# Towards An Efficient Algorithm for Unfolding Petri Nets

Victor Khomenko and Maciej Koutny

Department of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.
{Victor.Khomenko, Maciej.Koutny}@ncl.ac.uk

**Abstract.** Model checking based on the causal partial order semantics of Petri nets is an approach widely applied to cope with the state space explosion problem. One of the ways to exploit such a semantics is to consider (finite prefixes of) net unfoldings, which contain enough information to reason about the reachable markings of the original Petri nets. In this paper, we propose several improvements to the existing algorithms for generating finite complete prefixes of net unfoldings. Experimental results demonstrate that one can achieve significant speedups when transition presets of a net being unfolded have overlapping parts.
**Keywords:** Model checking, Petri nets, unfolding, concurrency.

## 1 Introduction

A distinctive characteristic of reactive concurrent systems is that their sets of local states have descriptions which are both short and manageable, and the complexity of their behaviour comes from highly complicated interactions with the external environment rather than from complicated data structures and manipulations thereon. One way of coping with this complexity problem is to use formal methods and, especially, computer aided verification tools implementing model checking [2] — a technique in which the verification of a system is carried out using a finite representation of its state space. The main drawback of model checking is that it suffers from the state space explosion problem. That is, even a relatively small system specification can (and often does) yield a very large state space. To help in coping with this, a number of techniques have been proposed, which can roughly be classified as aiming at an implicit compact representation of the full state space of a reactive concurrent system, or at an explicit generation of its reduced (though sufficient for a given verification task) representation. Techniques aimed at reduced representation of state spaces are typically based on the independence (commutativity) of some actions, often relying on the partial order view of concurrent computation. Such a view is the basis for algorithms employing McMillan's (finite prefixes of) Petri net unfoldings ([7, 14]), where the entire state space of a system is represented implicitly, using an acyclic net to represent relevant system's actions and local states.

In view of the development of fast model checking algorithms employing unfoldings ([10–12]), the problem of efficiently building them is becoming increasingly important. Recently, [6–8] addressed this issue — considerably improving

the original McMillan's technique — but we feel that the problem of generating net unfoldings deserves further investigation. Though there are negative theoretical results concerning this problem ([5, 10]), in practice unfoldings can often be built quite efficiently. [7] stated that the slowest part of their unfolding algorithm was building possible extensions of the branching process being constructed (the decision version of this problem is NP-complete, see [10]). To compute them, [6] suggests to keep the concurrency relation and provides a method of maintaining it. This approach is fast for simple systems, but soon deteriorates as the amount of memory needed to store the concurrency relation may be quadratic in the number of conditions in the already built part of the unfolding.

In this paper, we propose another method of computing possible extensions and, although it is compatible with the concurrency relation approach, we decided to abandon this data structure in order to be able to construct larger prefixes. We show how to find new transition instances to be inserted in the unfolding, not by trying the transitions one-by-one, but several at once, merging the common parts of the work. Moreover, we provide some additional heuristics. Experimental results demonstrate that one can achieve significant speedups if the transitions of a safe Petri net being unfolded have overlapping parts. All missing proofs can be found in [13].

## 2 Basic Notions

A *net* is a triple $N \stackrel{\text{df}}{=} (P, T, F)$ such that $P$ and $T$ are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. A *marking* of $N$ is a multiset $M$ of places, i.e. $M : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. We adopt the standard rules about drawing nets, viz. places are represented as circles, transitions as boxes, the flow relation by arcs, and markings are shown by placing tokens within circles. As usual, we will denote ${}^\bullet z \stackrel{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \stackrel{\text{df}}{=} \{y \mid (z, y) \in F\}$, for all $z \in P \cup T$, and ${}^\bullet Z \stackrel{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \stackrel{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. We will assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$.

A *net system* is a pair $\Sigma \stackrel{\text{df}}{=} (N, M_0)$ comprising a finite net $N = (P, T, F)$ and an *initial* marking $M_0$. A transition $t \in T$ is *enabled* at a marking $M$ if for every $p \in {}^\bullet t$, $M(p) \geq 1$. Such a transition can be *executed*, leading to a marking $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$. We denote this by $M[t\rangle M'$. The set of *reachable* markings of $\Sigma$ is the smallest (w.r.t. set inclusion) set $[M_0\rangle$ containing $M_0$ and such that if $M \in [M_0\rangle$ and $M[t\rangle M'$ (for some $t \in T$) then $M' \in [M_0\rangle$. $\Sigma$ is *safe* if for every reachable marking $M$, $M(P) \subseteq \{0, 1\}$; and *bounded* if there is $k \in \mathbb{N}$ such that $M(P) \subseteq \{0, \ldots, k\}$, for every reachable marking $M$. Unless stated otherwise, we will assume that a net system $\Sigma$ to be unfolded is safe, and use PreMax to denote the maximal size of transition preset in $\Sigma$.

**Branching processes** Two nodes (places or transitions), $y$ and $y'$, of a net $N = (P, T, F)$ are *in conflict*, denoted by $y \# y'$, if there are distinct transitions $t, t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and $(t, y)$ and $(t', y')$ are in the reflexive transitive closure of the flow relation $F$, denoted by $\preceq$. A node $y$ is in *self-conflict* if $y \# y$.

An *occurrence net* is a net $ON \stackrel{\text{df}}{=} (B, E, G)$ where $B$ is a set of *conditions* (places) and $E$ is a set of *events* (transitions). It is assumed that: $ON$ is acyclic (i.e. $\preceq$ is a partial order); for every $b \in B$, $|^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y \# y)$ and there are finitely many $y'$ such that $y' \prec y$, where $\prec$ denotes the irreflexive transitive closure of $G$. $Min(ON)$ will denote the set of minimal elements of $B \cup E$ with respect to $\preceq$. The relation $\prec$ is the *causality relation*. Two nodes are *concurrent*, denoted $y \ co\ y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$. We also denote by $x \ co\ C$, where $C$ is a set of pairwise concurrent nodes, the fact that a node $x$ is concurrent to each node from $C$. Two events $e$ and $f$ are *separated* if there is an event $g$ such that $e \prec g \prec f$.

A *homomorphism* from an occurrence net $ON$ to a net system $\Sigma$ is a mapping $h : B \cup E \to P \cup T$ such that: $h(B) \subseteq P$ and $h(E) \subseteq T$; for all $e \in E$, the restriction of $h$ to $^\bullet e$ is a bijection between $^\bullet e$ and $^\bullet h(e)$; the restriction of $h$ to $e^\bullet$ is a bijection between $e^\bullet$ and $h(e)^\bullet$; the restriction of $h$ to $Min(ON)$ is a bijection between $Min(ON)$ and $M_0$; and for all $e, f \in E$, if $^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$. If $h(x) = y$ then we will often refer to $x$ as *$y$-labelled*.

