Explicit Model Checking Engine for Reachability Analysis of Colored Petri Nets

Emil Normann Brandt, Jens Emil Fink Højriis,
Kira Stæhr Pedersen, and Jiří Srba

Aalborg University, Aalborg, Denmark

Abstract. Unfolding colored Petri nets into place/transition (P/T) nets is a standard approach for model-checking, leveraging established tools and techniques for basic Petri nets. However, the unfolding process often leads to a combinatorial explosion in the number of places and transitions, creating a significant bottleneck in analyzing complex colored Petri nets. We introduce a new verification engine for Petri nets with finite color domains that bypasses the costly unfolding process. Our engine employs an explicit, on-the-fly state-space exploration, utilizing an optimized binding generator and linear programming-based approximation techniques to enhance performance. Integrated into the open-source TAPAAL model checker, our engine is evaluated on an extensive benchmark from the Model Checking Contest (MCC) 2024. It demonstrates superior performance over the state-of-the-art unfolding approaches.

1 Introduction

Distributed systems are inherently complex, posing significant challenges to correct design and analysis. Petri nets (PNs) [24] have emerged as a powerful formalism for modeling complex distributed behaviors. To enhance their expressiveness and applicability, various extensions have been proposed. Among these, colored Petri nets (CPNs), suggested by Kurt Jensen [15] , introduce token colors, enabling more compact and flexible modeling of complex systems.

Colored Petri Nets (CPNs) are susceptible to the state-space explosion problem, particularly when large color types are used. Any CPN with finite color types is expressively equivalent to an ordinary Petri Net (PN), meaning it can be unfolded [22] into its PN counterpart. Unfolding is advantageous because it enables the direct application of existing PN optimization techniques and a large selection of tools. However, unfolding introduces a significant overhead, and the resulting PN may become so large that it exceeds time and memory constraints. On-the-fly (or explicit) verification techniques enable gradual exploration of the state space, allowing the identification of reachable markings with specific properties without requiring to complete the unfolding process.

We present a novel explicit verification engine for colored Petri nets that enables efficient reachability analysis on large nets without requiring their unfolding. We outline the design choices and implementation details that make our engine both efficient and competitive with state-of-the-art unfolding approaches.

Specifically, we detail the critical code components for successor marking generation and describe over-approximation techniques based on linear programming, which often enable fast resolution of negative reachability queries. Additionally, we present a method for handling fireability queries (checking transition enabledness) which are typically more challenging than pure cardinality queries.

We experimentally evaluate the performance of our explicit verification engine on a benchmark of colored Petri nets and queries from the annual Model Checking Contest (MCC) [1]. We compare its performance with the unfolding approach implemented in TAPAAL [9,14,4], the winner in the reachability category at MCC'24 [19] and MCC'25 [20]. Our results demonstrate a 9% improvement in the number of queries answered by our explicit engine compared to unfolding.

Related Work. The state-of-the-art CPN model checkers that competed in the model checking contest during the past years, including ITS-Tools [25], Lola [27] and TAPAAL [14] are solely based on the unfolding approach [1]. Additionally, it has been shown that TAPAAL unfolding implementation for the reachability analysis is currently the most efficient unfolding approach [4], compared to the MCC unfolder [8] (used also by TINA [2] and LoLA [27]), ITSTools unfolder [25] and Spike unfolder [7] (also used by MARCIE [12] and Snoopy [11]).

CPN-Tools [16] is a popular tool for modelling, simulation and state-space exploration of CPNs with complex color data structures and guards defined in standard ML, however, it does not require finiteness of the color types and reachability analysis is in general undecidable. While the state-space analysis is via unfolding, the tool ASAP [26] aimed at providing an explict exploration engine, however, neither the binary nor the source code of ASAP is accessible/maintained anymore and hence we cannot compare with its performance.

2 Colored Petri Nets

In our verification engine as well as in the MCC competition [1], we consider colored Petri nets over a finite set *Colors* of all possible colors and finitely many color types, each associated with a subset of *Colors*. More precisely, we support color types with integer ranges as well as finite enumeration color types with an ordering relation on their colors and cyclic successor and predecessor operations $++$ and $--$, respectively. Products of color types are supported as well and each color type can have a number of associated variables. By arc expressions AE_τ we understand the set of all multisets over colors and variables in the color type $\tau \in CT$. If we e.g. have a color type with two colors a and b then $(2'a + 3'b) - 1'(a{+}{+})$ is a notation for an arc expression that represents the multiset $\{a, a, b, b\}$. Let $AE = \cup_{\tau \in CT} AE_\tau$. By $\mathcal{G}$ we denote the set of guard expressions, i.e. Boolean combinations of atomic predicates comparing colors and variables using the standard comparison operators $<, \leq, =, \neq, \geq, >$.

