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 — themselves a class of acyclic Petri nets — which contain enough information, albeit implicit, to reason about the reachable markings of the original Petri nets.
In view of recent very efficient model checking algorithms, employing unfoldings ([10, 11]), the problem of efficiently building them becomes of great interest. [6–8] address this issue, considerably improving the original McMillan's technique, but the unfolding algorithms proposed there are still relatively slow.
In this paper, we put forward a method of computing possible extensions, which was the slowest part of the unfolding algorithm. Essentially, we show how to find new transition instances to be inserted in the unfolding, not trying all the transition one-by-one, but all at once, merging common parts of the work. Moreover, we propose some ways of reducing the number of candidate transitions to be inserted, and provide some additional heuristics, helping to speed up the algorithm.
Experimental results demonstrate that the resulting algorithms can achieve significant speedups if the transitions of the Petri net being unfolded have large presets.
Keywords: Model checking, Petri nets, unfolding, causality, 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, 12]), where the entire state space of a system is represented implicitly, using an acyclic net to represent system's actions and local states.

In view of recent very fast model checking algorithms, employing unfoldings ([11, 10]), the problem of efficiently building them becomes of great interest. [7, 6, 8] address this issue, considerably improving the original McMillan's technique, but the unfolding algorithms proposed there are still relatively slow.

Though there are strong negative results concerning this problem ([10, 5]), in practice there are many cases, when unfoldings can be built quite efficiently. [7] mentioned, that the slowest

part of their unfolding algorithm was building possible extensions of the unfolding being constructed (this is, in fact, an NP-complete problem, see [10]), but the question how actually to efficiently compute them was left open. [6] suggests to keep concurrency relation and provides a method of effectively maintaining it. This approach is quick for simple systems, but soon becomes intractable (the amount of memory to store this relation might be quadratical in the number of conditions in the already built part of the unfolding).

In this paper, we put forward another method of computing possible extensions. It is compatible with the concurrency relation, but we decided to abandon this data structure to be able to construct larger unfoldings. Essentially, we show how to find new transition instances to be inserted in the unfolding, not trying all the transition one-by-one, but all at once, merging common parts of the work. Moreover, we provide some additional heuristics, helping to speed up the algorithm.

Our experiments demonstrate that the resulting algorithms can achieve significant speedups if the transitions of the Petri net being unfolded have large presets.

2 Basic Notions

In this section, we first present basic definitions concerning Petri nets, and then recall (see also [4, 7, 8]) notions related to net unfoldings.

Petri nets A *net* is a triple $N \stackrel{\text{df}}{=} (S, T, F)$ such that S and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (S \times T) \cup (T \times S)$ is a *flow relation*. A *marking* of N is a multiset M of places, i.e. $M : S \to \mathbb{N} = \{0, 1, 2, \ldots\}$. 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 $z \in S \cup T$, 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 $Z \subseteq S \cup T$. 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 $N = (S, T, F)$ and an *initial* marking M_0. A transition $t \in T$ is *enabled* at a marking M if for every $s \in {}^\bullet t$, $M(s) \geq 1$. 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 Σ 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 $M' \in [M_0\rangle$.

A net system Σ is *safe* if for every reachable marking M, $M(S) \subseteq \{0, 1\}$; and *bounded* if there is $k \in \mathbb{N}$ such that $M(S) \subseteq \{0, \ldots, k\}$, for every reachable marking M.

Unless stated otherwise, we will assume in this paper that net systems to be unfolded are safe. However, we expect that several ideas and results can be extended to the unfolding of bounded net, e.g., by following the approach proposed in [8]. When evaluating the complexity of various algorithms, PreMax $\stackrel{\text{df}}{=} \max_{t \in T} |{}^\bullet t|$ will stand for the maximal size of transition's preset in Σ.

Branching processes Two nodes (places or transitions), y and y', of a net $N = (S, T, F)$ 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 the set of *conditions* (places) and E is the 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 Σ is a mapping $h : B \cup E \to S \cup T$ such that: $h(B) \subseteq S$ 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 Σ ([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 Σ. A branching process $\pi' = (B', E', G', h')$ of Σ 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 Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of Σ.

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

A branching process π of Σ is *complete* if for every reachable marking M of Σ, there is a configuration C in π such that $Mark(C) = M$, and for every transition t enabled by M, there is an event $e \notin C$ such that $h(e) = t$ and $C \cup \{e\}$ is a configuration.

ERV unfolding algorithm Although, in general, the unfolding of a finite bounded net system Σ may be infinite, it is possible to truncate it and obtain a finite complete prefix, Unf_Σ. [13] proposes a technique for this, based on choosing an appropriate set E_{cut} of *cut-off* events, and effectively generating the branching process comprising the events in $\bigcup_{e \in E_{cut}} [e]$. One can show ([7, 10]) that it suffices to designate an event e newly added during the construction of Unf_Σ as a cut-off event, if the already built part of a prefix contains a *corresponding* configuration C without cut-off events, such that $Mark([e]) = Mark(C)$ and $C \lhd [e]$, where $\lhd$ is an *adequate order* (see [7] for the definition) on the finite configurations of a branching process.

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

Definition 1. *Let π be a branching process of a net system Σ. A* possible extension *of π is a pair (t, D), where D is a co-set in π and t is a transition of Σ, such that $h(D) = {}^\bullet t$ and π 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, and [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).

```
input : Σ = (N, M_0) — a bounded net system
output : Unf_Σ — a finite and complete prefix of Σ's unfolding

Unf_Σ ← the empty branching process
add instances of the places from M_0 to Unf_Σ
pe ← PotExt(Unf_Σ)
cut_off ← ∅
while pe ≠ ∅ do
    choose e ∈ pe such that [e] ∈ min_◁{[f] | f ∈ pe}
    if [e] ∩ cut_off = ∅
    then
        add e and new instances of the places from h(e)• to Unf_Σ
        pe ← PotExt(Unf_Σ)
        if e is a cut-off event of Unf_Σ then cut_off ← cut_off ∪ {e}
    else pe ← pe \ {e}
```

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

Moreover, using total orders allows one to simplify some parts of the unfolding algorithm in figure 1, e.g., the 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 [7, 8]).[1]

3 Finding possible extensions

Almost all the steps of the unfolding algorithm in figure 1 can be implemented efficiently. The only hard part is calculating the set of the possible extensions, PotExt(Unf_Σ), and in this paper we will make it the focus of our attention. Since 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:

- reducing the number of calls to PotExt;
- using heuristics to reduce the cost of a single call; and
- merging the common parts of the work carried out by different calls.

3.1 Reducing the number of calls

An excellent example of a method aimed at reducing the number of calls to PotExt 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, 13], 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 only events consuming conditions from $e^\bullet$, where e is the latest event inserted into the unfolding. This can be formalised as follows.