A *branching process* of $\Sigma$ ([4]) is a quadruple $\pi \stackrel{\text{df}}{=} (B, E, G, h)$ such that $(B, E, G)$ is an occurrence net and $h$ is a homomorphism from $ON$ to $\Sigma$. A branching process $\pi' = (B', E', G', h')$ of $\Sigma$ is a *prefix* of a branching process $\pi = (B, E, G, h)$, denoted by $\pi' \sqsubseteq \pi$, if $(B', E', G')$ is a subnet of $(B, E, G)$ such that: if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and $h'$ is the restriction of $h$ to $B' \cup E'$. For each $\Sigma$ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of $\Sigma$.

A *configuration* of an occurrence net $ON$ is a set of events $C$ such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$. The configuration $[e] \stackrel{\text{df}}{=} \{f \mid f \preceq e\}$ is called the *local configuration* of $e \in E$. A set of conditions $B'$ such that for all distinct $b, b' \in B'$, $b \ co\ b'$, is called a *co-set*. A *cut* is a maximal (w.r.t. set inclusion) co-set. Every marking reachable from $Min(ON)$ is a cut.

Let $C$ be a finite configuration of a branching process $\pi$. Then $Cut(C) \stackrel{\text{df}}{=} (Min(ON) \cup C^\bullet) \setminus {}^\bullet C$ is a cut; moreover, the multiset of places $Mark(C) \stackrel{\text{df}}{=} h(Cut(C))$ is a reachable marking of $\Sigma$. A marking $M$ of $\Sigma$ is *represented* in $\pi$ if the latter contains a finite configuration $C$ such that $M = Mark(C)$. Every marking represented in $\pi$ is reachable, and every reachable marking is represented in the unfolding of $\Sigma$.

A branching process $\pi$ of $\Sigma$ is *complete* if for every reachable marking $M$ of $\Sigma$: (i) $M$ is represented in $\pi$; and (ii) for every transition $t$ enabled by $M$, there is a finite configuration $C$ and an event $e \notin C$ in $\pi$ such that $M = Mark(C)$, $h(e) = t$ and $C \cup \{e\}$ is a configuration. Although, in general, the unfolding of a finite bounded net system $\Sigma$ may be infinite, it is possible to truncate it and obtain a finite complete prefix, $Unf_\Sigma$. [15] proposes a technique for this, based on choosing an appropriate set $E_{cut}$ of *cut-off* events, beyond which the unfolding is not generated. One can show ([7, 9]) that it suffices to designate an event $e$ newly added during the construction of $Unf_\Sigma$ as a cut-off event, if the already built part of the prefix contains a *corresponding* configuration $C$

**input** : $\Sigma = (N, M_0)$ — a bounded net system
**output** : $\mathit{Unf}_\Sigma$ — a finite and complete prefix of $\Sigma$'s unfolding

$\mathit{Unf}_\Sigma \leftarrow$ the empty branching process
add instances of the places from $M_0$ to $\mathit{Unf}_\Sigma$
$pe \leftarrow \textsc{PotExt}(\mathit{Unf}_\Sigma)$
$\mathit{cut_off} \leftarrow \emptyset$
**while** $pe \neq \emptyset$ **do**
  **choose** $e \in pe$ **such that** $[e] \in \min_\lhd\{[f] \mid f \in pe\}$
  **if** $[e] \cap \mathit{cut_off} = \emptyset$
  **then**
    add $e$ and new instances of the places from $h(e)^\bullet$ to $\mathit{Unf}_\Sigma$
    $pe \leftarrow \textsc{PotExt}(\mathit{Unf}_\Sigma)$
    **if** $e$ is a cut-off event of $\mathit{Unf}_\Sigma$ **then** $\mathit{cut_off} \leftarrow \mathit{cut_off} \cup \{e\}$
  **else** $pe \leftarrow pe \setminus \{e\}$

**Fig. 1.** The unfolding algorithm presented in [7].

without cut-off events, such that $\mathit{Mark}(C) = \mathit{Mark}([e])$ and $C \lhd [e]$, where $\lhd$ is an *adequate order* on the finite configurations of a branching process (see [7] for the definition of $\lhd$).

The unfolding algorithm presented in [7, 8] is parameterised by an adequate order $\lhd$, and can be formulated as in figure 1. It is assumed that the function PotExt finds the set of possible extensions of the already constructed part of a prefix, which can be defined in the following way (see [7]).

**Definition 1.** *Let $\pi$ be a branching process of a net system $\Sigma$. A* possible extension *of $\pi$ is a pair $(t, D)$, where $D$ is a co-set in $\pi$ and $t$ is a transition of $\Sigma$, such that $h(D) = {}^\bullet t$ and $\pi$ contains no $t$-labelled event with the preset $D$.*

For simplicity, in figure 1 and later in this paper, we do not distinguish between a possible extension $(t, D)$ and a (virtual) $t$-labelled event $e$ with the preset $D$, provided that this does not create an ambiguity.

The efficiency of the algorithm in figure 1 heavily depends on a good adequate order $\lhd$, allowing early detection of cut-off events. It is advantageous to choose 'dense' (ideally, total) orders. [7, 8] propose such an order for safe net systems, and show that if a total order is used, then the number of the non-cut-off events in the resulting prefix will never exceed the number of reachable markings in the original net system (though usually it is much smaller). Using a total order allows one to simplify some parts of the unfolding algorithm in figure 1, e.g., testing whether an event is a cut-off event can be reduced to a single look-up in a hash table if only local corresponding configurations are allowed (using non-local ones can be very time consuming, see [9]).

## 3 Finding possible extensions

Almost all the steps of the unfolding algorithm in figure 1 can be implemented quite efficiently. The only hard part is to calculate the set of possible extensions, PotExt($Unf_\Sigma$), and we will make it the focus of our attention. As the decision version of the problem is NP-complete in the size of the already built part of the prefix ([10]), it is unlikely that we can achieve substantial improvements in the worst case for a single call to the PotExt procedure. However, the following approaches can still be attempted: (i) using heuristics to reduce the cost of a single call; and (ii) merging the common parts of the work performed to insert individual instances of transitions. An excellent example of a method aimed at reducing the amount of work is the improvement, proposed in [7], where a total order on configurations is used to reduce both the size of the constructed complete prefix and the number of calls to PotExt. Another method is outlined in [6, 15], where the algorithm does not have to recompute all the possible extensions in each step: it suffices to update the set of possible extensions left from the previous call, by adding events consuming conditions from $e^\bullet$, where $e$ is the last inserted event.