Definition 1. *A CPN is a tuple* $\mathcal{N} = (P, T, CT, \mathbb{V}, C, \mathcal{A}, \mathcal{I}, G, M_0)$ *where*

- P *and* T *are finite sets of places and transitions s.t.* $P \cap T = \emptyset$,

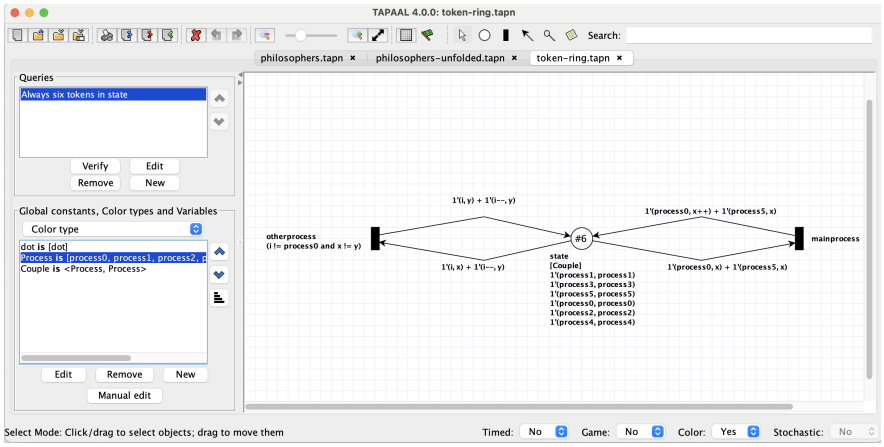


Fig. 1: Token ring [21] CPN example

- *CT is a finite set of color types and $\mathbb{V}$ is a finite set of variables,*
- *$C : P \to CT$ is a function assigning color types to places,*
- *$\mathcal{A} : (T \times P) \cup (P \times T) \hookrightarrow AE$ is a partial function assigning arc expressions to arcs connecting places and transitions and respecting the color type of the connected place, i.e. $\mathcal{A}(t,p) \in AE_{C(p)}$ and $\mathcal{A}(p,t) \in AE_{C(p)}$ for all $p \in P$ and $t \in T$ where the functions are defined,*
- *$\mathcal{I} : P \times T \to \mathbb{N}^{\infty}$ is an inhibitor arc weight function,*
- *$G : T \to \mathcal{G}$ is a function assigning guard expressions to transitions, and*
- *M_0 is the initial marking where a marking $M : P \to \mathcal{B}(Colors)$ is a function assigning multisets of colors to places while satisfying that $M(p)$ only contains colors from the color type $C(p)$ for every $p \in P$.*

Figure 1 depicts a CPN example in TAPAAL GUI [9], modelling a token ring [21]. The color type Process is an enumeration of six elements (process0 to process5) and Couple is a product color type containing pairs of processes. There are three variables x, y and i of the color type Process. The place 'state' contains six tokens in the initial marking and there are two transitions in the net. The transition 'otherprocess' is the only one that can fire in the initial marking M_0. In order for it to fire, we need to bind the variables to the concrete colors while satisfying the transition guard and at the same time the resulting multiset of colors on the connected arc expressions must be a subset of the colors present in the place 'state'[1]. For example, if we assume the binding $b(i) = \text{process1}$, $b(x) = \text{process1}$ and $b(y) = \text{process0}$, we obtain a valid binding that satisfies the guard $i \neq \text{process0}$ and $x \neq y$. We can fire the transition 'otherprocess' in this binding, which removes two tokens $(\text{process1}, \text{process1})$ and $(\text{process0}, \text{process0})$ from the place 'state' and adds the tokens $(\text{process1}, \text{process0})$ and $(\text{process0}, \text{process0})$. We denote such a transition firing by $M_0 \xrightarrow{t,b} M$ where M is the marking after removing and adding the two tokens.

[1] Our example has no inhibitor arcs; should an inhibitor arc (p,t) be connected to a transition t then in order to fire t we additionally require that $M_0(p) < \mathcal{I}(p,t)$.