Definition 2. *Let π be a branching process of a net system Σ, and e be its event. A possible extension (t, D) of π is a* (π, e)-extension *if $D \cap e^\bullet \neq \emptyset$, 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 efficiently implemented using, e.g., a binary heap. The call to PotExt(Unf_Σ) in the body of

[1] The more general case of non-local corresponding configuration involves performing a reachability analysis when checking whether an event is a cut-off event, which can be quite time consuming [9].

input : $\Sigma = (N, M_0)$ — a safe net system
output : Unf_Σ — a finite and complete prefix of Σ's unfolding

$Unf_\Sigma \leftarrow$ the empty branching process
add instances of the places from M_0 to Unf_Σ
add instances of the transitions enabled by M_0 to pe
$cut\_off \leftarrow \emptyset$
while $pe \neq \emptyset$ **do**
  /* pe can be implemented as a binary heap */
  choose $e \in pe$ **such that** $[e] = \min_\lhd\{[f] \mid f \in pe\}$
  $pe \leftarrow pe \setminus \{e\}$
  /* the following test can be implemented as one look-up in a hash table */
  if $\exists f \in Unf_\Sigma$ **such that** $Mark([e]) = Mark([f])$
  then $cut\_off \leftarrow cut\_off \cup \{e\}$
  else
    add e and new instances of the places from $h(e)^\bullet$ to Unf_Σ
    UpdatePotExt(pe, Unf_Σ, e)

for all $e \in cut\_off$ **do**
  add e and new instances of the places from $h(e)^\bullet$ to Unf_Σ

Fig. 2. An improved unfolding algorithm ($\lhd$ must be a total adequate order).

the main loop of the algorithm is replaced by UpdatePotExt(pe, Unf_Σ, e), which finds all the possible extensions which consume at least one condition produced by e, and inserts them into the queue. This approach also reduces the size of the NP-complete problems to be solved at least by one. Moreover, in the important special case of binary synchronisation, when the size of a transition's preset is at most 2, say ${}^\bullet t = \{h(c), p\}$, the problem is equivalent to finding the set $\{c' \in h^{-1}(p) \mid c' \ co \ c\}$, which can be efficiently computed (note that the problem becomes vacuous when $|{}^\bullet t| = 1$). Moreover, this technique leads to a further simplification in the algorithm of figure 1. Indeed, since now we can be sure that we do not compute any possible extension more then once, 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. The resulting unfolding algorithm is given in figure 2.

Note 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 (π, e)-extensions. In what follows, we identify two such cases.

In the pseudo-code given in [13], 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; 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 each time when there is a transition $t \in (h(e)^\bullet)^\bullet$ such that its instance 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.*

Proof. Since $h(e^\bullet \cap {}^\bullet f) \subseteq h(e)^\bullet \cap {}^\bullet h(f)$ always holds, there exists a place $p \in (h(e)^\bullet \cap {}^\bullet h(f)) \setminus h(e^\bullet \cap {}^\bullet f)$. By the definition of a net unfolding, there are distinct non-conflicting p-labelled

conditions $c \in e^\bullet$ and $d \in {}^\bullet f$. Hence, as the net system is safe, c and d are not concurrent, i.e., $c \prec d$ or $d \prec c$ holds. Since the latter contradicts $f \in (e^\bullet)^\bullet$, we have $c \prec d$, and so e and f are separated. □

Corollary 1. *Let π be a branching process of a safe net system, e be an event of π, and (t, D) be a (π, 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. Indeed, 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 3; note that in such a case e and f are separated).

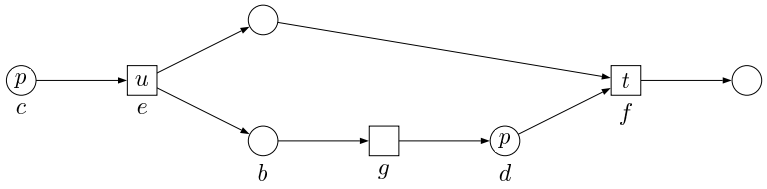

Fig. 3. A t-labelled event f cannot be inserted immediately after a u-labelled event e, if $p \in {}^\bullet t \cap ({}^\bullet u \setminus u^\bullet) \neq \emptyset$, even though it can consume a condition produced by e.

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

Proof. Let $p \in {}^\bullet h(f) \cap ({}^\bullet h(e) \setminus h(e)^\bullet)$, and $c \in {}^\bullet e$ and $d \in {}^\bullet f$ be two p-labelled conditions (see figure 3). Since e and f are distinct non-conflicting events, c and d are distinct non-conflicting conditions. Hence, as the net system is safe, c and d are not concurrent, i.e., $c \prec d$ or $d \prec c$ holds. Since the latter contradicts $f \in (e^\bullet)^\bullet$, we have $c \prec d$. Now, as $p \notin h(e)^\bullet$, there must be a non-p-labelled condition $b \in e^\bullet$ such that $b \prec d$, and so e and f are separated. □

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

In view of the above corollary, the algorithm may consider only the 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 can have the form given in figure 4. In order to efficiently find all the conditions which are concurrent to a condition d, one can maintain the concurrency relation, as suggested in [6]. However, such an approach is not suitable if we aim at producing large unfoldings. Another way is to mark in the procedure Cover all the conditions which are not concurrent to d as unusable, and unmark them during the backtracking.

It is interesting to apply this technique in the special case where a transition t has a self-loop, i.e., $t \in (t^\bullet)^\bullet$ (clearly, if $t \notin (t^\bullet)^\bullet$ then a new instance of t cannot be inserted after an event marked by t). We consider the following cases:

- ${}^\bullet t \subset t^\bullet$. Then t must be dead, otherwise the net is not safe (and even unbounded). Indeed, executing t produces all necessary tokens for t to be executed again. Therefore, if t is not dead, it can be executed successively arbitrary number of times, producing arbitrary many tokens on the places from $t^\bullet \setminus {}^\bullet t \neq \emptyset$.

- ${}^\bullet t = t^\bullet$ (such 'strange' transitions do appear in some of the examples attempted in section 5, e.g., in the ELEVATOR(n) series). Then if an instance e of t occurs in the unfolding then, according to proposition 1, only one instance f of t (with ${}^\bullet f = e^\bullet$) can be inserted directly after e, producing a cut-off event (note that $Mark([f]) = Mark([e])$ and $|[f]| > |[e]|$; moreover, if we allow non-local corresponding configurations, as suggested in [9], then every instance e of t is a cut-off event, with the corresponding configuration $[e] \setminus \{e\}$, and so we do not have to insert anything after it).
- ${}^\bullet t \cap t^\bullet \neq \emptyset$ and ${}^\bullet t \not\subseteq t^\bullet$. Then, by corollary 2, another instance of t cannot be inserted directly after t.