**Definition 2.** *Let $\pi$ be a branching process of a net system $\Sigma$, and $e$ be one of its events. A possible extension $(t, D)$ of $\pi$ is a $(\pi, e)$*-extension *if $e^\bullet \cap D \neq \emptyset$, and $e$ and $(t, D)$ are not separated.*

With this approach, the set $pe$ in the algorithm in figure 1 can be seen as a priority queue (with the events ordered according to the adequate order $\lhd$ on their local configurations) and implemented using, e.g., a binary heap. The call to PotExt($Unf_\Sigma$) in the body of the main loop of the algorithm is replaced by UpdatePotExt($pe$, $Unf_\Sigma$, $e$), which finds all $(\pi, e)$-extensions and inserts them into the queue. Note that in the important special case of binary synchronisation, when the size of transition preset is at most 2, say ${}^\bullet t = \{h(c), p\}$ and $c \in e^\bullet$, the problem becomes equivalent to finding the set $\{c' \in h^{-1}(p) \mid c' \ co\ c\}$, which can be efficiently computed (the problem is vacuous when $|{}^\bullet t| = 1$). This technique leads to a further simplification since now we never compute any possible extension more then once, and so we do not have to add the cut-off events (and their postsets) into the the prefix being built until the very end of the algorithm. Hence, we can altogether avoid checking whether a configuration contains a cut-off event.

We now observe that in definition 2, $e$ and $(t, D)$ are not separated events, which basically suggests that any sufficient condition for being a pair of separated events may help in reducing the computational cost involved in calculating the set of $(\pi, e)$-extensions. In what follows, we identify two such cases.

In the pseudo-code given in [15], the conditions $c \in e^\bullet$ are inserted into the unfolding one by one, and the algorithm tries to insert new instances of transitions from $h(c)^\bullet$ with $c$ in their presets. Such an approach can be improved as the algorithm is sub-optimal in the case when a transition $t$ can consume more then one condition from $e^\bullet$. Indeed, $t$ is considered for insertion after each condition from $e^\bullet$ it can consume has been added, and this may lead to a significant

overhead when the size of $t$'s preset is large. Therefore, it is better to insert into the unfolding the whole post-set $e^\bullet$ at once, and use the following simple result, which essentially means that possible extensions being added consume as many conditions from $e^\bullet$ as possible (note that this results in an improvement whenever there is a $(\pi, e)$-extension, which can consume more than one condition produced by $e$).

**Proposition 1.** *Let $e$ and $f$ be events in the unfolding of a safe net system such that $f \in (e^\bullet)^\bullet$ and $h(e^\bullet \cap {}^\bullet f) \neq h(e)^\bullet \cap {}^\bullet h(f)$. Then $e$ and $f$ are separated.*

**Corollary 1.** *Let $\pi$ be a branching process of a safe net system, $e$ be an event of $\pi$, and $(t, D)$ be a $(\pi, e)$-extension. Then $|e^\bullet \cap D| = |h(e)^\bullet \cap {}^\bullet t|$.*

Another way of reducing the number of calls to PotExt is to ignore some of the transitions from $(u^\bullet)^\bullet$, which the algorithm attempts to insert after a $u$-labelled event $e$. For in a safe net system, if the preset ${}^\bullet t$ of a transition $t \in (u^\bullet)^\bullet$ has non-empty intersection with ${}^\bullet u \setminus u^\bullet$, then $t$ cannot be executed immediately after $u$. Therefore, in the unfolding procedure, an instance $f$ of $t$ cannot be inserted immediately after a $u$-labelled event $e$ (though $f$ *may* actually consume conditions produced by $e$, as shown in figure 2; note that in such a case $e$ and $f$ are separated).

**Proposition 2.** *Let $e$ and $f$ be events in the unfolding of a safe net system such that $f \in (e^\bullet)^\bullet$ and $({}^\bullet h(e) \setminus h(e)^\bullet) \cap {}^\bullet h(f) \neq \emptyset$. Then $e$ and $f$ are separated.*

**Corollary 2.** *Let $\pi$ be a branching process of a safe net system, $e$ be an event of $\pi$, and $(t, D)$ be a $(\pi, e)$-extension. Then $({}^\bullet h(e) \setminus h(e)^\bullet) \cap {}^\bullet t = \emptyset$.*

In view of the above corollary, the algorithm may consider only transitions from the set $(h(e)^\bullet)^\bullet \setminus ({}^\bullet h(e) \setminus h(e)^\bullet)^\bullet$ rather than $(h(e)^\bullet)^\bullet$ as the candidates for insertion after $e$.

The resulting algorithm for updating the set of possible extensions after inserting an event $e$ into the unfolding is fairly straightforward ([13]); moreover, it is possible not to maintain the concurrency relation, as suggested in [6], by rather to mark conditions which are not concurrent to a constructed part of a transition preset as unusable, and to unmark them during the backtracking.

**Merging computation common to several calls** The presets of candidate transitions for inserting after an event $e$ often have overlapping parts besides the places from $h(e)^\bullet$, and the algorithm may be looking for instances of the same

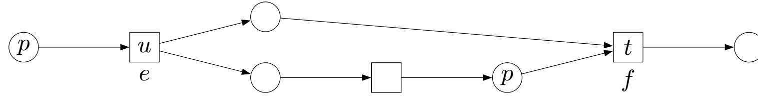


**Fig. 2.** A $t$-labelled event $f$ cannot be inserted immediately after a $u$-labelled event $e$ if $p \in ({}^\bullet u \setminus u^\bullet) \cap {}^\bullet t \neq \emptyset$, even though it can consume a condition produced by $e$.

places in the unfolding several times. To avoid this, one may identify the common parts of the presets, and treat them only once. The main idea is illustrated below.

Let $e$ be the last event inserted into the prefix being built and $h(e)^\bullet = \{p\}$. Moreover, let $t_1, t_2, t_3$ and $t_4$ be possible candidates for inserting after $e$ such that ${}^\bullet t_1 = \{p, p_1, p_2, p_3, p_4\}$, ${}^\bullet t_2 = \{p, p_1, p_2, p_3\}$, ${}^\bullet t_3 = \{p, p_1, p_2, p_3, p_5\}$, and ${}^\bullet t_4 = \{p, p_2, p_3, p_4, p_5\}$. The condition labelled by $p$ in each case comes from the postset of $e$. To insert $t_i$, the algorithm has to find a co-set $C_i$ such that $e\ co\ C_i$ and $h(C_i) = {}^\bullet t_i \setminus \{p\}$ (if there are several such co-sets, then several instances of $t_i$ should be inserted). By gluing the common parts of the presets, one can obtain a tree shown in figure 3(a), which can then be used to reduce the task of finding the co-sets $C_i$. Formally, we proceed as follows.