A P/T net is a colored Petri net with one color type called *dot* and a single color $\bullet$, no variables and all guards being *true*. The classical verification method for CPNs with finite color types is by unfolding [22] the colored net into an equivalent P/T net. Here for every place in the CPN we create a copy for every possible color from its color type. Similarly, for every transition and valid binding, we create a new transition and connect them by arcs accordingly. In our token ring example from Figure 1, the unfolding contains 36 places, 156 transitions and 624 arcs. Hence unfolding nets with large number of colors, in particular product colors, can result in very large P/T nets and the unfolding process itself can exceed the time and memory limits. We instead avoid the unfolding step and implement a direct (explicit) model checking engine.

We are interested in reachability analysis, asking whether there exists a marking M reachable from the initial marking M_0, such that M satisfies a given *cardinality* γ^c or *fireability* γ^f formula defined by the following abstract syntax

$$\gamma^c ::= p \bowtie n \mid \mathit{true} \mid \mathit{false} \mid \gamma^c \wedge \gamma^c \mid \gamma^c \vee \gamma^c \mid \neg\gamma^c$$
$$\gamma^f ::= t \mid \mathit{true} \mid \mathit{false} \mid \gamma^f \wedge \gamma^f \mid \gamma^f \vee \gamma^f \mid \neg\gamma^f$$

where $p \in P$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >, =, \neq\}$ and $t \in T$. To evaluate a formula on a marking M, we replace p with $|M(p)|$ (number of tokens in p) and t with true/false, depending on whether t is enabled in M or not.

3 Optimizations Implemented in the Verification Engine

We shall now outline three most-impactful techniques implemented in our tool.

Successor generator. The most critical part of the engine is the generation of marking successors as this operation is constantly repeated during the state-space search. We implement several improvements to the basic idea of iterating over all transitions and all possible bindings in order to identify the successor markings. The main components are summarized in Algorithm 1.

As the number of valid bindings for a transition can be large, we generate the successor markings one by one while storing the information about the last binding used on a given transition. We assume that both for transitions T and a set of bindings $\bar{\mathbb{B}}(M, t)$ in a marking M for a transition t, there is a function *first* that returns the first element of these sets (in some implicit order) and a function *next* that returns the next element or $\top$ if no further element exist.

Each call to `NextSuccessor`(M) at line 2 recovers from the global map the pair (t, b) of a transition and a binding to be explored next. As long as $t \neq \top$, we perform at line 4 an inexpensive optimization by testing whether some of the inhibitor arcs disables t or whether the number of tokens (irrelevant of the color) in some place is strictly smaller than the lowest possible cardinality of the corresponding arc after substituting variables with colors (this is precomputed and cached). If the test succeeds, we know that t cannot be enabled under any binding and we jump to line 15 where we consider the next possible transition together with its first binding. Otherwise, we cycle through the possible bindings

Algorithm 1: `NextSuccessor`(M) - successor generator

```
   /* store is a globally available map linking a marking M with
      transition-binding pair, initially (first(T), first(B̄(M, first(T)))) */
1
      Input  : A marking M
      Output : A successor marking M' of M or ⊤ if no further successor exists
2     (t, b) ← store[M] ;
3     while t ≠ ⊤ do
4        if |M(p)| > I(p, t) ∨ |M(p)| < min-cardinality(A(p, t)) for some p ∈ P
           then
5           go to line 15;            /* t cannot become enabled in M    */
6        end
7        while b ≠ ⊤ do
8           if M -t,b-> M' for some M' then
9              b ← next(b, B̄(M, t));
10             store[M] ← (t, b);
11             return M';
12          end
13          b ← next(b, B̄(M, t));
14       end
15       (t, b) ← (next(t,T), first(B̄(M, t)));
16    end
17    store[M] ← (t, b);
18    return ⊤ ;
```

for the transition t in order to discover possible successors. A major performance gain is achieved by constructing $\bar{\mathbb{B}}(M,t)$ (used at line 9) using an interval analysis of possible colors that each variable can be bound to in order to enable the transition t in the marking M, instead of naively iterating through all possible colors of variables. Each time a new successor is discovered, we save the next possible candidate in the global map (to be used in the next call to the successor generator) and return the discovered successor marking M' at line 11.

