An Improved Construction of Petri Net Unfoldings

César Rodríguez and Stefan Schwoon

LSV, ENS Cachan & CNRS, INRIA Saclay
61, Av. du Président Wilson
94235 Cachan Cedex, France
cesar.rodriguez@lsv.ens-cachan.fr, stefan.schwoon@lsv.ens-cachan.fr

Abstract

Petri nets are a well-known model language for concurrent systems. The unfolding of a Petri net is an acyclic net bisimilar to the original one. Because it is acyclic, it admits simpler decision problems though it is in general larger than the net. In this paper, we revisit the problem of efficiently constructing an unfolding. We propose a new method that avoids computing the concurrency relation and therefore uses less memory than some other methods but still represents a good time-space tradeoff. We implemented the approach and report on experiments.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, F.1.1 Models of Computation, F.3.1 Specifying and Verifying and Reasoning about Program

Keywords and phrases Concurrency, Petri nets, partial orders, unfoldings

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.47

1 Introduction

Model checking is a practical way of ensuring the correctness of concurrent systems, but suffers from the problem of *state-space explosion* (SSE). One source of SSE is the explicit representation of concurrent actions by their interleavings. Petri nets are a model of concurrent systems, and their *unfoldings* are an established approach for coping with this source of SSE.

An unfolding can be thought as a partial order that compactly represents the state space of a Petri net. Roughly speaking, the unfolding of a net N is another *acyclic* net $\mathcal{U}_N$ that behaves like N. Actually, one is usually interested in a prefix $\mathcal{P}_N$ of $\mathcal{U}_N$ that represents all reachable markings of a bounded net N. An unfolding can be seen as a time/space tradeoff: problems such as coverability or deadlock checking are PSPACE-complete in N, but only NP-complete in $\mathcal{P}_N$. On the other hand, $\mathcal{P}_N$ is usually rather larger than N but often exponentially smaller than its reachability graph, and the aforementioned problems can easily be encoded into SAT. Also, the same prefix can answer multiple queries once constructed. See [2] for a survey on unfoldings. Tools like Mole [11] or Punf [6] efficiently construct unfoldings of safe nets.

Unfoldings are built iteratively. The central challenge of their construction is to identify the events of $\mathcal{U}_N$, which requires to find sets of concurrent conditions of $\mathcal{U}_N$. This is a computationally difficult problem (NP-complete), and several approaches to it have been proposed in the literature. In [3], the authors propose using a *concurrency relation*, i.e., determine for all pairs of conditions of $\mathcal{U}_N$ whether they are part of some reachable marking. This tends to be fast but memory-intensive. An orthogonal technique are *prefix trees* [7], which try to reduce the combinatorial overhead associated to the search. These techniques can be combined, for instance Punf implements them both.

In this paper, we propose an alternative to using concurrency relations for the case of safe nets. Our contribution is an efficient traversal of the unfolding that detects concurrent pairs

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 47–52

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

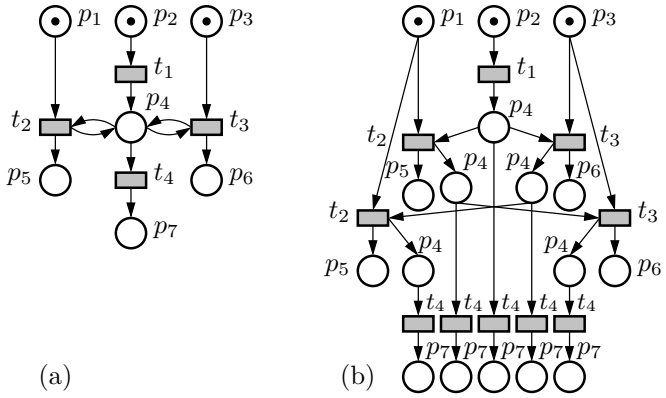

Figure 1 A Petri net N (a) and its unfolding $\mathcal{U}_N$ (b) and associated labelling.

of conditions 'on demand', without computing or storing the entire concurrency relation.

In Sec. 2, we formally introduce Petri nets and unfoldings. The algorithm that constitutes the main contribution of the paper is presented in Sec. 3. We implemented and tested the approach and report on benchmarks in Sec. 4 and conclude in Sec. 5.

2 Unfoldings of Petri Nets

A *Petri net*, or just *net*, is a tuple $N := \langle P, T, F, m_0 \rangle$, where P and T are the *places* and *transitions*, F is the *flow relation*, and $m_0 \colon P \to \mathbb{N}$ is the *initial marking*. Places and transitions together are called *nodes*. Fig. 1 (a) shows the usual graphical representation of a net with seven places and four transitions. The arrows depict the flow relation.