**Definition 3.** *Let $u$ be a transition of a net system $\Sigma$ and $U = (u^\bullet)^\bullet \setminus ({}^\bullet u \setminus u^\bullet)^\bullet$. A* preset tree *of $u$, $PT_u$, is a directed tree satisfying the following:*

- *Each vertex is labelled by a set of places, so that the root is labelled by $\emptyset$, and the sets labelling the nodes of any directed path are pairwise disjoint.*
- *Each transition $t \in U$ has an associated vertex $v$, such that the union of all the place sets along the path from the root to $v$ is equal to ${}^\bullet t \setminus u^\bullet$ (different transitions may have the same associated vertex).*
- *Each leaf is associated to at least one transition (unless the tree consists of one vertex only).*

*The* weight *of $PT_u$ is defined as the sum of the weights of all the nodes, where the* weight *of a node is the cardinality of the set of places labelling it.*

Having built a preset tree, we can use the algorithm in figure 4 to update the set of possible extensions, aiming at avoiding redundant work (sometimes there are gains even when PreMax $= 2$ and no two transitions have the same preset, see [13]). Note that we only need one preset tree $PT_u$ per transition $u$ of the net system, and it can be built during the preprocessing stage.

**Building preset trees** Two problems which we now address are: (i) how to evaluate the 'quality' of preset trees, and (ii) how to efficiently construct them. If we use the 'totally non-optimised' preset tree shown in figure 3(b) instead of that in figure 3(a) as an input to the algorithm in figure 4, it will work in a way very similar to that of the standard algorithm trying the candidate transitions one-by-one. However, gluing the common parts of the presets decreases both the weight of the preset tree and the number of times the algorithm attempts to find new conditions concurrent to the already constructed part of event presets. This suggests that preset trees with small weight should be preferred. Such a 'minimal weight' criterion may be seen as rather rough, since it is hard to predict during the preprocessing stage which preset tree will be better, as different ones might be better for different instances of the same transition. Another problem is that the reduction of the weight of a preset tree leads to the creation of new vertices and splitting of the sets of places among them, effectively decreasing the weight of a single node. This may reduce the efficiency of the heuristics,

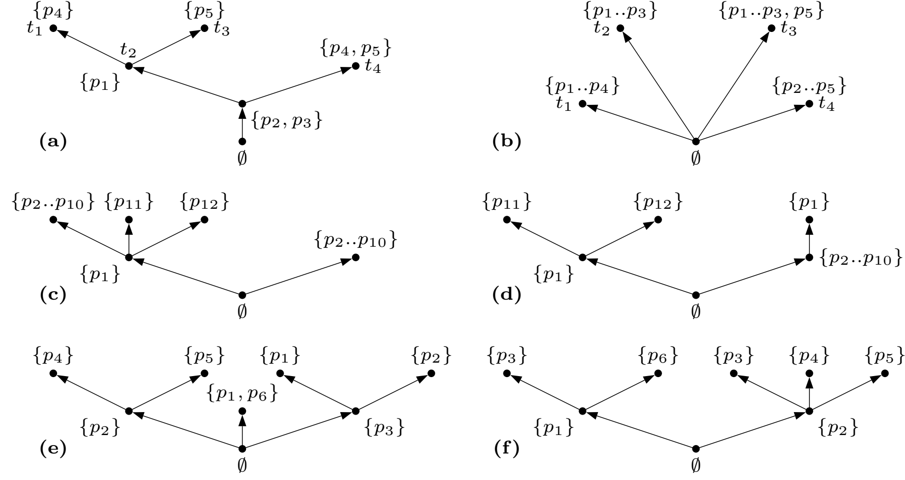


**Fig. 3.** An optimised **(a)** and non-optimised **(b)** preset trees of weight 7 and 15; **(c)** a tree of weight 21, produced by the bottom-up algorithm with $A_1 = \{p_1, \ldots, p_{10}\}$, $A_2 = \{p_2, \ldots, p_{10}\}$, $A_3 = \{p_1, p_{11}\}$, and $A_4 = \{p_1, p_{12}\}$ ($p_1$ was chosen on the first iteration of the algorithm), and **(d)** a tree of weight 13, corresponding to the same sets; **(e)** a tree of weight 8, produced by the top-down algorithm for the sets $A_1 = \{p_1, p_3\}$, $A_2 = \{p_1, p_6\}$, $A_3 = \{p_2, p_3\}$, $A_4 = \{p_2, p_4\}$, and $A_5 = \{p_2, p_5\}$ (the intersection $\{p_3\} = A_1 \cap A_3$ was chosen on the first iteration), and **(f)** a tree of weight 7, corresponding to the same sets.

which potentially might be used for finding co-sets in the algorithm in figure 4. But this drawback is usually more than compensated for by the speedup gained by merging the common parts of the work spent on finding co-sets forming the presets of newly inserted events.

Since there may exist a whole family of minimal-weight preset trees for the same transition, one could improve the criterion by taking into account the remark about heuristics for resolving the non-deterministic choice, and prefer minimal weight preset trees which also have the minimal number of nodes. Furthermore, we could assign coefficients to the vertices, depending on the distance from the root, the cardinality of the labelling sets of places, etc., and devise more complex optimality criterion. However, this may get too complicated and the process of building preset trees can easily become more time consuming then the unfolding itself. And, even if a very complicated criterion is used, the time spent on building a highly optimised preset tree can be wasted: the transition may be dead, and the corresponding preset tree will never be used by the unfolding algorithm. Therefore, in the actual implementation, we decided to adopt the simple 'minimal weight' criterion and, in the view of the next result, it was justifiable to implement a relatively fast greedy algorithm aiming at 'acceptably light' preset trees.

**procedure** UPDATEPOTEXT($pe$, $Unf_{\Sigma}$, $e$)
  $tree \leftarrow$ preset tree for $h(e)$ /* pre-calculated */
  $C \leftarrow$ all conditions concurrent to $e$
  COVER($C$, $tree$, $e$, $\emptyset$)

**procedure** COVER($C$, $tree$, $e$, $preset$)
  **for all** transitions $t$ labelling the root of $tree$ **do**
    $pe \leftarrow pe \cup \{(t, (e^{\bullet} \cap h^{-1}({}^{\bullet}t)) \cup preset)\}$
  **for all** sons $tree'$ of $tree$ **do**
    $R \leftarrow$ places labelling the root of $tree'$
    **for all** co-sets $CO \subseteq C$ **such that** $h(CO) = R$ **do**
      COVER($\{c \in C \mid c \ co \ CO\}$, $tree'$, $e$, $preset \cup CO$)

**Fig. 4.** An algorithm for updating the set of possible extensions.

**Proposition 3.** *Building a minimal-weight preset tree is an NP-complete problem in the size of a Petri net, even if* PREMAX $= 3$.

*Proof.* The decision version of this problem is to determine whether there exists a preset tree of the weight at most $w$, where $w$ is given. It is in NP as the size of a preset tree is polynomial in the size of a Petri net, and we can guess it and check its weight in polynomial time.

