instance, A_1 will not be applied if the prior execution of $(A_2; A_3)$ has produced a complete solution.

4.5 Experimental Results

We applied these algorithms to compute the dead places and dead transitions of the 13,116 models presented in Sect. 3.3. Our experiments are parameterized by a duration t, which is the number of seconds allocated to algorithm A_1 to symbolically explore (a fragment of) the graph of reachable markings — using the CUDD library for Binary Decision Diagrams. If t is zero, only the initial marking is explored and no transition is fired by A_1.

Our experiments reveal that at least 16.2% (resp. 15.9%) of the models contain dead places (resp. transitions), and that at least 20.4% (resp. 37.7%) of dead places (resp. transitions) are globally present among the models. Such important ratios confirm the practical relevance of detecting and eliminating dead places and transitions as a means to reduce the complexity of Petri nets.

Table 2 provides, for various values of t, three metrics about the computation of dead places (resp. transitions). The first metrics ("% complete vectors") gives the percentage of models for which the solution vector can be completely

problem	value of t	0	5	10	15	30	45	60	120	180	240	300
dead places	% complete vectors	44.6	93.0	93.6	93.8	94.4	94.6	95.1	95.3	95.4	95.5	95.6
	% unknowns values	48.9	33.5	32.0	31.3	28.9	28.3	27.9	27.1	26.5	25.9	25.8
	% vector completion	69.3	97.0	97.3	97.5	97.7	97.9	97.9	98.1	98.1	98.2	98.2
dead trans.	% complete vectors	29.3	92.3	92.9	93.2	93.7	94.0	94.1	94.4	94.7	94.9	95.0
	% unknowns values	68.7	65.0	63.5	62.0	61.0	59.3	57.8	54.6	45.2	39.9	29.8
	% vector completion	50.9	95.8	96.2	96.4	96.7	96.8	96.9	97.1	97.2	97.3	97.3

Table 2: Experimental results for dead places and dead transitions

computed within t seconds. The second metrics ("% unknowns values") gives the percentage of unknown values that remain after t seconds in all the computed solution vectors. The third metrics ("% vector completion") gives the mean, over all models, of percentage of known values in the computed solution vectors.

The first metrics shows, for $t = 0$, that algorithms A_2 and A_3 alone are sufficient to completely handle 44.6% (resp. 29.3%) of the models; but algorithm A_1, as soon as turned on, gives an major boost, pushing the percentage of models completely solved to 93.0% (resp. 92.3%) of the models; from there, increasing the value of t slowly increases this percentage; applying algorithm A_2'' again after A_1 fully solves 0.1% (resp. 0.1%) more models. The second metrics gives quite similar results concerning the resolution of unknown values, although the influence of A_1 is not as strong as with the first metrics; the percentage of unknown values does not quickly converge down to zero, due to a small number of large models that remain incompletely solved for long, with thousands of unknown values. The third metrics corroborates the first one, but exhibits higher success percentages, as each model counts for the proportion of known values in its solution vector rather than for a binary value (one for a fully complete model, and zero otherwise).

Additional measurements indicate that: (i) the shortcuts of algorithm A_1 are effective, as they are triggered for more than 82.6% (resp. 79.6%) of the models; (ii) 94.8% (resp. 94.0) of the models are fully solved in less than one second; the other models are (incompletely) processed in less than $1.56 \times t$ (resp. $1.48 \times t$) seconds; (iii) all nets having less than 74 places, 92 transitions, and 366 arcs can be fully processed with $t = 60$ (resp. $t = 180$).

5 Algorithms for Concurrent Places

This section presents various algorithms for the Concurrent Places Problem. In the sequel, we consider the elements of the half matrix as (unordered) pairs of places. The diagonal elements of the half matrix are also considered as pairs, even if both elements of these pairs are equal.