For any node x, let ${}^\bullet x := \{\, y \in P \cup T \colon (y, x) \in F \,\}$ be the *preset*, and $x^\bullet := \{\, y \in P \cup T \colon (x, y) \in F \,\}$ the *postset* of x. We lift these notions to sets of nodes in the expected way. A *marking* is a function $m \colon P \to \mathbb{N}$ that assigns *tokens* to every place. A transition t is *enabled* at m if $m(p) \geq 1$ for all $p \in {}^\bullet t$. Such t can *fire*, producing marking m', where $m'(p) = m(p) - |\{p\} \cap {}^\bullet t| + |\{p\} \cap t^\bullet|$. A sequence $\sigma = t_1 \dots t_n \in T^*$ is a *run* leading to marking m if t_1 is enabled at m_0, all t_i, $i \geq 2$, are enabled at the marking produced by t_{i-1}, and m is produced by t_n. A marking m is *reachable* if some run σ leads to it. N is *safe* if $m(p) \leq 1$ for all reachable m and $p \in P$. In this paper we only consider safe nets, and identify their markings with subsets of P. A set $X \subseteq P$ of places is *coverable* if $X \subseteq m$ for some reachable marking m.

The *unfolding* of N is a net $\mathcal{U}_N := \langle B, E, G, \tilde{m}_0 \rangle$ equipped with a *labelling* $h \colon (B \cup E) \to (P \cup T)$ that maps places and transitions of $\mathcal{U}_N$, called *conditions* and *events*, to places and transitions of N, respectively. When $h(x) = y$, we say that x is a "copy" of y or that x is y-labelled, and naturally extend h to sets and sequences. $\mathcal{U}_N$ and h are defined inductively:

$$\frac{p \in m_0}{c := \langle \bot, p \rangle \in B \quad h(c) := p \quad c \in \tilde{m}_0} \textsc{Ini} \qquad \frac{t \in T \quad X \subseteq B \quad h(X) = {}^\bullet t \quad X \text{ is coverable}}{e := \langle X, t \rangle \in E \quad {}^\bullet e := X \quad h(e) := t} \textsc{Ev}$$

$$\frac{e \in E \quad h(e) = t \quad t^\bullet = \{\, p_1, \dots, p_n \,\}}{c_i := \langle e, p_i \rangle \in B \quad e^\bullet := \{\, c_1, \dots, c_n \,\} \quad h(c_i) := p_i} \textsc{Cond}$$

Intuitively, $\mathcal{U}_N$ is an acyclic version of N: One starts with a "copy" of marking m_0, i.e. one condition $\langle \bot, p \rangle \in \tilde{m}_0$ for each $p \in m_0$ (see Ini). Then, whenever $\mathcal{U}_N$ can reach a marking $\tilde{m}$ such that $h(\tilde{m})$ enables t, we attach a copy of t to $\mathcal{U}_N$ (Ev). This copy, called $e = \langle X, t \rangle$ satisfies $X = {}^\bullet e$, $h(X) = {}^\bullet t$, and has 'fresh' copies of $t^\bullet$ in its postset (Cond). Thus, $\mathcal{U}_N$

is 'acyclic', and all conditions have at most one event in their preset. Fig. 1 (b) shows the unfolding, and associated labelling, of the net shown in Fig. 1 (a).

$\mathcal{U}_N$ has the same reachable markings and firing sequences as N, modulo h. In general, $\mathcal{U}_N$ is infinite, and applications usually compute a finite prefix $\mathcal{P}_N$ of it that is *complete* w.r.t. some application-dependent criterion, e.g., $\mathcal{P}_N$ is called *marking-complete* when for all reachable marking m in N there exists a reachable marking $\tilde{m}$ in $\mathcal{P}_N$ with $h(\tilde{m}) = m$. The details of such completeness criteria are beyond the scope of this exposition, see e.g. [4, 8].

For every pair of nodes x, y in $\mathcal{U}_N$ exactly one of three cases holds [4]:

- x and y are *causally related*, denoted $x < y$ (or $y < x$), if there is a path of flow arcs from x to y (resp. from y to x) in G. By construction, $<$ is an irreflexive partial order; if $x < y$, then x needs to occur before y in a finite firing sequence. We denote $\leq$ as reflexive closure of $<$. The *cone* of node x contains the causal predecessors of x, i.e., $[x] := \{\, y \colon y \leq x \,\}$.
- x and y are *in conflict*, written $x \# y$, if they compete for a token, i.e., $\#$ is the least symmetric relation on nodes satisfying (1) $e \# e'$ if $e, e' \in E$ with $e \neq e'$ and ${}^\bullet e \cap {}^\bullet e' \neq \emptyset$; and (2) $x \# z$ if there is $y \in B \cup E$ such that $x \# y$ and $y < z$. Intuitively, if $x \# y$, then no run fires or marks *both* x and y.
- x and y are called *concurrent*, written $x \parallel y$, if they are neither causally related nor in conflict. Thus, if x and y are conditions, then $\{\, x, y \,\}$ is coverable.