The proof of NP-hardness is by reduction from the vertex cover problem. Given an undirected graph $G = (V, E)$, construct a Petri net as follows: take $V \cup \{p\}$ as the set of places, and for each edge $\{v_1, v_2\} \in V$ take a transition with $\{p, v_1, v_2\}$ as its preset. Moreover, take another transition $t$ with the postset $\{p\}$ (note that all the other transitions belong to $(t^{\bullet})^{\bullet}$). There is a bijection between minimal weight preset trees for $t$ and minimal size vertex covers for $G$. Therefore, the problem of deciding whether there is a preset tree of at most a given size is NP-hard. □

In figure 5, we outlined simple bottom-up and top-bottom algorithms for constructing 'light' preset trees. In each case, the input is a set of sets of places $\{A_1, \ldots, A_k\} = \{{}^{\bullet}t \setminus u^{\bullet} \mid t \in U\} \cup \{\emptyset\}$ and, as it is obvious how to assign vertices to the transitions, we omit this part. $Tree(v, \{Tr_1, \ldots, Tr_l\})$ is a tree with the root $v$ and the sons $Tr_1, \ldots, Tr_l$, which are also trees, and '$\cdot$' stands for a set of son trees if their identities are irrelevant.

The two algorithms do not necessarily give an optimal solution, but in most cases the results are acceptable. We implemented both, to check which approach performs better. The tests indicated that in most cases the resulting trees had the same weight, but sometimes a bad choice on the first step(s) causes the bottom-up approach to yield very poor results, as illustrated in figure 3(c,d). The top-down algorithm appeared to be more stable, and only in rare cases (see figure 3(e,f)) produced 'heavier' trees then the bottom-up one. Therefore, we will focus our attention on its efficient implementation.

A sketch of a possible implementation of the top-down algorithm for building preset trees is shown in figure 6. It computes all pairwise intersections of the sets

```
function BuildTree(S = {A_1, ..., A_k})     /* bottom − up */
    root ← ⋂_{A∈S} A
    S ← {A_1 \ root, ..., A_k \ root}
    TS ← ∅
    while ⋃_{A∈S} A ≠ ∅ do  /* while there are non-empty sets */
        choose p ∈ ⋃_{A∈S} A such that |{A ∈ S | p ∈ A}| is maximal
        Tree(v, ts) ← BuildTree({A \ {p} | A ∈ S ∧ p ∈ A})
        TS ← TS ∪ {Tree(v ∪ {p}, ts)}
        S ← {A ∈ S | p ∉ A}
    return Tree(root, TS)

function BuildTree({A_1, ..., A_k})     /* top − down */
    TS ← {Tree(A_1, ∅), ..., Tree(A_k, ∅)}
    while |TS| > 1 do
        choose Tree(A′, ·) ∈ TS and Tree(A″, ·) in TS
                such that A′ ≠ A″ and |A′ ∩ A″| is maximal
        I ← A′ ∩ A″
        T_⊂ ← {Tree(B \ I, ts) | Tree(B, ts) ∈ TS ∧ I ⊂ B}
        T_= ← ⋃{ts | Tree(I, ts) ∈ TS ∧ ts ≠ ∅}
        TS ← TS \ {Tree(B, ·) ∈ TS | I ⊆ B}
        TS ← TS ∪ {Tree(I, T_⊂ ∪ T_=)}
    /* |TS| = 1 */
    return the remaining tree Tr ∈ TS
```

**Fig. 5.** Two algorithms for building trees.

$A_i$ before the main loop starts, and then maintain this data structure. On each step, the algorithm chooses a set $I$ of maximal cardinality from *Intersec*, and updates the variables *TS* and *Intersec* in the following way: (i) it finds all the supersets of $I$ in *TS*, and removes them; (ii) it removes from *Intersec* all the intersections corresponding to these sets; (iii) the intersections of $I$ with the sets remaining in *TS* are added into *Intersec*; and (iv) $I$ is inserted into *TS*.

**Proposition 4.** *The worst case time complexity of the top-down algorithm is* $O(\text{PreMax}\cdot k^2 \cdot \log k)$. *It is also possible to implement it so that the average case complexity is given by* $O(\text{PreMax}\cdot k^2)$ *(see [13] for implementation details).*

It is essential for the correctness of the algorithm that *Intersec* is a multiset, and we have to handle duplicates in our data structure. It is better to implement this by maintaining a counter for each set inserted into *Intersec*, rather then by keeping several copies of the same set, since the multiplicity of simple sets (e.g., singletons or the empty set) can by very high. Moreover, if multiplicities are calculated, we often can reduce the weights of produced trees. The idea is to choose in figure 6 among the sets with maximal cardinality those which have the maximal number of supersets in *TS* (note that this would improve the tree in figure 3(e), forcing $\{p_2\}$ to be chosen on the first iteration). Such sets have the highest multiplicity among the sets with the maximal cardinality. Indeed,

**function** BUILDTREE($\{A_1, ..., A_k\}$)
  $TS \leftarrow \{Tree(A_1, \emptyset), \ldots, Tree(A_k, \emptyset)\}$
  /* *Intersec* is a multiset of sets */
  $Intersec \leftarrow \{A' \cap A'' \mid A' \neq A'' \wedge Tree(A', \cdot) \in TS \wedge Tree(A'', \cdot) \in TS\}$
  **while** $|TS| > 1$ **do**
    **choose** $I \in Intersec$ **such that** $|I|$ is maximal
    $T_{\subset} \leftarrow \{Tree(B \setminus I, ts) \mid Tree(B, ts) \in TS \wedge I \subset B\}$
    $T_{=} \leftarrow \bigcup\{ts \mid Tree(I, ts) \in TS \wedge ts \neq \emptyset\}$
    **for all** $Tree(A, ts) \in TS$ **such that** $I \subseteq A$ **do**
      $TS \leftarrow TS \setminus \{Tree(A, ts)\}$
      **for all** $Tree(B, \cdot) \in TS$ **do**
        $Intersec \leftarrow Intersec \setminus \{A \cap B\}$
    **for all** $Tree(A, \cdot) \in TS$ **do**
      $Intersec \leftarrow Intersec \cup \{I \cap A\}$
    $TS \leftarrow TS \cup \{Tree(I, T_{\subset} \cup T_{=})\}$
  /* $|TS| = 1$ */
  **return** the remaining tree $Tr \in TS$

**Fig. 6.** A top-down algorithm for building preset trees.

each time this choice is made by the algorithm, the values of *TS* and *Intersec* are 'synchronised' in the sense that *Intersec* contains all pairwise intersections of the sets marking the roots of the trees from *TS*, with the proper multiplicities. Now, let $I \in Intersec$ be a set with the maximal cardinality, which has $n$ supersets in *TS* (note that $n \geq 2$). The intersection of two sets can be equal to $I$ only if they both are supersets of $I$. Moreover, since there is no set in *Intersec* with cardinality greater then $|I|$, the intersections of any two distinct supersets of $I$ from *TS* is exactly $I$. Hence the multiplicity of $I$ is $C_n^2 = n(n-1)/2$. This function is strictly monotonic for all positive $n$, and so there is a monotonic one-to-one correspondence between the multiplicities of sets with the maximal cardinality from *Intersec* and the numbers of their supersets in *TS*. Thus, among the sets of maximal cardinality, those having the maximal multiplicity have the maximal number of supersets in *TS*. One can implement this improvement without affecting the asymptotic running time given by proposition 4 ([13]).