Our optimized successor generator guarantees that the sequence of markings obtained by successive calls to `SuccessorGenerator`(M) eventually contains the value $\top$ and enumerates all successor markings of M .

Overapproximation by color removal. The idea of removing information about the concrete colors in markings and hence overapproximating the CPN behavior was presented in [18], however, without correctly dealing with arc expressions containing subtraction. We improve the technique to also work for expressions of the form $2'a-1'x$ where $2'a$ stands for two tokens with color a from which the color that the variable x binds to is subtracted—depending on the binding of x, this arc expression evaluates to either one (if x binds to a) or two (if x binds to a different color) tokens of color a. In general, for any arc expression ae present in the CPN, we precompute the interval [*min-cardinality*(ae), *max-cardinality*(ae)] such that the number of tokens required by ae is within this interval for any possible binding. Our tool then constructs the *color ignorant* P/T net where

each transition is replaced by a number of new transitions for each possible permutation of cardinalities for incoming and outgoing arcs. Let $t \in T$ and let $\mathit{perm}_t : (\{t\} \times P) \cup (P \times \{t\}) \to \mathbb{N}^0$ be a function that takes a place-transition pair and returns the cardinality for the appropriate arc s.t. $\mathit{min\text{-}cardinality}(\mathcal{A}(t,p)) \leq \mathit{perm}_t(t,p) \leq \mathit{max\text{-}cardinality}(\mathcal{A}(t,p))$ and similarly $\mathit{min\text{-}cardinality}(\mathcal{A}(p,t)) \leq \mathit{perm}_t(p,t) \leq \mathit{max\text{-}cardinality}(\mathcal{A}(p,t))$. The set of all valid permutations for a transition $t \in T$ is denoted by $\mathit{Permutations}_t$.

For a given *colored Petri net* (CPN) $\mathcal{N} = (P, T, CT, \mathbb{V}, C, \mathcal{A}, \mathcal{I}, G, M_0)$, our tool constructs the color ignorant P/T net $\mathcal{N}^i = (P^i, T^i, \mathcal{A}^i, \mathcal{I}^i, M_0^i)$ where

- $P^i = P$ and $T^i = \{t_{\mathit{perm}_t} \mid t \in T, \mathit{perm}_t \in \mathit{Permutations}_t\}$,
- $\mathcal{A}^i(t_{\mathit{perm}_t}, p) = \mathit{perm}_t(t,p)'\bullet$ and $\mathcal{A}^i(p, t_{\mathit{perm}_t}) = \mathit{perm}_t(p,t)'\bullet$,
- $\mathcal{I}^i(p, t_{\mathit{perm}_t}) = \mathcal{I}(p,t)$, and
- $M_0^i(p) = |M_0(p)|'\bullet$.

The construction forgets the concrete colors of tokens and accounts for all combinations of how the number of tokens can change. Hence the method preserves that for every marking M reachable in the CPN $\mathcal{N}$ there is a marking M' reachable in the color ignorant net $\mathcal{N}^i$ s.t. $|M(p)| = |M'(p)|$ for all $p \in P$.

This implies that if a cardinality formula is not reachable in the color ignorant net (we use here the fast state equation check using linear programming [10,23]) then it is not reachable in the original colored net either. In many cases this allows us to conclude on negative reachability queries in a fraction of time that is otherwise required by the explicit state-space exploration.

Simplification of fireability propositions. When evaluating fireability formulae that contain the enabledness check of a transition t in $\mathcal{N}$, we can replace t with $\bigvee_{\mathit{perm}_t \in \mathit{Permutations}_t} t_{\mathit{perm}_t}$ when exploring the color ignorant net. For performance reasons, our tool replaces t with a single transition $t_{\mathit{min}} \in \{t_{\mathit{perm}_t} \mid \mathit{perm}_t \in \mathit{Permutations}_t\}$ that minimizes $\mathit{perm}_t(p,t)$ and $\mathit{perm}_t(t,p)$ for all $p \in P$ and $t \in T$. Clearly, t_{min} is enabled in a marking if and only if $\bigvee_{\mathit{perm}_t \in \mathit{Permutations}_t} t_{\mathit{perm}_t}$ evaluates to true.