The principal algorithmic challenge to construct a prefix of $\mathcal{U}_N$ is to identify coverable sets of conditions X in applying the rule Ev. Given a prefix $\mathcal{P}_N$ of $\mathcal{U}_N$, it is NP-complete to decide whether $\mathcal{P}_N$ can be extended with another event [9]. The following approaches were proposed and implemented in tools:

- Since X is coverable iff $c \parallel c'$ for all $c, c' \in X$, it is promising to construct the *concurrency relation* $\parallel$ restricted to conditions. In [3], it is shown how $\parallel$ can be computed "on the fly" while constructing $\mathcal{P}_N$. This approach is implemented in the tool Mole.
- Eschewing the computation and storage of $\parallel$, [7] proposes several techniques to optimize the computations of relevant coverable sets using only memory linear in the size of $\mathcal{P}_N$; these techniques are implemented in Punf.

Experimentation over realistic benchmarks suggest that the first approach is usually faster but also more memory-intensive, in the worst-case quadratic in $|B|$; the second approach therefore succeeds to solve some big instances where the first runs out of memory. Punf actually allows to switch from the first to the second after the unfolding exceeds a given size.

3 The Algorithm

In this section, we describe the contribution of this paper: a new way of computing the events of $\mathcal{U}_N$. Like [7], this method does not employ the concurrency relation between conditions and uses only a constant amount of memory per condition and event, yet it is orthogonal to the tricks proposed in [7].

Before presenting the new contribution, we first describe a generic abstract algorithm for building $\mathcal{U}_N$, used for instance in [3, 7]. The algorithm maintains a set PE of so-called *possible extension*s, i.e., events that may be added by applying rule Ev to the prefix $\mathcal{P}_N$ generated so far. Its steps are:

1. Start with $\tilde{m}_0$, generated by the rule Ini, and fill PE with events $\langle X, t \rangle$ where $X \subseteq \tilde{m}_0$.
2. As long as PE is non-empty, remove an event e from PE. Let $\mathcal{P}'_N$ be the prefix obtained by adding e and its postset to $\mathcal{P}_N$, by means of rules Ev and Cond.
3. Identify and add to PE the set of possible extensions of $\mathcal{P}'_N$ that were not possible in $\mathcal{P}_N$. For any such extension $\langle X, t \rangle$, X necessarily intersects $e^\bullet$.
4. Set $\mathcal{P}_N := \mathcal{P}'_N$ and continue at step 2.

As mentioned in Sec. 2, this procedure may not terminate, and practical applications usually truncate $\mathcal{P}_N$ at certain *cutoff points*. This aspect is irrelevant to our contribution and we do not discuss it, focusing instead on step 3, the only difficult one. For the rest of this section, let $e := \langle X, t\rangle$ be the event in step 3. We proceed in two substeps:

3a. For each place $p \in {}^\bullet(t^{\bullet\bullet}) \setminus t^\bullet$, determine the set $C(p, e)$ of p-labelled conditions $\langle x, p\rangle$ that are concurrent with e. For $p \in t^\bullet$, we set $C(p, e) := \{\langle e, p\rangle\}$

3b. For all $t' \in t^{\bullet\bullet}$, use the sets $C(p, e)$ to discover new possible extensions, i.e., find coverable subsets X' with $h(X') = t'$ and add these to PE.

While step 3a can be implemented in time linear in $|\mathcal{P}_N|$, it is known that step 3b is NP-complete, even when the concurrency relation is given [5]. On the other hand, profiling on many benchmarks (for instance, using MOLE) suggests that step 3a is more expensive in practice than step 3b.

On the one hand, [3] proposes to compute sets $C(p, e)$ using the concurrency relation and discusses the on-the-fly computation and maintenance of the latter, giving little detail on step 3b. On the other hand, [7] presents heuristics for speeding up step 3b without discussing step 3a in detail. We shall discuss a method for implementing step 3a efficiently but without storing the concurrency relation and using only $\mathcal{O}(|\mathcal{P}_N|)$ memory. This method can be combined with the optimizations of step 3b from [7].