As a result, a new instance of t can be inserted after a t-labelled event only if ${}^\bullet t = t^\bullet$.

```
procedure UpdatePotExt(pe, Unf_Σ, e)
    extensions ← ∅  /* global */
    for all t ∈ (h(e)•)• \ (•h(e) \ h(e)•)• do
        preset ← e• ∩ h^{-1}(•t) /* not complete yet */
        C ← all conditions concurrent to e
        Cover(C, t, preset)
    pe ← pe ∪ extensions

procedure Cover(C, t, preset)
    if |•t| = |preset|
    then extensions ← extensions ∪ {(t, preset)}
    else
        choose p ∈ •t \ h(preset)
        for all d ∈ C ∩ h^{-1}(p) do
            C' ← {c ∈ C | c co d}
            Cover(C', t, preset ∪ {d})
```

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

3.2 Merging computation common to several calls

The presets of candidate transitions for inserting after an event e have often common parts besides the places from $h(e)^\bullet$, and the algorithm may be finding instances of the same 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 by the following example.

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 the 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. Hence, 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, we can obtain a tree shown in figure 5(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 t be a transition of a net system Σ and $U = (t^\bullet)^\bullet \setminus ({}^\bullet t \setminus t^\bullet)^\bullet$. A* preset tree *of t, PT_t, is a directed tree satisfying:*

- *Each vertex is labelled by a set of places, in such a way that the root is labelled by $\emptyset$, and the sets labelling the nodes of any directed path are mutually disjoint.*

- *Each transition $u \in U$ has an associated vertex v, such that the union of all the place sets along path from the root to v is equal to ${}^\bullet u \setminus t^\bullet$ (note that different transitions may have the same associated vertex).*
- *Each leaf is associated to at least one transition.*

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

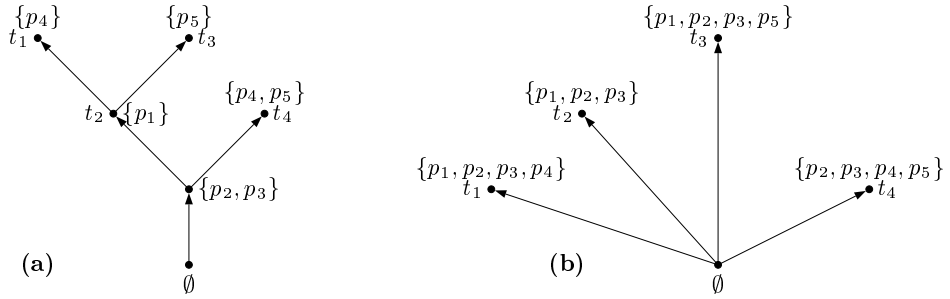

Fig. 5. **(a)** An optimised preset tree of weight 7, and **(b)** a non-optimised one with weight 15.

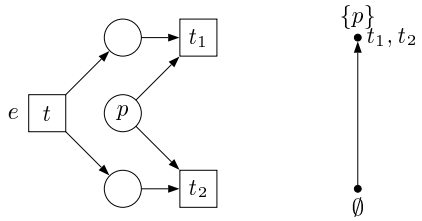

Fig. 6. Using preset trees may be useful even if PreMax = 2 and no two transitions have the same preset.

Having built a preset tree, we can use the algorithm shown in figure 7 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 figure 6). Note that we only need one preset tree PT_t per transition t of a net system, which can be built during the preprocessing stage.

Building preset trees Two important problems which we will 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 5(b) instead of that in figure 5(a) as an input to the algorithm in figure 7, it will work in a way very similar to that of the algorithm in figure 4. But one can see that gluing the common parts of the presets decreases both the weight of the preset tree and the number of times the algorithm in 7 attempts to find new conditions concurrent to the already constructed part of event presets. Therefore, in general, preset trees with small weight should be preferred.

Such a 'minimal weight' criterion is rather rough, but it is hard to predict during the preprocessing stage which preset tree will be better, as different preset trees 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 greedy heuristics, which might be used for resolving the non-deterministic choice in the algorithm in

```
procedure UpdatePotExt(pe, Unf_Σ, e)
    tree ← preset tree for h(e)  /* pre-calculated */
    extensions ← ∅  /* global */
    C ← all conditions concurrent to e
    preset ← e• ∩ h^{-1}(•t) /* not complete yet */
    Cover(C, tree, preset)
    pe ← pe ∪ extensions

procedure Cover(C, tree, preset)
    for all transitions t labelling the root of tree do
        extensions ← extensions ∪ {(t, preset)}
    for all sons tree' of the root of tree do
        v ← root of tree'
        R ← places labelling v
        for all co-sets CO ⊆ C such that h(CO) = R do
            C' ← {c ∈ C | c co CO}
            Cover(C', tree', preset ∪ CO)
```

Fig. 7. An algorithm for updating the set of possible extensions.

figure 7. But this drawback is usually more than compensated for by the speedups gained by merging the common parts of the work spent on finding the co-sets forming the presets of newly inserted events.

Note that there may exist a whole family of minimal-weight preset trees for the same transition. We 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 criteria. However, this may get too complicated and the process of building a preset tree can easily become more time consuming then the unfolding itself. And, even if a very complicated criterion is used, the effort spent to build a highly optimised preset tree can be wasted: the transition might be dead, and the corresponding preset tree will never be used by the algorithm.[2] Therefore, in the actual implementation, we decided to adopt the simple 'minimal weight' criterion.

Minimal-weight preset trees One can see that building a 'minimal weight' preset tree can be reduced to the Steiner tree problem for graphs which is known to be NP-complete (and, in such a reduction, the size of the graph itself can be exponential in the number of candidate transitions). We do not know whether there is an exact polynomial-time algorithm for solving this problem. Therefore, we decided to implement a relatively fast approximate greedy algorithm. This decision can be justified by the fact that the weight of a preset tree is only a rough estimate of its quality, and so it is reasonable not to solve the optimisation problem in the strictest sense, but rather to quickly build 'acceptably light' preset trees.