A fireability predicate in a P/T net can be encoded as a cardinality one (as a conjunction of the minimum number of required tokens in places that enable a given transition) and we can therefore apply the overapproximation technique to quickly establish that a transition cannot be enabled. However, this technique cannot be used to determine that a transition is enabled in some reachable marking of a CPN as this requires information on the specific colors. In our tool, we evaluate subexpressions of the fireability formula and use the color ignorant net for efficient reachability checks by using the state equations and linear programming before we perform a state-space exploration of the color ignorant net (which is cheaper than exploring the original CPN).

Let Ψ be a function, defined in Figure 2, that for a given fireability formula returns either the value $\top$ (reachable), $\bot$ (unreachable) or ? (inconclusive). We can see that if $\Psi(\gamma^f) = \top$ then γ^f is reachable in the original CPN and if $\Psi(\gamma^f) = \bot$ then γ^f is not reachable in the original CPN. In case of an inconclusive answer, the explicit state-space exploration is executed.

$$\Psi(t) = \begin{cases} \bot & \text{if } M \not\models t_{min} \text{ for all reachable markings } M \text{ in } \mathcal{N}^i \\ ? & \text{otherwise} \end{cases}$$

$$\Psi(\mathit{true}) = \top \qquad \Psi(\mathit{false}) = \bot$$

$$\Psi(\gamma_1^f \wedge \gamma_2^f) = \begin{cases} \top & \text{if } \Psi(\gamma_1^f) = \top \text{ and } \Psi(\gamma_2^f) = \top \\ \bot & \text{if } \Psi(\gamma_1^f) = \bot \text{ or } \Psi(\gamma_2^f) = \bot \\ ? & \text{otherwise} \end{cases} \qquad \Psi(\neg\gamma^f) = \begin{cases} \top & \text{if } \Psi(\gamma^f) = \bot \\ \bot & \text{if } \Psi(\gamma^f) = \top \\ ? & \text{if } \Psi(\gamma^f) = ? \end{cases}$$

Fig. 2: Definition of fireability query simplification function Ψ

4 Implementation and Experiments

The explicit verification engine is implemented in C++ as part of the open source project verifypn [14]. The engine supports four search strategies BFS, DFS, random DFS and a heuristic search, using the successor generator described in Section 3 as well as its variant that evenly cycles through all transitions in the net. Our benchmarking showed that the latter one together with random DFS is the best performing variant and is therefore used in our experiments.

As input, our engine accepts CPN descriptions in the standard Petri Net Markup Language (PNML) [5] and formula queries in XML syntax as used in MCC [1]. The reachability search algorithm compresses the passed list of already visited markings using the PTrie data structure [17].

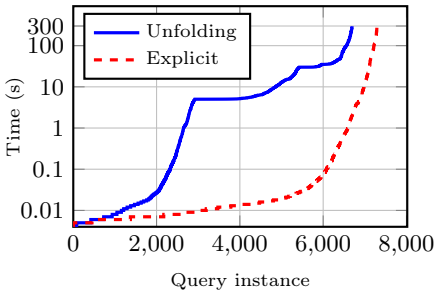
The tool can return traces that certify the reachability of a marking satisfying a given formula; in order to save memory the bindings in the trace are stored in a compressed format. TAPAAL GUI allows us to visualize the returned traces. The engine also implements a simulation mode with a communication protocol, allowing the user to simulate the behaviour of CPNs without unfolding the net. The source code of our explicit verification engine is available on GitHub [13].

Experiments. We benchmark the performance of our engine on all of the colored models from the Model Checking Contest (MCC) [19] 2024. There are 272 CPN models in the dataset, each with 16 cardinality and 16 fireability reachability queries, giving a total of 8 704 queries in each category. Each query is run on a single core of an AMD EPYC 9334 processor with clock speed 2.7 GHz, restricted to 16 GB of memory and a five minute timeout. We compare the performance against the unfolding approach implemented in TAPAAL, the MCC 2024 winner in the reachability category and currently the leading unfolding tool [3], using the best (default) parameters of the unfolding engine. Reproducibility package is available at [6].