We start with a series of simple observations, which are valid for unfoldings of safe Petri nets. Let p be a place of N and $h^{-1}(p)$ the set of conditions labelled by p in $\mathcal{U}_N$. Since N is safe, no two elements of $h^{-1}(p)$ can be concurrent. Thus, the causality relation $<$, restricted to $h^{-1}(p)$, forms a forest where any pair of conditions c, c' that are not causally related are in conflict. Let us call this the *p-forest*. Now, let $c, c' \in h^{-1}(p)$ and e an event. We observe that (i) if $c \# e$ and $c < c'$ then $c' \# e$; (ii) if $c < e$ and $c' < c$ then $c' < e$; and in particular in both cases $c' \parallel e$ does not hold. Moreover, let $C' = h^{-1}(p) \cap [e]$ for some event e. Then no two elements of C' can be in conflict, and therefore (iii) C' must be totally ordered w.r.t. $<$.

Based on these observations, our algorithm for step 3a consists of the following steps:

I. We traverse the causal predecessors of e, i.e. the cone $[e]$. This serves two purposes:
- Mark all elements of $[e]$ with a special bit that allows to determine, in constant time, whether any given node of $\mathcal{P}_N$ belongs to $[e]$;
- Update the p-forest for all $p \in t^\bullet$: if $C' := h^{-1}(p) \cap [e]$ is empty, then the condition $\langle e, p\rangle \in e^\bullet$ is a root of the p-forest, otherwise it is a child of the maximal element of C', due to (iii).

The traversal takes linear time in $|[e]|$.[1]

II. Now, let $p \in {}^\bullet(t^{\bullet\bullet}) \setminus t^\bullet$. We determine $C(p, e) \subseteq h^{-1}(p)$ by traversing the p-forest in an order that respects $<$, starting at the roots of the forest. Let $c \in h^{-1}(p)$.
- if $c < e$ (constant-time check due to I.), then $c \notin C(p, e)$; however, some of its children in the p-forest may be, so we continue to explore those;
- if $c \# e$, then $c \notin C(p, e)$, and neither are any of its children in the p-forest, cf. (i).

To determine $c \# e$, we traverse the cone $[c]$ in reverse $<$-order. If we encounter an event $e' < e$, then no conflict can be detected by exploring e' or its causal precedessors, so we skip $[e']$. But if we encounter a condition $c' < e$ in the traversal, then we can conclude that $c \# e$ holds (because if $e' \in ([c] \cap c'^\bullet) \setminus [e]$ is the event that led us to c' in the traversal, then $c' \in {}^\bullet e' \cap {}^\bullet e''$ for some $e'' \leq e$). If we find no such c' in $[c]$, then $c \parallel e$ holds.

[1] Such a traversal is anyway necessary in most unfolding-based implementations to collect information relevant for determining which events are *cutoff points* [3, 8], so this step comes at almost no extra cost.

Table 1 Experimental results. Time and memory for PUNF and MOLE are *ratios*, see text.

Net	Unfolding		New Alg.		PUNF	MOLE	
Name	Events	Cond.	Time	Mem	Time (r)	Time (r)	Mem (r)
DPD(7)	10457	30248	0.34	9	6.59	1.76	2.18
FTP(1)	89046	178085	16.06	36	6.40	0.07	1.18
BYZ	14724	42276	0.73	21	11.48	2.66	3.15
Q(1)	7469	20969	0.21	9	6.81	2.14	2.04
ELEV(4)	16935	32354	0.50	9	5.06	0.24	1.08
BDS(1)	6330	12310	0.04	4	5.75	1.00	1.19
DME(6)	1830	6451	0.04	6	4.50	3.50	2.66
DME(7)	2737	9542	0.08	8	4.88	3.88	3.11
DME(8)	3896	13465	0.13	12	5.92	4.54	3.37
DME(9)	5337	18316	0.22	17	6.64	4.95	3.62
DME(10)	7090	24191	0.34	24	7.50	5.47	3.93
DME(11)	9185	31186	0.53	33	8.13	5.92	4.05
RW(1,2)	49179	147607	1.58	24	0.52	0.39	1.11
RW(3,1)	15401	28138	1.04	9	3.85	0.16	1.18
KEY(3)	6968	13941	0.23	5	2.52	0.30	1.03
KEY(4)	67954	135914	15.94	33	2.34	0.06	1.08
FURN(3)	25394	58897	0.69	13	3.48	1.01	1.61
FURN(4)	146606	342140	25.75	75	3.02	0.67	1.76
MMGT(3)	5841	11575	0.15	4	1.93	0.20	1.08
MMGT(4)	46902	92940	9.95	18	1.68	0.06	1.18

Notice that step II is repeated for different places p and conditions c. We make some further optimizations to avoid unnecessary repeated work during the computation for the same e:

- If we conclude that $c \parallel e$ holds for some c, we remember this information in the elements of $[c]$ (as a single bit), and any further conflict checks can safely skip $[c]$.
- If we conclude that $c \# e$ holds due to some condition $c' < e$ like above, then we propagate this information along the trail of nodes that led us from c to c'. Any further conflict checks that encounter one of those nodes can immediately stop and deduce a conflict, too.