In figures 8 and 9 we outlined simple bottom-up and top-bottom algorithms for solving this problem. Note that the input in each case is a set of sets of places $\{A_1, \ldots, A_k\} = \{{}^\bullet u \setminus t^\bullet \mid u \in U\} \cup \{\emptyset\}$. As it is trivial to assign vertices to the transitions, we do not show this part in the algorithms in figures 8 and 9. We denote by $Tree(v, \{Tr_1, \ldots, Tr_l\})$ a tree with the root v and sons $Tr_1, \ldots, Tr_l$, which are trees, and use $\cdot$ instead of the set of son trees if

[2] In the DPFM(11) example, the net has 5633 transitions, but the built complete prefix has only 199 non-cut-off events; in our experiments, the preprocessing stage took more time then the process of unfolding, even though the simple 'minimal weight' criterion was used. This suggest that one might generate preset trees 'on demand' during a run of the unfolding algorithm and cache them.

their identities are irrelevant. Moreover, in the further discussion, we will often identify a tree with the set of places at its root, providing that this will not create an ambiguity. The two algorithms do not necessarily give an optimal solution, but in most cases the results are acceptable. We implemented them both, to check which approach performs better. The tests indicated that in most cases the produced trees had the same weight, but sometimes the bottom-up approach suffered from the effect which can be illustrated by the following example. Let $A_1 = \{p_1, \ldots, p_{10}\}$, $A_2 = \{p_2, \ldots, p_{10}\}$, $A_3 = \{p_1, p_{11}\}$, and $A_4 = \{p_1, p_{12}\}$. On the first iteration of the algorithm p_1 is chosen, and this results in the tree of weight 21, shown in figure 10(a), whereas it is possible to build a tree of weight 13 (figure 10(b)).

function BUILDTREE($S = \{A_1, ..., A_k\}$)
  if $k = 0$ **then** **return** 'empty tree'
  $root \leftarrow \bigcap_{A \in S} A$
  $S \leftarrow \{A_1 \setminus root, \ldots, A_k \setminus root\}$
  $TS \leftarrow \emptyset$
  while $\bigcup_{A \in S} A \neq \emptyset$ **do** /* while there are non-empty sets */
    choose $p \in \bigcup_{A \in S} A$ **such that** $|\{A \in S \mid p \in A\}|$ is maximal
    $Tree(v, ts) \leftarrow$ BUILDTREE($\{A \setminus \{p\} \mid A \in S \wedge p \in A\}$)
    $TS \leftarrow TS \cup \{Tree(v \cup \{p\}, ts)\}$
    $S \leftarrow \{A \in S \mid p \notin A\}$
  return $Tree(root, TS)$

Fig. 8. A bottom-up algorithm for building trees.

function BUILDTREE($\{A_1, ..., A_k\}$)
  $TS \leftarrow \{Tree(A_1, \emptyset), \ldots, Tree(A_k, \emptyset)\}$
  while $|TS| > 1$ **do**
    choose $Tree(A', \cdot) \in TS$ and $Tree(A'', \cdot) \in TS$
    such that $A' \neq A''$ and $|A' \cap A''|$ is maximal
    $A \leftarrow A' \cap A''$
    $T \leftarrow \{Tree(B \setminus A, \cdot) \in TS \mid A \subset B\}$
    $TS \leftarrow TS \setminus \{Tree(B, \cdot) \in TS \mid A \subseteq B\}$
    $TS \leftarrow TS \cup \{Tree(A, T)\}$
  /* $|TS| = 1$ */
  return the remaining tree $Tr \in TS$

Fig. 9. A top-down algorithm for building trees.

The top-down algorithm is more stable, and only in rare (see, for example, figure 11) cases it produces 'heavier' trees then the bottom-up one. Therefore we will focus our attention on the top-down algorithm and its efficient implementation.

Top-down algorithm A sketch of a possible implementation of the top-down algorithm for building preset trees is shown in figure 12. Note that the size of any set which can appear during the calculations does not exceed PREMAX. Therefore, we can assume that the cardinalities are attached to the sets, and the remaining operations can be performed in $O(\text{PREMAX})$ worst case time. In practice, the sets usually quickly degrade to singletons or to the empty set, so

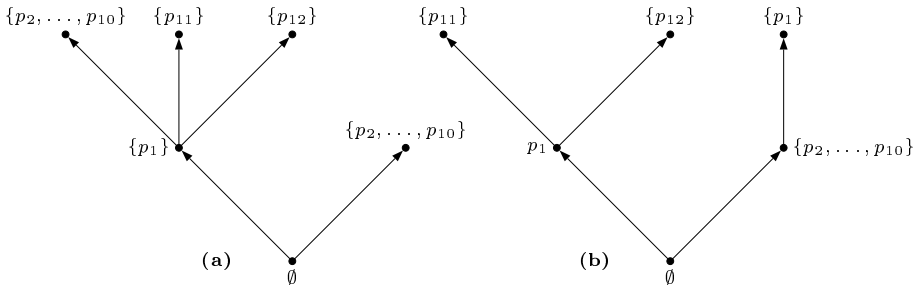

Fig. 10. **(a)** A tree of weight 21, produced by the bottom-up algorithm; **(b)** a tree of weight 13, corresponding to the same sets. Arcs in series were merged unless the middle vertex can be associated with a transition.

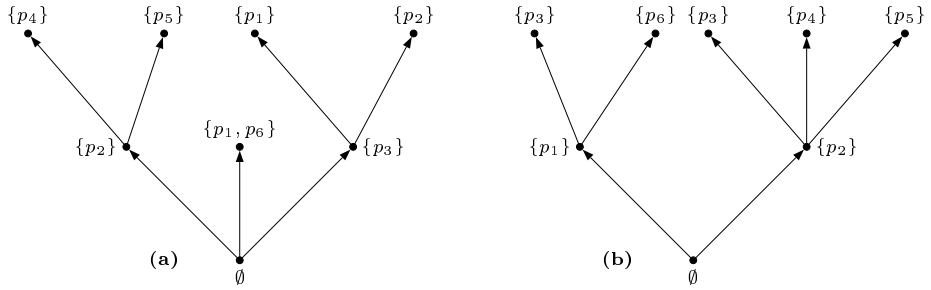

Fig. 11. **(a)** A tree of weight 8, produced by the top-down algorithm for the sets $A_1 = \{p_1, p_3\}$, $A_2 = \{p_1, p_6\}$, $A_3 = \{p_2, p_3\}$, $A_4 = \{p_2, p_4\}$, and $A_5 = \{p_2, p_5\}$ (the intersection $\{p_3\} = A_1 \cap A_3$ was chosen on the first iteration); **(b)** a tree of weight 7, produced by the bottom-up algorithm for the same sets. Arcs in series were merged unless the middle vertex can be associated with a transition.

that the set operations (we only need to compute intersection of sets, cardinality of a set, and check set inclusion and equality) in our case are relatively inexpensive.