## 4 Experimental results

The results of our experiments are summarised in tables 1 and 2, where we use *time* to indicate that the test had not stopped after 15 hours, and *mem* to indicate that the test terminated because of memory overflow. They were measured on a PC with $Pentium^{TM}$ III/500MHz processor and 128M RAM. For comparison, the `ERVunfold 4.5.1` tool, available from the Internet, was used. The methods implemented in it are described in [6, 7]; in particular, it maintains the concurrency relation.

The meaning of the columns in the tables is as follows (from left to right): the name of the problem; the number of places and transitions, and the aver-

age/maximal size of transition presets in the original net system; the number of conditions, events and cut-off events in the complete prefix; the time spent by the `ERVunfold` tool (in seconds); the time spent by our algorithm on building the preset trees and unfolding the net; the ratio $W_{rat} = W_{opt}/W$, where $W_{opt}$ is the sum of the weights of the constructed preset trees, and $W$ is the sum of the weights of the 'totally non-optimised' preset trees as in figure 3(b). This ratio may be used as a rough approximation of the effect of employing preset trees: $W_{rat} = 1$ means that there is no optimisation. Note that when transition presets are large enough, employing preset trees gives certain gains, even if this ratio is close to 1 (see, e.g., the DME($n$) series).

We attempted (table 1) the popular set of benchmark examples, collected by J.C. Corbett ([3]), K. McMillan, S. Melzer, S. Römer (this set was also used in [6, 9, 10, 12, 16]), and available from K. Heljanko's homepage.

The transitions in these examples usually have small sizes of presets (in fact, they do not exceed 2 for most of the examples in table 1; the only example in this set with a big maximal preset is BYZ(4,1), but it in fact has only one transition with the preset of size 30, and one transition with the the preset of size 13; the sizes of the other transition presets in this net do not exceed 5). Thus, the advantage of using preset trees is not substantial, and `ERVunfold` is usually faster as it maintains the concurrency relation. But when the size of this relation becomes greater then the amount of the available memory, `ERVunfold` slows down because of page swapping (e.g., in FTP(1), GASNQ(5), and KEY(4) examples). As for our algorithm, it is usually slower for these examples, but its running time is acceptable. Moreover, sometimes it scales better (e.g., for the DME($n$), ELEV($n$) and MMGT($n$) series).

In order to test the algorithms on nets with larger presets, we have built a set of examples RND($m, n$) in the following way. First, we created $m$ loops consisting of $n$ places and $n$ transitions each; the first place of each loop was marked with one token. Then 500 additional transitions were added to this skeleton, so that each of them takes a token from a randomly chosen place in each loop and puts it back on another randomly chosen place in the same loop (thus, the net has $m{\cdot}n$ transitions with presets of size 1 and 500 transitions with presets of size $m$). It is easy to see that the nets built in this way are safe. The experimental results are shown in table 2.

To test a practical example with large transition presets, we looked at a data intensive application (where processes being modelled compute functions depending on many variables), namely the priority arbiter circuit described in [1]. We generated two series of examples: SPA($n$) for $n$ processes and linear priorities, and SPA($m, n$) for $m$ groups and $n$ processes in each group. The results are summarised in table 2. Our algorithm scales better and is able to produce much larger unfoldings. We expect that other areas, where Petri nets with large presets are needed, will be identified (such nets may result from net transformations, e.g. adding complementary places or converting bounded nets into safe ones, see [8]). But even for nets with small transition presets our algorithm is quite quick, and may be used if the size of the finite prefix is expected to be large (note

that it was slower than `ERVunfold` for some of the examples because we did not maintain the concurrency relation, trading speed for a possibility of building large prefixes — in principle, maintaining concurrency relation is compatible with all the described heuristics).

**Future work** We plan to develop an effective parallel algorithm for constructing large unfoldings. Another promising direction is to consider non-local correspondent configurations proposed in [9].

**Acknowledgements** Proposition 3 and its proof are due to P. Rossmanith. We would like to thank A. Bystrov for his suggestion to consider dual-rail logics circuits and help with modelling priority arbiters. We would also like to thank J. Esparza, K. Heljanko, and the anonymous referees for helpful comments. The first author was supported by an ORS Awards Scheme grant ORS/C20/4 and by an EPSRC grant GR/M99293.

## References

1. A. Bystrov, D. J. Kinniment and A. Yakovlev: Priority Arbiters. Proc. *ASYNC 2000*, IEEE Computer Society Press (2000) 128–137.
2. E. M. Clarke, E. A. Emerson and A. P. Sistla: Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS* 8 (1986) 244–263.
3. J. C. Corbett: *Evaluating Deadlock Detection Methods*. Univ. of Hawaii at Manoa (1994).
4. J. Engelfriet: Branching processes of Petri Nets. *Acta Inf.* 28 (1991) 575–591.
5. J. Esparza: Decidability and Complexity of Petri Net Problems — An Introduction. *Lectures on Petri Nets I: Basic Models* Springer, LNCS 1491 (1998) 374–428.
6. J. Esparza and S. Römer: An Unfolding Algorithm for Synchronous Products of Transition Systems. Proc. *CONCUR'99*, Springer, LNCS 1664 (1999) 2–20.
7. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. *TACAS'96*, Springer, LNCS 1055 (1996) 87–106.
8. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* (2001) to appear.
9. K. Heljanko: Minimizing Finite Complete Prefixes. Proc. *CS&P'99* (1999) 83–95.
10. K. Heljanko: Deadlock and Reachability Checking with Finite Complete Prefixes. Report A56, Laboratory for Theoretical Computer Science, HUT, Espoo (1999).
11. K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fund. Inf.* 37 (1999) 247–268.
12. V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. CS-TR-711, Dept. of Computing Science, Univ. of Newcastle (2000).
13. V. Khomenko and M. Koutny: An Efficient Algorithm for Unfolding Petri Nets. CS-TR-726, Dept. of Computing Science, Univ. of Newcastle (2001).
14. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. *CAV'92*, Springer, LNCS 663 (1992) 164–174.
15. K. L. McMillan: *Symbolic Model Checking*. PhD thesis, CMU-CS-92-131 (1992).
16. S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. *CAV'97*, Springer, LNCS 1254 (1997) 352–363.