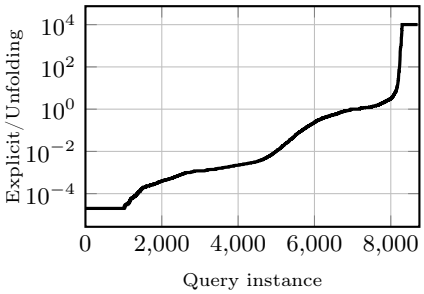
Figure 3a depicts the total number of answers that the unfolding and explicit engines solve within the time and memory constraints, including details about the cardinality and fireability subcategories and the distribution of positive queries (where a trace exists) and negative ones. Our explicit engine improves the number of answers in all columns. A more detailed insight is provided in the cactus plot in Figure 3b where the query instances (on the x-axis) are

Engine	Total answers			Cardinality			Fireability		
	All	+	−	All	+	−	All	+	−
Unfolding	6697	3557	3140	3745	1192	2553	2952	2365	587
Explicit	7291	4055	3236	4040	1418	2622	3251	2637	614

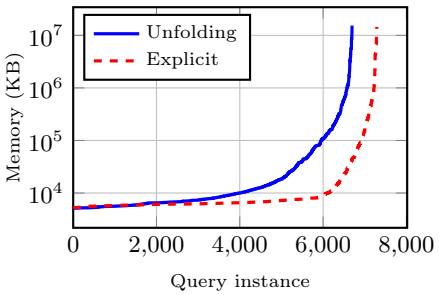
(a) Number of answered queries (+ for positive and − for negative answers)



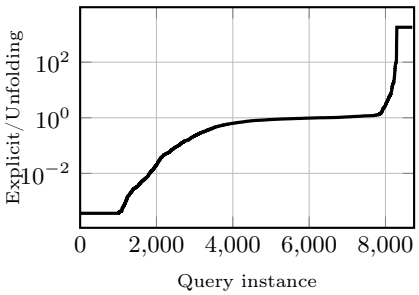
(b) Ordered verification times



(c) CPU time ratio Explicit/Unfolding



(d) Ordered peak memory



(e) Memory ratio Explicit/Unfolding

Fig. 3: Comparison of the explicit and unfolding approach

independently sorted by their running times (on y-axis). This shows large improvements in the running time—almost three times as many queries are solved within one second using the explicit approach compared to the unfolding. The sudden slope change in the unfolding curve is caused by a 5 second timeout of the color partitioning [4] technique in the unfolding approach. Similarly, Figure 3c offers query by query comparison of the performance, showing that on a large majority of queries, the explicit approach achieves several order of magnitude improvements. Similar conclusions can be drawn from Figures 3d and 3e regarding the peak memory consumption.

Finally, Table 1 depicts the comparison in the MCC setup (run locally on our cluster computer) with the 2024 competition script (running for 60 minutes) and its variant where we added 120 seconds of explicit state-space exploration using our engine before the unfolding approach (still finishing within 1 hour). It shows that we increased the number of solved queries from 82% to 91% and the

MCC Script	Answers	Cardinality	Fireability
MCC'24 competition script (without explicit)	7088 (82%)	4051 (93%)	3037 (70%)
With explicit engine running for 120 seconds	7862 (91%)	4267 (98%)	3595 (83%)

Table 1: Answered queries and the total percentage in the MCC setup

improvement is particularly pronounced in the fireability category (increment by 13 percentage points). These additional answers were an important factor for TAPAAL defending its first place also in MCC'25 [20].

Acknowledgments. We thank to Peter Gjøl Jensen his technical coding assistance.

References

1. Amat, N., Amparore, E., Berthomieu, B., Bouvier, P., Zilio, S.D., Hulin-Hubard, F., Jensen, P.G., Jezequel, L., Kordon, F., Li, S., Paviot-Adet, E., Petrucci, L., Srba, J., Thierry-Mieg, Y., Wolf, K.: Behind the scene of the model checking contest, analysis of results from 2018 to 2023. In: Beyer, D., Hartmanns, A., Kordon, F. (eds.) TOOLympics Challenge 2023. pp. 52–89. Springer Nature Switzerland, Cham (2025)
2. 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**, 2741–2756 (2004)
3. Bilgram, A., Jensen, P., Pedersen, T., Srba, J., Taankvist, P.: Methods for efficient unfolding of colored Petri nets. Fundamenta Informaticae **189**(3–4), 297–320 (2023)
4. Bilgram, A., Jensen, P.G., Pedersen, T., Srba, J., Taankvist, P.H.: Improvements in unfolding of colored Petri nets. In: Bell, P.C., Totzke, P., Potapov, I. (eds.) Reachability Problems. pp. 69–84. Springer International Publishing, Cham (2021)
5. Billington, J., Christensen, S., Van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri net markup language: concepts, technology, and tools. In: 24th International Conference on Applications and Theory of Petri Nets. p. 483–505. LNCS, Springer (2003)
6. Brandt, E.N., Højriis, J.E.F., Pedersen, K.S., Srba, J.: Reproducibility package for "Explicit Model Checking Engine for Reachability Analysis of Colored Petri Nets" (September 2025). https://doi.org/10.5281/zenodo.17077809
7. Chodak, J., Heiner, M.: Spike - Reproducible Simulation Experiments with Configuration File Branching. In: Computational Methods in Systems Biology. pp. 315–321. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-31304-3_19
8. Dal Zilio, S.: MCC: A Tool for Unfolding Colored Petri Nets in PNML Format. In: Application and Theory of Petri Nets and Concurrency. pp. 426–435. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51831-8_23
9. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K., Møller, M., Srba, J.: TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In: 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12). LNCS, vol. 7214, p. 492–497. Springer (2012)
10. Esparza, J., Melzer, S.: Verification of safety properties using integerprogramming: Beyond the state equation. Form. Methods Syst. Des. **16**(2), 159–189 (2000). https://doi.org/10.1023/A:1008743212620