The idea of the algorithm is to compute all pairwise intersections of the sets from TS before the main loop starts, and then maintain this data structure. On each step, the algorithm chooses a set I of maximal cardinality from *Intersections*, and updates the variables TS and *Intersections* in the following way. It finds all the supersets of I in TS, and removes them (this can be done using $O(|\,TS|)$ operations with sets, if TS is implemented as, e.g., a double linked list). Moreover, the algorithm also removes all the corresponding to these sets intersections from *Intersections*. Then the intersections of I with the remaining in TS sets are added into *Intersections*, and I is inserted into TS.

Since we have k sets in the beginning of the algorithm, the main loop of the algorithm cannot be executed more then $k-1$ times, because on each step we remove at least 2 sets from the set TS and then add one. Therefore, the number of sets which ever appear in TS does not exceed $2k-1$, i.e., the algorithm performs $O(k^2)$ insertions into *Intersections*, $O(k^2)$ removals from there, and $O(k)$ operations of finding a set with maximal cardinality (note that the size of *Intersections* is $O(k^2)$). Using a suitable data structure, e.g., a red-black tree, for representing *Intersections*, we can perform any of these operation in $O(\textsc{PreMax}\cdot \log k^2) = O(\textsc{PreMax}\cdot \log k)$ worst case time (note that in this case we have to introduce another operation on sets, viz. checking if $A_i \prec_{tot} A_j$, where $\prec_{tot}$ is an arbitrary total order, refining the size order $A_i \prec_{size} A_j \Leftrightarrow |A_i| < |A_j|$. Therefore, the worst case time of the algorithm in figure 12 is $O(\textsc{PreMax}\cdot k^2 \cdot \log k)$.

```
function BuildTree({A_1, ..., A_k})
  TS ← {Tree(A_1, ∅), ..., Tree(A_k, ∅)}
  /* Intersections is a multiset of sets */
  Intersections ← {A' ∩ A'' | A' ≠ A'' ∧ Tree(A', ·) ∈ TS ∧ Tree(A'', ·) ∈ TS}
  while |TS| > 1 do
    choose I ∈ Intersections such that |I| is maximal
    T ← {Tree(B \ I, ·) ∈ TS | I ⊂ B}
    for all Tree(A, ts) ∈ TS such that I ⊆ A do
      TS ← TS \ {Tree(B, ts)}
      for all Tree(B, ·) ∈ TS do
        Intersections ← Intersections \ (A ∩ B)
    for all Tree(A, ·) ∈ TS do
      Intersections ← Intersections ∪ (I ∩ A)
    TS ← TS ∪ {Tree(I, T)}
  /* |TS| = 1 */
  return the remaining tree Tr ∈ TS
```

Fig. 12. A top-down algorithm for building preset trees.

We can achieve $O(\textsc{PreMax}\cdot k^2)$ average time, using instead of red-black trees the following simple data structure for *Intersections*. For each possible size of a set (as it was already mentioned, these sizes do not exceed PreMax), we keep a double linked list of sets, having this cardinality (reducing, thus, inserting a set into *Intersections* to adding it into the list corresponding to its cardinality). In order to facilitate removing an item from *Intersections*, we can maintain a hash table (note, that in this case we need an additional operation with sets — computing the hash code of a set, and may assume that this takes $O(\textsc{PreMax})$ in the worst case). With this idea, removing of a set can be done in $O(\textsc{PreMax})$ average time. Moreover, if we have a variable to store the current maximal cardinality (note, that the size of sets added into *Intersections* cannot exceed the size of the last removed from there set, and, therefore, the number of times this variable is updated does not exceed the number of possible cardinalities) then the total time spent by algorithm on finding sets with maximal cardinality is $O(k + \textsc{PreMax})$.

It is essential for the correctness of the algorithm that *Intersections* is a multiset, not a set, and we have to keep duplicates in our data structure. It is probably better to implement this by keeping a counter for each set inserted into *Intersections*, 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 (note that inserting a set is now slightly more complicated, but we can use the same hash table as for removing, and perform it in $O(\textsc{PreMax})$ average time). Moreover, if the multiplicities are calculated, we often can reduce the weights of produced trees. The idea is to choose 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 11(a), forcing $\{p_2\}$ to be chosen on the first iteration). Let us show that such sets have the highest multiplicity among the sets with the maximal cardinality. Indeed, each time at the moment this choice is made by the algorithm, the values of TS and *Intersections* are 'synchronised' in the sense that *Intersections* contains all pairwise intersections of the sets from TS, with proper multiplicity. Now, let $I \in$ *Intersections* be a set with the maximal cardinality, and there are n its 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 *Intersections* with cardinality greater then $|I|$, the intersections of any two distinct supersets of I from TS is exactly I. Therefore, the multiplicity of I is $C_n^2 = n(n-1)/2$. This function is strictly monotonic for all positive n, thus, there is monotonic one-to-one correspondence between the multiplicities of sets with the maximal cardinality from *Intersections*, and the numbers of their supersets in TS. Therefore,

among the sets of maximal cardinality, those having the maximal multiplicity have the maximal number of supersets in TS.

It is easy to implement this improvement if a red-black tree is used to represent *Intersections*. The only thing to be done is to choose the total ordering on the nodes of the tree refining the following *size-multiplicity* one: $A_i \prec_{sm} A_j \Leftrightarrow |A_i| < |A_j| \vee |A_i| = |A_j| \wedge \mathit{Intersections}(A_i) < \mathit{Intersections}(A_j)$. We may assume that it can be computed in $O(\textsc{PreMax})$ worst case time, so the worst case complexity of the algorithm remains $O(\textsc{PreMax}{\cdot}k^2{\cdot}\log k)$.

But if we want to use the described above data structure based on a hash table, some changes are to be made. The idea is to 'slice out' the sets from *Intersections*, having the maximal cardinality, and handle them separately. The remaining sets can be processed in the same way as before (we never have to look for a set with the maximal multiplicity among them, because their cardinalities are not maximal). The multiplicities of the 'sliced out' sets are of the form C_i^2, where $i \leq |TS| \leq k$, and we can hold them in an array of double linked lists, such that the items of the i-th list have the multiplicity C_i^2 (note that the size of the array does not exceed $i_{max} \leq k$, where $C_{i_{max}}^2$ is the maximal multiplicity). This additional data structure can be built in time linear in the number of sets in *Intersections*, having the maximal cardinality, therefore the total time spent on building such a structure is $O(k^2)$, since the total number of sets which appear in *Intersections* is $O(k^2)$.

Now, we observe that the sizes of all the sets, which are inserted into *Intersections*, are less than the current maximal cardinality, so we don't have to modify our additional structure when inserting new items. Therefore, the only operations we still need are removing a set and finding a set of maximal multiplicity. The latter is simple if we maintain the index of the non-empty list containing the sets of maximal multiplicity. Since we never add new sets to the lists, and the multiplicity of the sets of maximal cardinality can only decrease, the value of this index can only decrease for each particular cardinality (though it may be increased when the algorithm, having exhausted the current 'slice' of sets of the maximal cardinality, switches to smaller ones). Since the sum of i_{max}'s for all slices does not exceed $2k$ (because for each slice at least once we replace $i_{max} \geq 2$ sets from TS by one set), the total number of updates of this index is $O(k + \textsc{PreMax})$.