| Problem | Net | | | Unfolding | | | Time, [s] | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\|S\|$ | $\|T\|$ | $a/m\ \|^{\bullet}t\|$ | $\|B\|$ | $\|E\|$ | $\|E_{cut}\|$ | *ERV* | *p-trees* | *Unf* | $W_{rat}$ |
| BDS(1) | 53 | 59 | 1.88/2 | 12310 | 6330 | 3701 | 1.30 | <0.01 | 3.87 | 0.53 |
| BYZ(1,4) | 504 | 409 | 3.33/30 | 42276 | 14724 | 752 | 126 | 0.14 | 231 | 0.71 |
| FTP(1) | 176 | 529 | 1.98/2 | 178085 | 89046 | 35197 | *time* | 0.16 | 2625 | 0.52 |
| Q(1) | 163 | 194 | 1.89/2 | 16123 | 8417 | 1188 | 8.69 | 0.03 | 39.43 | 0.81 |
| DME(7) | 470 | 343 | 3.24/5 | 9542 | 2737 | 49 | 6.37 | 0.19 | 7.28 | 0.93 |
| DME(8) | 537 | 392 | 3.24/5 | 13465 | 3896 | 64 | 14.12 | 0.09 | 16.08 | 0.92 |
| DME(9) | 604 | 441 | 3.24/5 | 18316 | 5337 | 81 | 27.78 | 0.11 | 31.82 | 0.92 |
| DME(10) | 671 | 490 | 3.24/5 | 24191 | 7090 | 100 | 51.67 | 0.13 | 58.14 | 0.92 |
| DME(11) | 738 | 539 | 3.24/5 | 31186 | 9185 | 121 | 89.18 | 0.16 | 98.96 | 0.92 |
| DPD(4) | 36 | 36 | 1.83/2 | 594 | 296 | 81 | 0.01 | <0.01 | 0.02 | 0.71 |
| DPD(5) | 45 | 45 | 1.82/2 | 1582 | 790 | 211 | 0.04 | <0.01 | 0.16 | 0.71 |
| DPD(6) | 54 | 54 | 1.81/2 | 3786 | 1892 | 499 | 0.22 | <0.01 | 0.83 | 0.71 |
| DPD(7) | 63 | 63 | 1.81/2 | 8630 | 4314 | 1129 | 1.16 | <0.01 | 5.49 | 0.71 |
| DPFM(5) | 27 | 41 | 1.98/2 | 67 | 31 | 20 | 0.00 | 0.01 | <0.01 | 1.00 |
| DPFM(8) | 87 | 321 | 2/2 | 426 | 209 | 162 | 0.01 | 0.08 | 0.01 | 1.00 |
| DPFM(11) | 1047 | 5633 | 2/2 | 2433 | 1211 | 1012 | 0.05 | 89.35 | 0.74 | 1.00 |
| DPH(5) | 48 | 67 | 1.97/2 | 2712 | 1351 | 547 | 0.10 | <0.01 | 0.36 | 1.00 |
| DPH(6) | 57 | 92 | 1.98/2 | 14590 | 7289 | 3407 | 2.16 | <0.01 | 9.74 | 1.00 |
| DPH(7) | 66 | 121 | 1.98/2 | 74558 | 37272 | 19207 | 57.43 | 0.01 | 263 | 1.00 |
| ELEV(2) | 146 | 299 | 1.95/2 | 1562 | 827 | 331 | 0.02 | 0.13 | 0.14 | 0.60 |
| ELEV(3) | 327 | 783 | 1.97/2 | 7398 | 3895 | 1629 | 0.61 | 1.59 | 2.73 | 0.60 |
| ELEV(4) | 736 | 1939 | 1.99/2 | 32354 | 16935 | 7337 | 16.15 | 25.57 | 68.43 | 0.61 |
| FURN(1) | 27 | 37 | 1.65/2 | 535 | 326 | 189 | 0.01 | <0.01 | 0.02 | 0.50 |
| FURN(2) | 40 | 65 | 1.71/2 | 4573 | 2767 | 1750 | 0.19 | <0.01 | 0.54 | 0.44 |
| FURN(3) | 53 | 99 | 1.75/2 | 30820 | 18563 | 12207 | 8.18 | <0.01 | 29.10 | 0.41 |
| GASNQ(3) | 143 | 223 | 1.97/2 | 2409 | 1205 | 401 | 0.09 | 0.03 | 0.36 | 0.96 |
| GASNQ(4) | 258 | 465 | 1.98/2 | 15928 | 7965 | 2876 | 4.54 | 0.10 | 18.45 | 0.97 |
| GASNQ(5) | 428 | 841 | 1.99/2 | 100527 | 50265 | 18751 | 785 | 0.32 | 817 | 0.98 |
| GASQ(2) | 78 | 97 | 1.95/2 | 346 | 173 | 54 | <0.01 | <0.01 | 0.02 | 0.93 |
| GASQ(3) | 284 | 475 | 1.99/2 | 2593 | 1297 | 490 | 0.11 | 0.12 | 0.40 | 0.97 |
| GASQ(4) | 1428 | 2705 | 2/2 | 19864 | 9933 | 4060 | 7.93 | 7.91 | 29.70 | 0.99 |
| KEY(2) | 94 | 92 | 1.97/2 | 1310 | 653 | 199 | 0.06 | 0.01 | 0.15 | 0.93 |
| KEY(3) | 129 | 133 | 1.98/2 | 13941 | 6968 | 2911 | 2.51 | 0.03 | 10.48 | 0.94 |
| KEY(4) | 164 | 174 | 1.98/2 | 135914 | 67954 | 32049 | 6247 | 0.06 | 864 | 0.94 |
| MMGT(2) | 86 | 114 | 1.95/2 | 1280 | 645 | 260 | 0.03 | 0.03 | 0.08 | 0.64 |
| MMGT(3) | 122 | 172 | 1.95/2 | 11575 | 5841 | 2529 | 1.75 | 0.07 | 6.09 | 0.64 |
| MMGT(4) | 158 | 232 | 1.95/2 | 92940 | 46902 | 20957 | 188 | 0.14 | 504 | 0.64 |
| RW(6) | 33 | 85 | 1.99/2 | 806 | 397 | 327 | 0.01 | 0.01 | 0.01 | 1.00 |
| RW(9) | 48 | 181 | 1.99/2 | 9272 | 4627 | 4106 | 0.21 | 0.03 | 0.34 | 1.00 |
| RW(12) | 63 | 313 | 2/2 | 98378 | 49177 | 45069 | 14.46 | 0.10 | 15.30 | 1.00 |
| SYNC(2) | 72 | 88 | 1.89/3 | 3884 | 2091 | 474 | 0.29 | <0.01 | 1.38 | 0.91 |
| SYNC(3) | 106 | 270 | 2.21/4 | 28138 | 15401 | 5210 | 14.15 | 0.06 | 74.84 | 0.77 |