5.1 Marking Graph Exploration

Concurrent places can be determined by an exploration of reachable markings similar to the one presented in Sect. 4.1, based upon the fact that the places of each reachable marking are pairwise concurrent. If the state space can be generated exhaustively, one obtains a complete half matrix; otherwise, if the state space is too large for being explored entirely, the half matrix contains only ones and unknown values, but no zeros.

There are no obvious algorithmic shortcuts, apart from halting the exploration if the half matrix gets entirely full of known values, which only occurs if all unknown values turn out to be equal to one, since the exploration, as long as it has not been done exhaustively, only produces ones but not zeros. In practice, such a situation is unlikely, as Sect. 5.6 shows that, statistically, there are much less ones than zeros in half matrices.

5.2 Structural Rules

We now study the case where the state-space exploration of Sect. 5.1 has not been exhaustively performed, and propose complementary algorithms, with a lower complexity, that help reducing the number of unknown values in the half matrix. The two following rules exploit the information gained about non-dead places and transitions during marking-graph construction.

Proposition 15 *The places of the initial marking M_0 are pairwise concurrent.*

Proposition 16 *If a transition is not dead, its input places (resp. output places) are pairwise concurrent.*

We then apply algorithm A_2, i.e., the structural rules of Sect. 4.2, to identify (a subset of) the dead places and dead transitions. Based on this information, further unknown values can be eliminated from the half matrix.

Proposition 17 *(1) A non dead place is concurrent with itself. (2) A dead place is non concurrent with any other place, including itself.*

Proposition 18 *If a dead transition has two (distinct) input places, these places are non concurrent.*

The next rule exploits the assumption that the Petri nets considered are safe.

Proposition 19 *If a transition t (dead or not) has a single input place p, this place is non concurrent with any output place of t different from p.*

Proof. By contradiction: if there exists a reachable marking M containing p and some output place of t, the net is unsafe, as t can fire from M. Notice that, if t is dead, p is dead too, and the result follows directly from Prop. 17(2). □

The previous rule can be easily generalized to sequences of transitions having each a single input place. It is implemented using a transitive-closure algorithm.

Proposition 20 *For any path $(p_1, t_1, p_2, t_2, ..., p_n, t_n, p_{n+1})$ such that each transition t_i has a single input place p_i and at least one output place p_{i+1}, the places p_1 and p_{n+1} are non concurrent if they are distinct.*

The last rule exploits the unit-safeness property for those nets known to be unit safe by construction.

Proposition 21 (from [5, Prop. 6]) *If the net is a unit-safe NUPN, any two distinct places located in non-disjoint units are non concurrent. Formally:*

$$(\forall p_1 \in P)\ (\forall p_2 \in P)\ (p_1 \neq p_2) \wedge \neg \mathsf{disjoint}\,(\mathsf{unit}\,(p_1), \mathsf{unit}\,(p_2)) \Rightarrow p_1 \nparallel p_2$$

In particular, any two distinct places located in the same unit are non concurrent.

5.3 Quadratic Under-Approximation

From now on, if P' and P'' are two sets of places, we write $P' \otimes P''$ for the set of (unordered) pairs of places defined as $\{\{p', p''\} \mid (p' \in P') \wedge (p'' \in P'')\}$, assuming that the set notation $\{p', p''\}$ actually denotes a singleton if $p' = p''$. We represent the half matrix of concurrent places by R_0 and R_1, which are, respectively, the sets of pairs of places known to be non concurrent and concurrent, the pairs of $(P \otimes P) \setminus (R_0 \cup R_1)$ being unknown. For instance, the structural rules of Prop. 15, 16, and 17(1) can be summarized as follows:

$$R_1 := R_1 \ \cup\ (M_0 \otimes M_0) \ \cup \bigcup_{t \in T_0} ({}^\bullet t \otimes {}^\bullet t) \cup (t^\bullet \otimes t^\bullet)) \ \cup \bigcup_{p \in P_0} \{\{p\}\}$$