Therefore, the only thing which still needs to be explained is how to remove a set of the maximal cardinality from *Intersections*. If the multiplicity of this set is one, we can just remove it from the list to which it belongs. But if its multiplicity is greater than one then it is of the form C_i^2, and, since the multiplicities of the sets of the maximal cardinalities corresponds to the number of their supersets in TS, we can be sure that this set will be removed for several times by the end of the current iteration, so that eventually its multiplicity will be of the form C_j^2 for some $j < i$. Therefore, we can use the algorithm shown in figure 13 (note that we allow the multiplicities of the set in the i-th list to be greater than C_i^2, but it is guaranteed that they will have the proper form by the end of the current iteration of the main loop). The average time for removal an item is $O(\textsc{PreMax})$, thus, this implementation of the algorithm for building trees takes in average $O(\textsc{PreMax}{\cdot}k^2)$ time units.

4 Further optimisations

In this sections we describe some optimisations. Even though they do not have major influence on the performance of the developed algorithm, they turned out to be quite useful in practice.

Cut-offs 'in advance' In the algorithm shown in figure 2, the newly computed possible extensions are inserted into *pe*, and the test for being a cut-off event is performed only when an event reaches the top of the queue. Since such a test is relatively inexpensive in case of the total ordering (one look up in a hash table), one can afford to perform it each time a new possible extension e is computed. In some cases the cut-off property might be detected in this way earlier, and we can add e directly into *cut_off*, without inserting it into the queue (which

```
procedure REMOVE(A)
    if the cardinality of A is not maximal
    then remove A in the usual way
    else
        if the multiplicity of A is 1
        then remove A from the 2nd list /* 1 = C_2^2 */
        else
            if ∃i such that the multiplicity of A is C_i^2
            then                move A from the i-th list to (i − 1)-th list
                 decrement the multiplicity of A
```

Fig. 13. Removing a set from *Intersections*.

potentially involves computing 'that complex order' several times). If the test is not passed, the algorithm proceeds in the usual way and inserts e into pe. The cut-off criterion for e will be tested once again, when it reaches the top of the queue, but the overhead is small comparing with the potential gains.

We can extend this test even further, involving events which are currently in the queue. Indeed, if there is $f \in pe$ such that $Mark([e]) = Mark([f])$ then either e or f is cut-off event: if $[f] \lhd [e]$ then e can be added into *cut_off*; otherwise, f is added into *cut_off*, and one should replace f by e in pe (note that if pe is implemented as a binary heap, the heap property may be destroyed, but we can easily restore it by shifting e on the proper place). Such an approach usually allows to reduce the number of comparisons of configurations.

In the case when non-local corresponding configurations are allowed, checking the cut-off condition involves reachability analysis and so can be expensive. But in the experiments conducted in [9], the computed corresponding configurations often were local (there were quite a few examples where this was true of *all* corresponding configurations). This suggest to quickly check (using a hash table) whether an event is a cut-off event with a local corresponding configuration, and perform the complete test only if this not the case. Therefore, before inserting e into the queue, we can check only the fast sufficient condition, based on local corresponding configurations.

Computing final markings In order to check the cut-off criterion we have to compute final markings of configurations. We can use the definition $Mark(C) = h(Cut(C))$, and calculate the cut using the formula $Cut(C) = (Min(ON) \cup C^\bullet) \setminus {}^\bullet C$, but this approach is not efficient.

A better method is to compute the vector $(h(C)(t_1), \ldots, h(C)(t_n))$, where $t_1, \ldots, t_n$ are the transitions of Σ, and consider it as the Parikh vector x_σ of some linearisation σ of the partial order execution represented by C. Having computed this vector, we can use the marking equation $M = M_0 + \mathcal{N} \cdot x_\sigma$ (where $\mathcal{N}$ is the incidence matrix of the net system) to calculate the final marking of C. The advantage of this approach is that the calculation is performed using the original net system rather than its unfolding, which can be much larger.

In the case of safe net systems, this approach can be further refined. Indeed, we can perform all calculations modulo 2, which is useful if final markings are represented as bit vectors. As our experiments showed, this approach is so inexpensive that we can recompute final markings each time they are needed rather than attach them to the corresponding events (we estimate that the time overhead is less than 5%, yet the memory gains my be significant).

5 Experimental results

The results of our experiments are summarised in tables 1–4, 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 average/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 $\mathrm{W}_{opt}/\mathrm{W}$, where W_{opt} is the sum of the weights of the constructed preset trees, and W is the sum of the weights of the 'non-optimised' preset trees (see figure 5). This ratio may be used as a rough approximation of the effect of employing preset trees: $\mathrm{W}_{opt}/\mathrm{W} = 1$ means that there is no optimisation. Note that when transition presets are large, employing preset trees gives certain gains, even if this ratio is close to 1 (see, e.g., the DME(n) series).

The implementations of unfolding algorithms are usually quite intricate and error prone. In order to increase the confidence in the developed implementation, we checked that the prefixes produced by it are isomorphic to those generated by `ERVunfold`.[3] For this, a special utility for 'sorting' prefixes was developed, so that if two prefixes were isomorphic then after 'sorting' they become equal (technically, the resulting files where they are stored are equal). It assumes that initially non-cut-off events are sorted accordingly to the adequate order $\lhd$ on their local configurations and proceeds in the following way:

1. Separate cut-off events, pushing them to the end (note that if an unfolding algorithm checks cut-offs 'in advance' then we only can guarantee that non-cut-off events are sorted accordingly to $\lhd$).
2. Separate post-cut-off conditions, pushing them to the end.
3. Sort non-post-cut-off conditions according to the following ordering: $c' \lessdot c'' \iff$

   $$\begin{array}{ll} \text{(a)} & e' \lhd e'' \text{ or} \\ \text{(b)} & e' = e'' \text{ and } h(c') \ll h(c''), \end{array}$$

   where $\{e'\} \in {}^\bullet c'$, $\{e''\} \in {}^\bullet c''$, and $\ll$ is an arbitrary total order on the places of the original net system (e.g., the size-lexicographical ordering on their names). Note that non-cut-off events are sorted accordingly to the the adequate order on their local configurations.
4. Sort the presets of the events (including the cut-offs) accordingly to $\lessdot$.
5. Sort the cut-off events accordingly to the following ordering: $e' \lessdot e'' \iff$

   $$\begin{array}{ll} \text{(a)} & {}^\bullet e' \lessdot_{sl} {}^\bullet e'' \text{ or} \\ \text{(b)} & {}^\bullet e' = {}^\bullet e'' \text{ and } h(e') \ll h(e''), \end{array}$$

   where $\lessdot_{sl}$ is the size-lexicographical order, built upon $\lessdot$, and $\ll$ is an arbitrary total order on the set of the transitions of the original net system (e.g., the size-lexicographical ordering on their names). Note that the conditions which can appear in the presets of the events are already sorted.
6. Sort post-cut-off conditions according to $\lessdot$. Note that all events have already been sorted by this step.
7. Sort the postsets of the events (including the cut-offs) accordingly to the $\lessdot$ ordering. Note that all conditions have already been sorted by this step.