11. Heiner, M., Herajy, M., Liu, F., Rohr, C., Schwarick, M.: Snoopy – A Unifying Petri Net Tool. In: Application and Theory of Petri Nets. pp. 398–407. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31131-4_22
12. Heiner, M., Rohr, C., Schwarick, M.: MARCIE - Model Checking and Reachability Analysis Done Efficiently. In: Application and Theory of Petri Nets and Concurrency. pp. 389–399. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_21
13. Højriis, J.E.F., Brandt, E.N., Stæhr Pedersen, K.: Github with source code of the explicit engine in verifypn (2025), https://github.com/TAPAAL/verifypn
14. Jensen, J., Nielsen, T., Østergaard, L., Srba, J.: TAPAAL and Reachability Analysis of P/T Nets, pp. 307–318. LNCS, Springer, Germany (2016). https://doi.org/10.1007/978-3-662-53401-4_16
15. Jensen, K.: Coloured Petri nets and the invariant-method. Theoretical Computer Science **14**(3), 317–336 (1981). https://doi.org/10.1016/0304-3975(81)90049-9
16. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**(3–4), 213–254 (Jun 2007)
17. Jensen, P., Larsen, K., Srba, J.: PTrie: Data structure for compressing and storing sets via prefix sharing. In: Proceedings of the 14th International Colloquium on Theoretical Aspects of Computing (ICTAC'17). LNCS, vol. 10580, pp. 248–265. Springer (2017). https://doi.org/10.1007/978-3-319-67729-3_15
18. Klostergaard, A.H.: Efficient Unfolding and Approximation of Colored Petri Nets with Inhibitor Arcs. Master's thesis, AAU (2018), https://projekter.aau.dk/projekter/files/281079031/main.pdf
19. Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Jezequel, L., Nivon, E.P.A.Q., , Amat., N., Berthomieu, B., Dal Zilio, S., Jensen, P., Morard, D., Smith, B., Srba, J., Thierry-Mieg, Y., Wolf, K.: Complete Results for the 2024 Edition of the Model Checking Contest. https://mcc.lip6.fr/2024/results.php (June 2024)
20. Kordon, F., Hulin-Hubard, F., Jezequel, L., Paviot-Adet, E., Nivon, Q., , Amat., N., Berthomieu, B., Dal Zilio, S., , Ding, Z., He, Y., Li, S., Jiang, C., Jensen, P., Srba, J., Thierry-Mieg, Y.: Complete Results for the 2025 Edition of the Model Checking Contest. https://mcc.lip6.fr/2025/results.php (June 2025)
21. Marechal, A.: Token ring CPN model, available at https://mcc.lip6.fr/2024/models.php
22. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Computer Aided Verification. pp. 164–177. Springer (1993)
23. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
24. Petri, C.A.: Kommunikation mit automaten (1962), https://api.semanticscholar.org/CorpusID:117254333
25. Thierry-Mieg, Y.: Symbolic model-checking using its-tools. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 231–237. Springer, Berlin, Heidelberg (2015)
26. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An extensible platform for state space analysis. In: Applications and Theory of Petri Nets. vol. 5606, pp. 303–312. Springer (2009)
27. Wolf, K.: Petri net model checking with LoLA 2. In: Application and Theory of Petri Nets and Concurrency. pp. 351–362. Springer, Cham (2018)