In this section, we propose an algorithm to detect more concurrent places. This algorithm starts from the set R_1 computed during prior phases and extends this set by examining all transitions having one or two input places, namely by combining Prop. 13, 16, and 18 together with the following result:

Proposition 22 *If two distinct places p_1 and p_2 are concurrent, p_2 is also concurrent with each output place of any transition t such that ${}^\bullet t = \{p_1\}$.*

The algorithm below stores, in a set R', pairs of places found to be concurrent. The algorithm is said to perform a *quadratic* approximation because each visited marking M is abstracted away and represented by its set of concurrent pairs $M \otimes M$ — contrary to algorithm A_3 of Sect. 4.3, which performs a *linear* approximation by storing only the set of places that appear in at least one visited

marking. The algorithm performs an *under*-approximation because it may miss exploring certain concurrent pairs that are actually reachable.

```
 1  R' := ∅
 2  while R' ≠ R1 loop
 3        assert R' ⊊ R1
 4        let {p1, p2} = oneof (R1 \ R')              — possibly with p1 = p2
 5        R' := R' ∪ {{p1, p2}}
 6        for t ∈ T | •t = {p1, p2} loop
 7              assert (1 ≤ |•t| ≤ 2) ∧ (t ∉ T1)      — from Prop. 13 and 18
 8              R1 := R1 ∪ (t• ⊗ t•)                  — from Prop. 16(b)
 9        end loop
10        for t ∈ T | (•t = {p1}) xor (•t = {p2}) loop
11              assert (|•t| = 1) ∧ (p1 ≠ p2) ∧ (t ∉ T1)   — from Prop. 13
12              R1 := R1 ∪ (({p1, p2} \ •t) ⊗ t•)          — from Prop. 22
13        end loop
14  end loop
```

5.4 Quadratic Over-Approximation

Our last algorithm is based upon the works of Kovalyov and Esparza, who proposed various algorithms [10, 12, 11] of polynomial complexity that compute a least fix-point for three rules derived from the Petri-net token game, and produce an over-approximation of the concurrency relation (i.e., a superset of concurrent pairs) from which one can safely obtains a subset of R_0, the set of non-concurrent pairs. Our algorithm below evolves their algorithms in several ways: (i) it does not assume that all places and all transitions are not dead and, instead, exploits the pre-existing set T_1 of dead transitions; (ii) it requires the input nets to be safe and uses this assumption to produce more accurate results by discarding unsafe markings, whereas the algorithms of Kovalyov and Esparza handle non-safe nets, with the alternative definition of diagonal values discussed in Sect 2.4 above; (iii) to get better and faster results, our algorithm reuses the sets of pairs R_0 and R_1 precomputed, e.g., by the algorithms of Sect. 5.1 to 5.3, whereas the algorithms of Kovalyov and Esparza start with no prior knowledge about concurrent pairs, i.e., $R_0 = R_1 = \varnothing$. We now formalize the over-approximation (which we call *quadratic* due to its memory cost) that underlies all these algorithms.

Definition 12 *Let R and R' be two sets containing pairs of places, t a transition, and M a marking. We write $R \stackrel{t}{\Longrightarrow} R'$ if we have ${}^\bullet t \otimes {}^\bullet t \subseteq R$ and $R' = R \cup (t^\bullet \otimes t^\bullet) \cup \{\{p\} \otimes t^\bullet \mid (p \in P \setminus {}^\bullet t) \wedge (\{p\} \otimes {}^\bullet t \subseteq R)\}$. We write $R \stackrel{*m}{\Longrightarrow} M$ if there exists R' such that $R \stackrel{*}{\Longrightarrow} R'$ and $M \otimes M \subseteq R'$.*

In contrast with the firing relation $\stackrel{t}{\longrightarrow}$ defined between two markings (cf. Def 2),

this relation $\overset{t}{\Longrightarrow}$ is defined between two sets of pairs. With $\overset{t}{\longrightarrow}$, the state space is the set of all reachable markings M, whereas, with $\overset{t}{\Longrightarrow}$, the (abstracted) state space is the union of all sets of pairs $M \otimes M$, for each reachable marking M.