4 Experiments

We experimentally compared our approach with other unfolding algorithms. MOLE computes a concurrency relation [3] and therefore uses quadratic memory in the worst-case. PUNF, when used with option `-n=0`, employs a linear-time exploration of $\mathcal{P}_N$ for step 3a [7]. Our implementation is based on MOLE, but we replaced MOLE's concurrency relation by our approach.

Tab. 1 summarizes our comparison on 20 classical benchmarks from the unfolding literature. For every net, we present the unfolding size together with the time (in seconds) and memory (in megabytes) of our approach, listed under 'New Alg.'. For PUNF and MOLE, the data is a *ratio relative to our approach*. Memory usage for PUNF could not readily be determined, but should be asymptotically the same as for our approach. The computed unfolding is obviously the same for the three approaches.

Quite positively, our approach runs faster than PUNF on all examples except one, with an overall running time 3.6 times smaller than that of PUNF. We interpret this as an encouraging result for our approach. Remark that PUNF implements an optimization called *prefix trees* in step 3b, which is still missing in our implementation. This technique is actually orthogonal to our contributions and seems to be particularly effective for RW(1,2). We therefore expect that our running times could be further improved in some cases by implementing prefix trees.

Also positively, our implementation consumes in average 48% of the memory that MOLE uses and still runs faster than it on roughly one half of the cases. These cases notably include all the instances of the DME series, where we obtain improved running times of up to 6 times. Here, the cost of computing the concurrency relation is actually larger than its benefit.

However, our approach is still overall slower than MOLE. Our accumulated running time is 2.3 times larger. The worst case seems to be MMGT(4), where MOLE runs 17 times faster using roughly the same memory. The concurrency relation proves to be very effective for this net, in average a condition is concurrent to only 0.2‰ of the other conditions. Compare this ratio with that of DME(11), where our approach performs 6 times faster. There, the aforementioned average is 38‰, making MOLE's approach inefficient. The same analysis holds for KEY(4), where MOLE is 16 times faster than our approach, and where the concurrency relation is even comparatively smaller than in MMGT(4).

Overall, the new approach seems to present a practical tradeoff in terms of time. For better comparison, we show only examples in which all tools terminated. However, it was already pointed out in [7] that there exist cases where the concurrency relation becomes too big to fit into memory, and approaches like PUNF and ours succeed where MOLE does not.

5 Conclusions

We presented an algorithmic improvement for the construction of Petri net unfoldings. While our implementation is still preliminary, the experimental results are promising; its running time beats the one of [7] (which also uses a linear amount of memory), and it represents an acceptable time/memory trade-off compared with [3], which uses more memory; in some instances it even performs faster.

For future work, it would be interesting to improve the implementation to incorporate the tricks from [7], which should further improve the running time. Moreover, we are interested in generalizing the approach to nets with read arcs, where it could be used as an alternative to [1] within the tool CUNF [10].

References

1 Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König, César Rodríguez, and Stefan Schwoon. Efficient unfolding of contextual Petri nets. *TCS*, 449:2–22, 2012.

2 Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

3 Javier Esparza and Stefan Römer. An unfolding algorithm for synchronous products of transition systems. In *Proc. CONCUR*, LNCS 1664, pages 2–20, 1999.

4 Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

5 Keijo Heljanko. *Deadlock and reachability checking with finite complete prefixes*. Licentiate's thesis, Helsinki University of Technology, 1999.

6 Victor Khomenko. PUNF. `homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/`.

7 Victor Khomenko and Maciej Koutny. Towards an efficient algorithm for unfolding Petri nets. In *Proc. CONCUR*, LNCS 2154, pages 366–380, 2001.

8 Victor Khomenko, Maciej Koutny, and Walter Vogler. Canonical prefixes of Petri net unfoldings. *Acta Informatica*, 40(2):95–118, 2003.

9 Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. CAV*, LNCS 663, pages 164–177, 1992.

10 César Rodríguez. CUNF. `http://www.lsv.ens-cachan.fr/~rodriguez/tools/cunf/`.

11 Stefan Schwoon. MOLE. `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`.