Test cases We attempted (tables 1 and 2) the popular set of benchmark examples,[4] collected by J.C. Corbett ([3]), K. McMillan, S. Melzer, S. Römer (this set was also used in [6, 9–11, 14]). It was available from K. Heljanko's homepage.

[3] Note that when the algorithms use the same adequate total order, then the resulting prefixes must be isomorphic (providing that the implementations are correct).

[4] The examples CYCLIC(n), DAC(n), OVER(n) and RING(n) from this set are not shown in the tables, since they were trivial for both algorithms.

The transitions in these examples usually have small sizes of presets (in fact, they do not exceed 2 for all the examples in table 1). Thus, the advantage of using preset trees is not very big, and `ERVunfold` is usually quicker due to maintaining 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 on these examples, but its running time on all the examples in table 1 is quite acceptable. Moreover, sometimes using preset trees does have certain advantages and it scales better (e.g., for the ELEVATOR(n) and MMGT(n) series).

In table 2, the results concerning nets with slightly bigger transition presets are shown. Our algorithm is almost as fast as `ERVunfold`, and scales better. Unfortunately, the larger instances of the DME(n) series were not available for us (DME(12) appeared to be unsafe). The only example in this set with big maximal preset is BYZAGR(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 presets of the other transitions in this net do not exceed 5.

In order to test the algorithms on nets with larger presets, we have built a set of examples RND(m,n,k) in the following way. First, there were created m loops consisting of n places and n transitions each; the first place of each loop was marked with one token. Then k additional transitions were added to this carcass, so that each of them takes 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 mn transitions with presets of size 1 and k transitions with presets of size m). It is easy to see that the built in this way nets are safe. The experimental results for several such nets are shown in table 3.

One can argue that random nets is not a practical example, and that in practice the presets of transitions are small (and do not contain more than two places in the case of binary synchronisation). So we tried to spot areas where nets with large transitions' presets naturally arise. The first candidate is data intensive applications, where processes being modelled compute functions depending on many variables. As an example, we modelled priority arbiters based on dual-rail logic, described in [1]. Basically, a priority arbiter handles requests from several concurrent processes and decides (using some priority system) which request should be granted. 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 with the following priority function:

1. Group with the largest number of requests is handled first.
2. Among several groups with the same number of requests, the one having the smallest number is handled first.
3. Within a group, the process with the smallest number, sending a request, is granted first.

The results are summarised in table 4. Our algorithm scales better and was able to produce much larger unfoldings.

We expect that many other areas, where Petri nets with large presets are needed, will be found. Potentially, such nets appear as the result of net transformation, e.g., introducing complementary places or converting bounded nets into safe ones.[5] But even for nets with small transitions' presets our algorithm is quite quick (note that it was slower than `ERVunfold` for the examples in tables 1–2 because we abandoned the concurrency relation, trading speed for possibility of building large prefixes — in principle, maintaining concurrency relation is compatible with all the described heuristics), and may be used if the size of the finite prefix is expected to be large.

6 Conclusions

Experimental results indicate that the algorithm we propose in this paper can build quite large unfoldings. But it might still be slow for practical size problems. In general, generating

[5] Such transformation is the basis of the unfolding algorithm for arbitrary bounded nets described in [8].

unfoldings is still a bottleneck for the unfolding based verification of Petri nets, and our future research will aim at developing an effective parallel algorithm for constructing larger unfoldings. Another promising area is the approach allowing non-local correspondent configurations, proposed in [9]. It sometimes allows to significantly reduce the size of complete prefixes. We plan to investigate if this idea can be efficiently implemented.

Acknowledgements

We would like to thank Alex Bystrov for his observtion that transitions of Petri nets modelling dual-rail logics circuits would have large presets, and help with modelling priority arbiters.

References

1. A. Bystrov, D. J. Kinniment and A. Yakovlev: Priority Arbiters. Proc. of *Sixth IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (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*. University of Hawaii at Manoa (1994).
4. J. Engelfriet: Branching processes of Petri Nets. *Acta Informatica* 28 (1991) 575–591.
5. J. Esparza: Decidability and complexity of Petri net problems — an introduction. *Lectures on Petri Nets I: Basic Models* (Springer-Verlag, Lecture Notes in Computer Science 1491) 1998. 374–428
6. J. Esparza and S. Römer: An Unfolding Algorithm for Synchronous Products of Transition Systems. Proc. of *CONCUR'99*, Invited paper, LNCS 1664 (1999) 2–20.
7. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *TACAS'96*, Margaria T., Steffen B. (Eds.). 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. of *Workshop Concurrency, Specification and Programming 1999 (CS&P'99)*, (1999) 83–95.
10. K. Heljanko: Deadlock and Reachability Checking with Finite Complete Prefixes. Technical Report A56, Laboratory for Theoretical Computer Science, HUT, Espoo, Finland (1999).
11. V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. Technical Report CS-TR-711, Department of Computing Science, University of Newcastle (2000).
12. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *4th CAV*, LNCS 663 (1992) 164–174.
13. K. L. McMillan: *Symbolic Model Checking*. PhD thesis, CMU-CS-92-131 (1992).
14. S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. of *Computer Aided Verification (CAV'97)*, O. Grumberg (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1254 (1997) 352–363.