Proposition 23 *In a safe Petri net, if a marking M is reachable from the initial marking M_0 (i.e., $M_0 \overset{*}{\longrightarrow} M$), then $M_0 \otimes M_0 \overset{*m}{\Longrightarrow} M$.*

Proof. By induction on firing sequences from $M_0 \otimes M_0$.

Our algorithm starts from $M_0 \otimes M_0$, to which the known concurrent pairs of R_1 are added, and explores, using two variables R' and R'', the state space of all pairs that can be reached by firing the relation $\overset{*}{\Longrightarrow}$ for all non-dead transitions. The non-concurrent pairs of R_0 are systematically excluded from the state space. Upon termination, all pairs that have not been explored are non concurrent for sure, and can thus be added to R_0. To speed up calculations, we reuse the counter $c[t]$ of Sect. 4.3, which now stores how many pairs of $({}^\bullet t \times {}^\bullet t)$ have not been yet proven concurrent; a transition is considered to be fireable when its counter drops to zero. For the conciseness of the algorithm, we introduce an auxiliary function $\mathsf{fire}\,(M, t, R) \overset{\text{def}}{=} (M \otimes {}^\bullet t \subseteq R) \wedge ((M \otimes {}^\bullet t) \cap R_0 = \varnothing)$.

```
 1  R' := ∅ ; R'' := (M0 ⊗ M0) ∪ R1
 2  T1 := T1 ∪ {t ∈ T | ((•t ⊗ •t) ∩ R0 ≠ ∅) ∨ ((t• ⊗ t•) ∩ R0 ≠ ∅)}
 3  for t ∈ T \ T1 loop c[t] := |•t| × (|•t| + 1)/2
 4  while R' ≠ R'' loop
 5        assert (R' ⊊ R'') ∧ (R'' ∩ R0 = ∅)
 6        assert (∀t ∈ T \ T1) c[t] = |(•t ⊗ •t) \ R'|
 7        let {p1, p2} = oneof (R'' \ R')              — possibly with p1 = p2
 8        R' := R' ∪ {{p1, p2}}
 9        for t ∈ T \ T1 | {p1, p2} ⊆ •t loop
10              c[t] := c[t] − 1
11              if c[t] = 0 then
12                    for p ∈ (P \ •t) ∪ t• | fire ({p}, t, R'') loop
13                          assert (p ∉ •t \ t•) ∧ (M0 ⊗ M0 ⟹*m {p} ∪ •t)
14                          R'' := R'' ∪ ({p} ⊗ t•)
15                    end loop
16              end if
17        end loop
18        for t ∈ T \ T1 | (c[t] = 0) ∧ ((p1 ∈ •t) xor (p2 ∈ •t)) ∧
19            fire (({p1, p2} \ •t), t, R'') loop
20              assert (|{p1, p2} \ •t| = 1) ∧ (M0 ⊗ M0 ⟹*m {p1, p2} ∪ •t)
21              R'' := R'' ∪ (({p1, p2} \ •t) ⊗ t•)
22        end loop
23        assert (∀t ∈ T \ T1) (∀p ∈ (P \ •t) ∪ t•) (c[t] = 0) ∧ fire ({p}, t, R')
24                ⇒ ({p} ⊗ t• ⊆ R'')
25  end loop
```

26 **assert** $(R' = R'') \wedge (R' \cap R_0 = \varnothing) \wedge (R_0 \subseteq (P \otimes P) \setminus R')$
27 $R_0 := (P \otimes P) \setminus R'$

5.5 Ordering of Algorithms

Let C_1, C_2, C_3, and C_4 denote the algorithms presented in Sections 5.1, 5.2, 5.3, and 5.4, respectively. Each of these algorithms needs to be applied only once. Analysis of dependencies suggests that these algorithms are best applied in the following order $(C_1; C_2; C_3; C_4)$, knowing that C_2 also invokes algorithm A_2 of Sect. 4.2. This execution sequence stops as soon as the number of unknown values in the half matrix drops to zero.