**Table 1.** Experimental results: nets with small transition presets.

| Problem | Net | | | Unfolding | | | Time, [s] | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\|S\|$ | $\|T\|$ | $a/m\ \|^\bullet t\|$ | $\|B\|$ | $\|E\|$ | $\|E_{cut}\|$ | *ERV* | *p-trees* | *Unf* | $W_{rat}$ |
| Rnd(5,5) | 25 | 525 | 4.81/5 | 55698 | 14029 | 11689 | 11.45 | 7.36 | 3.66 | 0.39 |
| Rnd(5,6) | 30 | 530 | 4.77/5 | 84451 | 21774 | 17269 | 31.43 | 8.68 | 12.21 | 0.44 |
| Rnd(5,7) | 35 | 535 | 4.74/5 | 144700 | 36019 | 28922 | 82.92 | 8.90 | 30.69 | 0.50 |
| Rnd(5,8) | 40 | 540 | 4.70/5 | 235600 | 56691 | 46559 | 196 | 8.79 | 62.96 | 0.54 |
| Rnd(5,9) | 45 | 545 | 4.67/5 | 304656 | 72895 | 59840 | 324 | 7.43 | 105 | 0.58 |
| Rnd(5,10) | 50 | 550 | 4.64/5 | 419946 | 98477 | 82279 | 554 | 9.07 | 160 | 0.60 |
| Rnd(5,11) | 55 | 555 | 4.60/5 | 573697 | 132344 | 112310 | 994 | 6.20 | 246 | 0.63 |
| Rnd(5,12) | 60 | 560 | 4.57/5 | 627303 | 145378 | 122465 | 1187 | 5.72 | 322 | 0.65 |
| Rnd(5,13) | 65 | 565 | 4.54/5 | 718762 | 166093 | 140147 | 1560 | 5.27 | 420 | 0.67 |
| Rnd(5,14) | 70 | 570 | 4.51/5 | 802907 | 185094 | 156417 | 1952 | 5.58 | 507 | 0.69 |
| Rnd(5,15) | 75 | 575 | 4.48/5 | 842181 | 195228 | 163722 | 6685 | 6.63 | 616 | 0.70 |
| Rnd(5,16) | 80 | 580 | 4.45/5 | 886158 | 206265 | 171957 | *time* | 7.10 | 717 | 0.71 |
| Rnd(5,17) | 85 | 585 | 4.42/5 | 987605 | 229284 | 191576 | — | 3.78 | 863 | 0.72 |
| Rnd(5,18) | 90 | 590 | 4.39/5 | 1025166 | 239069 | 198524 | — | 5.62 | 998 | 0.73 |
| Rnd(10,2) | 20 | 520 | 9.65/10 | 34884 | 7136 | 6125 | 12.46 | 7.34 | 1.14 | 0.25 |
| Rnd(10,3) | 30 | 530 | 9.49/10 | 1415681 | 153628 | 144548 | 1638 | 3.90 | 82 | 0.49 |
| Rnd(10,4) | 40 | 540 | 9.33/10 | 2344821 | 252320 | 237000 | *mem* | 3.51 | 207 | 0.59 |
| Rnd(10,5) | 50 | 550 | 9.18/10 | 2485903 | 271083 | 250600 | — | 7.90 | 331 | 0.64 |
| Rnd(10,6) | 60 | 560 | 9.04/10 | 2535070 | 280560 | 255010 | — | 11.32 | 485 | 0.67 |
| Rnd(10,7) | 70 | 570 | 8.89/10 | 2537646 | 285323 | 254767 | — | 11.91 | 663 | 0.70 |
| Rnd(10,8) | 80 | 580 | 8.76/10 | 2534970 | 289550 | 254000 | — | 14.84 | 872 | 0.72 |
| Rnd(15,2) | 30 | 530 | 14.21/15 | 1836868 | 135307 | 128358 | *mem* | 32.28 | 70.24 | 0.37 |
| Rnd(15,3) | 45 | 545 | 13.84/15 | 3750719 | 271074 | 255560 | — | 14.69 | 259 | 0.57 |
| Rnd(15,4) | 60 | 560 | 13.50/15 | 3787575 | 280560 | 257515 | — | 7.54 | 456 | 0.67 |
| Rnd(15,5) | 75 | 575 | 13.17/15 | 3795090 | 288075 | 257515 | — | 6.38 | 718 | 0.73 |
| Rnd(20,2) | 40 | 540 | 18.59/20 | 4744587 | 256197 | 245750 | *mem* | 46.71 | 176 | 0.43 |
| Rnd(20,3) | 60 | 560 | 17.96/20 | 5040080 | 280560 | 260020 | — | 16.36 | 427 | 0.61 |
| Rnd(20,4) | 80 | 580 | 17.38/20 | 5050100 | 290580 | 260020 | — | 9.03 | 771 | 0.71 |
| Spa(4) | 98 | 81 | 2.77/5 | 1048 | 421 | 96 | 0.04 | 0.01 | 0.07 | 0.72 |
| Spa(5) | 121 | 113 | 3.34/6 | 3594 | 1362 | 457 | 0.26 | 0.03 | 0.53 | 0.63 |
| Spa(6) | 144 | 161 | 4.20/7 | 13334 | 4860 | 2145 | 3.79 | 0.08 | 5.51 | 0.56 |
| Spa(7) | 167 | 241 | 5.38/8 | 52516 | 18712 | 9937 | 64.22 | 0.28 | 75.54 | 0.49 |
| Spa(8) | 190 | 385 | 6.82/9 | 216772 | 76181 | 45774 | *time* | 1.26 | 943 | 0.43 |
| Spa(9) | 213 | 657 | 8.35/10 | 920270 | 320582 | 209449 | — | 6.66 | 12571 | 0.38 |
| Spa(2,1) | 52 | 37 | 2.16/4 | 111 | 52 | 4 | <0.01 | <0.01 | <0.01 | 0.87 |
| Spa(2,2) | 98 | 81 | 2.77/5 | 1206 | 476 | 110 | 0.04 | 0.01 | 0.10 | 0.72 |
| Spa(2,3) | 144 | 161 | 4.20/7 | 15690 | 5682 | 2512 | 5.53 | 0.08 | 8.28 | 0.56 |
| Spa(2,4) | 190 | 385 | 6.82/9 | 253219 | 88944 | 52826 | *time* | 1.29 | 1326 | 0.43 |
| Spa(3,1) | 75 | 57 | 2.40/4 | 324 | 141 | 19 | 0.01 | <0.01 | 0.02 | 0.79 |
| Spa(3,2) | 144 | 161 | 4.20/7 | 15690 | 5682 | 2512 | 5.49 | 0.08 | 9.09 | 0.56 |
| Spa(3,3) | 213 | 657 | 8.35/10 | 1142214 | 398850 | 256600 | *time* | 6.67 | 20594 | 0.38 |
| Spa(4,1) | 98 | 81 | 2.77/5 | 1048 | 421 | 96 | 0.04 | 0.01 | 0.09 | 0.72 |
| Spa(4,2) | 190 | 385 | 6.82/9 | 253219 | 88944 | 52826 | *time* | 1.27 | 1326 | 0.43 |

**Table 2.** Experimental results: nets with larger transition presets.