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	$ave/max\|^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-trees*	*Unf*	W_{opt}/W
ABP(1)	43	95	1.97/2	337	167	56	<0.01	0.01	0.01	0.72
BDS(1)	53	59	1.88/2	12310	6330	3701	1.30	<0.01	3.87	0.53
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
SPEED(1)	33	39	1.77/2	4929	2882	1219	0.36	<0.01	0.86	0.54
DP(6)	36	24	2/2	204	96	30	<0.01	<0.01	0.01	1.00
DP(8)	48	32	2/2	368	176	56	0.01	<0.01	0.02	1.00
DP(10)	60	40	2/2	580	280	90	0.01	<0.01	0.02	1.00
DP(12)	72	48	2/2	840	408	132	0.03	0.01	0.04	1.00
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(2)	7	5	1.80/2	12	5	2	0.00	<0.01	<0.01	1.00
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(4)	39	46	1.96/2	680	336	117	0.01	<0.01	0.03	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
ELEVATOR(1)	63	99	1.89/2	296	157	59	0.01	0.01	0.01	0.62
ELEVATOR(2)	146	299	1.95/2	1562	827	331	0.02	0.13	0.14	0.60
ELEVATOR(3)	327	783	1.97/2	7398	3895	1629	0.61	1.59	2.73	0.60
ELEVATOR(4)	736	1939	1.99/2	32354	16935	7337	16.15	25.57	68.43	0.61
FURNACE(1)	27	37	1.65/2	535	326	189	0.01	<0.01	0.02	0.50
FURNACE(2)	40	65	1.71/2	4573	2767	1750	0.19	<0.01	0.54	0.44
FURNACE(3)	53	99	1.75/2	30820	18563	12207	8.18	<0.01	29.10	0.41
GASNQ(2)	71	85	1.94/2	338	169	46	0.01	0.01	0.01	0.94
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(1)	28	21	1.86/2	43	21	4	<0.01	<0.01	<0.01	0.86
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
HART(25)	127	77	1.66/2	179	102	1	0.01	0.01	<0.01	0.98
HART(50)	252	152	1.66/2	354	202	1	0.02	0.01	0.02	0.99
HART(75)	377	227	1.67/2	529	302	1	0.05	0.01	0.05	0.99
HART(100)	502	302	1.67/2	704	402	1	0.09	0.04	0.09	1.00
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(1)	50	58	1.95/2	118	58	20	0.01	0.01	<0.01	0.66
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
SENTEST(25)	104	55	1.89/2	383	216	40	0.01	<0.01	0.03	0.84
SENTEST(50)	179	80	1.93/2	458	241	40	0.02	0.01	0.04	0.87
SENTEST(75)	254	105	1.94/2	533	266	40	0.04	0.01	0.06	0.89
SENTEST(100)	329	130	1.95/2	608	291	40	0.06	0.01	0.09	0.90

Table 1. Experimental results.

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	$ave/max\|^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-trees*	*Unf*	W_{opt}/W
BYZAGR(1,4)	504	409	3.33/30	42276	14724	752	126	0.14	231	0.71
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
DME(2)	135	98	3.24/5	487	122	4	0.01	0.01	0.02	0.93
DME(3)	202	147	3.24/5	1210	321	9	0.07	0.01	0.09	0.93
DME(4)	269	196	3.24/5	2381	652	16	0.25	0.03	0.34	0.93
DME(5)	336	245	3.24/5	4096	1145	25	0.79	0.03	1.01	0.93
DME(6)	403	294	3.24/5	6451	1830	36	2.37	0.05	2.96	0.93
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

Table 2. Experimental results.

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	$ave/max\|^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-trees*	*Unf*	W_{opt}/W
RND(5,2,500)	10	510	4.92/5	4685	1048	1017	0.06	0.73	0.04	0.03
RND(5,3,500)	15	515	4.88/5	7989	2156	1914	0.32	2.32	0.16	0.16
RND(5,4,500)	20	520	4.85/5	20814	5645	4712	1.86	8.02	0.79	0.30
RND(5,5,500)	25	525	4.81/5	55698	14029	11689	11.45	7.36	3.66	0.39
RND(5,6,500)	30	530	4.77/5	84451	21774	17269	31.43	8.68	12.21	0.44
RND(5,7,500)	35	535	4.74/5	144700	36019	28922	82.92	8.90	30.69	0.50
RND(5,8,500)	40	540	4.70/5	235600	56691	46559	196	8.79	62.96	0.54
RND(5,9,500)	45	545	4.67/5	304656	72895	59840	324	7.43	105	0.58
RND(5,10,500)	50	550	4.64/5	419946	98477	82279	554	9.07	160	0.60
RND(5,11,500)	55	555	4.60/5	573697	132344	112310	994	6.20	246	0.63
RND(5,12,500)	60	560	4.57/5	627303	145378	122465	1187	5.72	322	0.65
RND(5,13,500)	65	565	4.54/5	718762	166093	140147	1560	5.27	420	0.67
RND(5,14,500)	70	570	4.51/5	802907	185094	156417	1952	5.58	507	0.69
RND(5,15,500)	75	575	4.48/5	842181	195228	163722	6685	6.63	616	0.70
RND(5,16,500)	80	580	4.45/5	886158	206265	171957	*time*	7.10	717	0.71
RND(5,17,500)	85	585	4.42/5	987605	229284	191576	—	3.78	863	0.72
RND(5,18,500)	90	590	4.39/5	1025166	239069	198524	—	5.62	998	0.73
RND(10,2,500)	20	520	9.65/10	34884	7136	6125	12.46	7.34	1.14	0.25
RND(10,3,500)	30	530	9.49/10	1415681	153628	144548	1638	3.90	82	0.49
RND(10,4,500)	40	540	9.33/10	2344821	252320	237000	*mem*	3.51	207	0.59
RND(10,5,500)	50	550	9.18/10	2485903	271083	250600	—	7.90	331	0.64
RND(10,6,500)	60	560	9.04/10	2535070	280560	255010	—	11.32	485	0.67
RND(10,7,500)	70	570	8.89/10	2537646	285323	254767	—	11.91	663	0.70
RND(10,8,500)	80	580	8.76/10	2534970	289550	254000	—	14.84	872	0.72
RND(15,2,500)	30	530	14.21/15	1836868	135307	128358	*mem*	32.28	70.24	0.37
RND(15,3,500)	45	545	13.84/15	3750719	271074	255560	—	14.69	259	0.57
RND(15,4,500)	60	560	13.50/15	3787575	280560	257515	—	7.54	456	0.67
RND(15,5,500)	75	575	13.17/15	3795090	288075	257515	—	6.38	718	0.73
RND(20,2,500)	40	540	18.59/20	4744587	256197	245750	*mem*	46.71	176	0.43
RND(20,3,500)	60	560	17.96/20	5040080	280560	260020	—	16.36	427	0.61
RND(20,4,500)	80	580	17.38/20	5050100	290580	260020	—	9.03	771	0.71

Table 3. Experimental results.

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	$ave/max\|^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-trees*	*Unf*	W_{opt}/W
SPA(1)	29	19	1.95/4	32	16	1	<0.01	<0.01	<0.01	0.96
SPA(2)	52	37	2.16/4	111	52	4	<0.01	<0.01	<0.01	0.87
SPA(3)	75	57	2.40/4	324	141	19	0.01	0.01	0.01	0.79
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 4. Experimental results.