5.6 Experimental Results

We applied these algorithms to compute the half matrices of concurrent places for the 13,116 models presented in Sect. 3.3. As for algorithm A_1 in Sect.4.5, the execution of algorithm C_1 is parameterized by a maximum duration t allocated to the symbolic exploration of reachable markings. Because the computation of concurrent places for large models can be much longer than the computation of dead places and dead transitions, each execution run was bounded by a timeout of 4000 seconds, which, for various values of t, hits at most 0.82% of our models.

Our experiments reveal that, over the 26,577,437,180 pairs of places present in all half matrices of the models not interrupted by the timeout, 4.0% are concurrent, 67.0% are non-concurrent, the others being unknown.

Table 3 reuses the three metrics of Table 2 by adapting them from vectors to half matrices. The first metrics shows that 94.0% of the models can be completely solved for $t = 60$, which is slightly less than in Table 2, although the Concurrent Places Problem usually requires more CPU time. The second metrics decreases more slowly than in Table 2 and seems to stabilize at a much higher percentage of unknown values, which can be explained by the quadratic size of a few large, incomplete half matrices. However, for most models, the third metrics converges to a high completion rate similar to those of Table 2.

Additional measurements indicate that: (i) alone, the algorithms C_2, C_3 and C_4 can completely handle 51.0% of the models for $t = 0$ but, as soon as algorithm C_1 is turned on, it performs so well on the models whose reachable markings

value of t	0	5	10	15	30	45	60	120	180	240	300	360	420
% complete matrices	51.0	91.6	92.2	92.5	93.0	93.6	94.0	94.2	94.4	94.5	94.6	94.7	94.7
% unknowns values	45.0	44.7	44.7	44.4	44.4	43.7	43.7	43.7	43.6	43.6	43.6	43.6	43.6
% matrix completion	81.6	96.3	96.6	96.8	97.0	97.1	97.2	97.3	97.4	97.4	97.4	97.5	97.5

Table 3: Experimental results for concurrent places

can be fully explored that algorithms C_2, C_3 and C_4 only contribute for 1% to the number of complete half matrices; (ii) however, on large models that can not be fully explored using Binary Decision Diagrams, the algorithms C_2, C_3 and C_4 play a greater role by eliminating 22.5% of unknown values, in addition to the 33.8% already eliminated by C_1; (iii) all nets having less than 66 places, 64 transitions, and 256 arcs can be completely processed with $t = 60$.

6 Conclusion

In the present article, we studied three reachability problems that, although practically useful, are not well supported by mainstream Petri-net tools. As we could not find a unified algorithm for addressing each problem, we proposed instead a combination of methods (static vs dynamic state-space exploration, exact vs approximate solution, polynomial or exponential cost), which we implemented in two software tools (written in C and in Python) that cross check each other. We observed that our approach statistically performs well on a large number of realistic models but, given that the three problems are PSPACE-complete, there will always exist models too large for being processed in reasonable time.

Future work should focus on refined algorithms capable of handling even larger models, either by computing solutions with less unknown values or by providing equivalent results in shorter time. Among the various approaches that might be profitably applied, we can mention invariants, semi-flows, partial orders, stubborn sets (e.g., [14, Sect. 11]), SAT solving, explicit-state model checking (to specifically spot the unknown values that remain in a solution vector or half matrix computed by other algorithms), and net reductions (in this respect, we can already mention a recent tool named Kong[7] that computes concurrent places by invoking our CÆSAR.BDD tool in combination with net reductions based on polyhedral abstractions). To foster such research, we suggest [6] that our three problems, possibly extended to unsafe nets and/or colored nets, become integral part of Model Checking Contest.

Acknowledgements

The experiments of Sect. 5.6 have been performed using the French Grid'5000 testbed.

References

[1] Elvio Gilberto Amparore, Bernard Berthomieu, Gianfranco Ciardo, Silvano Dal-Zilio, Francesco Gallà, Lom Messan Hillah, Francis Hulin-Hubard, Peter Gjøl Jensen, Loïg Jezequel, Fabrice Kordon, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Andrew S. Miner, Emmanuel Paviot-Adet, Jirí Srba,

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

Yann Thierry-Mieg, Tom van Dijk, and Karsten Wolf. Presentation of the 9th Edition of the Model Checking Contest. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Proceedings (Part III) of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19: TOOLympics), Prague, Czech Republic*, volume 11429 of *Lecture Notes in Computer Science*, pages 50–68. Springer, April 2019.

[2] Pierre Bouvier, Hubert Garavel, and Hernán Ponce de León. Automatic Decomposition of Petri Nets into Automata Networks – A Synthetic Account. In Ryszard Janicki, Natalia Sidorova, and Thomas Chatain, editors, *Proceedings of the 41st International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'20), Paris, France*, volume 12152 of *Lecture Notes in Computer Science*, pages 3–23. Springer, June 2020.

[3] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity Results for 1-Safe Nets. *Theoretical Computer Science*, 147(1–2):117–136, 1995.

[4] Jörg Desel and Javier Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

[5] Hubert Garavel. Nested-Unit Petri Nets. *Journal of Logical and Algebraic Methods in Programming*, 104:60–85, April 2019.

[6] Hubert Garavel. Proposal for Adding Useful Features to Petri-Net Model Checkers. Available from `https://arxiv.org/abs/2101.05024`, December 2020.

[7] Hubert Garavel and Wendelin Serwe. State Space Reduction for Process Algebra Specifications. *Theoretical Computer Science*, 351(2):131–145, February 2006.

[8] 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.

[9] Ryszard Janicki. Nets, Sequential Components and Concurrency Relations. *Theoretical Computer Science*, 29:87–121, 1984.

[10] Andrei Kovalyov. Concurrency Relations and the Safety Problem for Petri Nets. In Kurt Jensen, editor, *Proceedings of the 13th International Conference on Application and Theory of Petri Nets (ICATPN'92), Sheffield, UK*, volume 616 of *Lecture Notes in Computer Science*, pages 299–309. Springer, June 1992.

[11] Andrei Kovalyov. A Polynomial Algorithm to Compute the Concurrency Relation of a Regular STG. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, chapter 6, pages 107–126. Springer, Boston, MA, USA, January 2000.

[12] Andrei Kovalyov and Javier Esparza. 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*, pages 1–6, 1996.

[13] Tadao Murata. Petri Nets: Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[14] Karsten Schmidt. Stubborn Sets for Standard Properties. In Susanna Donatelli and H. C. M. Kleijn, editors, *Proceedings of the 20th International Conference on Application and Theory of Petri Nets (ICATPN'99), Williamsburg, Virginia, USA*, volume 1639 of *Lecture Notes in Computer Science*, pages 46–65. Springer, June 1999.

[15] Alexei Semenov and Alexandre Yakovlev. Combining Partial Orders and Symbolic Traversal for Efficient Verification of Asynchronous Circuits. In Tatsuo Ohtsuki and Steven Johnson, editors, *Proceedings of the 12th International Conference on Computer Hardware Description Languages and their Applications (CHDL'95), Makuhari, Chiba, Japan.* IEEE, August–September 1995.

[16] Remigiusz Wiśniewski, Andrei Karatkevich, Marian Adamski, and Daniel Kur. 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*, pages 216–220. IEEE, June 2014.