Parallelisation of the Petri Net Unfolding Algorithm

Keijo Heljanko[1], Victor Khomenko[2], and Maciej Koutny[2]

[1] Department of Computer Science and Engineering,
Helsinki University of Technology
FIN-02015 HUT Finland
Keijo.Heljanko@hut.fi

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

Abstract. In this paper, we first present theoretical results, helping to understand the unfolding algorithm presented in [5, 6]. We then propose a modification of this algorithm, which can be efficiently parallelised, and prove its correctness. We also present additional optimisations.
Keywords: Model checking, Petri nets, parallel algorithms, 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 [1] — 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 ([5, 15]), 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 the development of fast model checking algorithms employing unfoldings ([9–11]), the problem of efficiently building them is becoming increasingly important. Recently, [4–6, 13, 14] addressed this issue — considerably improving the original McMillan's technique — but we feel that generating net unfoldings deserves further investigation.

The contribution of this paper is twofold. First, we present theoretical results, helping to understand the unfolding algorithm presented in [5, 6]. Second, we propose a modification of that algorithm, which can be efficiently parallelised, and prove its correctness. It does not perform any comparisons of configurations except those needed for checking the cut-off criterion, reducing the total number of times two configuration are compared w.r.t. the *adequate* total order proposed in [5] down to the number of cut-off events in the resulting prefix. This allows to gain certain speedup even in a sequential implementation. Some other optimisations are also described.

Our experiments demonstrate that the degree of parallelism is usually quite high and resulting algorithms potentially can achieve significant speedup comparing with the sequential case.

2 Basic Notions

In this section, we first present basic definitions concerning Petri nets, and then recall (see also [3, 5, 6]) notions related to net unfoldings.

Petri nets A *net* is a triple $N \stackrel{\mathrm{df}}{=} (P, T, F)$ 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. $M : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. As usual, we will denote ${}^\bullet z \stackrel{\mathrm{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \stackrel{\mathrm{df}}{=} \{y \mid (z, y) \in F\}$, for all $z \in P \cup T$, and ${}^\bullet Z \stackrel{\mathrm{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \stackrel{\mathrm{df}}{=} \bigcup_{z \in Z} z^\bullet$, 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 $\Sigma \stackrel{\mathrm{df}}{=} (N, M_0)$ 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 $M' \stackrel{\mathrm{df}}{=} M - {}^\bullet t + t^\bullet$. 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, $M(P) \subseteq \{0, 1\}$; and *bounded* if there is $k \in \mathbb{N}$ such that $M(P) \subseteq \{0, \ldots, k\}$, for every reachable marking M.

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 $ON \stackrel{\mathrm{df}}{=} (B, E, G)$ 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 $h : B \cup E \to P \cup T$ 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 Σ ([3]) is a quadruple $\pi \stackrel{\mathrm{df}}{=} (B, E, G, h)$ 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.

Sometimes it is convenient to start a branching process with a (virtual) initial event $\perp$, which has the postset $Min(ON)$, empty preset, and no label. We will assume that $h(\perp)^\bullet = M_0$.

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: (i) M is represented in π; and (ii) for every transition t enabled by M, there is a finite configuration C and an event $e \notin C$ in π such that $M = Mark(C)$, $h(e) = t$ 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 always possible to truncate it and obtain a finite complete prefix, $Pref_\Sigma$. [16] 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 ([5, 8]) that it suffices to designate an event e newly added during the construction of $Pref_\Sigma$ as a cut-off event, if the already built part of a prefix contains a *corresponding* configuration C without cut-off events, such that $Mark(C) = Mark([e])$ and $C \lhd [e]$, where $\lhd$ is an *adequate order*, defined in the following way ([5, 6]).

Definition 1. *A strict partial order $\lhd$ on the finite configurations of the unfolding of a net system is an* adequate order *if*

- *$\lhd$ is well-founded,*
- *$\lhd$ refines $\subset$, i.e., $C_1 \subset C_2 \Rightarrow C_1 \lhd C_2$,*
- *$\lhd$ is preserved by finite extensions, i.e., if $C_1 \lhd C_2$ and $Mark(C_1) = Mark(C_2)$ then $C_1 \oplus E \lhd C_2 \oplus I_{C_1}^{C_2}(E)$ for all finite extensions $C_1 \oplus E$ of C_1.*

Here $C \oplus E$ denotes the fact that $C \cup E$ is a configuration and $C \cap E = \emptyset$, and $I_{C_1}^{C_2}$ is a mapping from the finite extensions of C_1 onto the finite extensions of C_2, i.e., it maps $C_1 \oplus E$ onto $C_2 \oplus I_{C_1}^{C_2}(E)$ (see [5, 6] for details).

We will also write $e \lhd f$ whenever $[e] \lhd [f]$.

In order to detect cut-off events earlier (and thus decrease the size of the resulting complete prefix), it is advantageous to choose 'dense' (ideally, total) orders, and [5, 6] propose such an order $\lhd_{erv}$ for safe net systems; moreover, it is shown there that if a total order is used then the number of 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). The $\lhd_{erv}$ order refines the McMillan's partial adequate order $\lhd_m$ ([5, 16]), which is defined as $C_1 \lhd_m C_2 \iff |C_1| < |C_2|$.

It is often assumed that a corresponding configuration of an event e is the local configuration of some event f, which is called a *correspondent* of a cut-off event e.[1]

The unfolding algorithm presented in [4–6, 13, 14] is parameterised by an adequate order $\lhd$ and can be formulated as shown in figure 1. It is assumed that the function call PotExt(Unf_Σ) finds the set of *possible extensions* of a branching process Unf_Σ (see the definition below).

Definition 2. *Let π be a branching process of a net system Σ, and e be its event. 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. It is a* (π, e)-extension *if $e^\bullet \cap D \neq \emptyset$, and e and (t, D) are not separated.*

[1] The more general case of non-local corresponding configurations involves performing a reachability analysis each time when checking whether an event is cut-off, which can be quite time consuming ([8]).

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

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

Fig. 1. The unfolding algorithm presented in [5].

Note that in the algorithm and further in the 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. We will also denote by Unf_Σ^S, where $S \subseteq \textsc{PotExt}(\mathit{Unf}_\Sigma)$, the branching process obtained by adding events from S (together with their postsets) into Unf_Σ.

When $\triangleleft$ is a total order, the algorithm in figure 1 is deterministic, and thus always yields the same result for a given net system. A surprising fact is that this is also the case for an arbitrary adequate order.

Theorem 1. *If Σ is a bounded net system then the prefixes produced by two arbitrary runs of the algorithm in figure 1 are isomorphic.*

Proof. Suppose that Pref'_Σ and Pref''_Σ are non-isomorphic prefixes generated by two different runs of the algorithm, with the sets of events E' and E'' and the sets of cut-off events CO' and CO'' respectively.

Let $\mathcal{M} = E' \cap E'' \cap ((CO' \setminus CO'') \cup (CO'' \setminus CO'))$ be the set of events shared by the two prefixes, which are cut-off in one of them, but not in the other. We first observe that $\mathcal{M} \neq \emptyset$. Indeed, since the prefixes are not isomorphic, there exists, without loss of generality, $g \in E' \setminus E''$. Therefore there is $\hat{g} \in [g] \subseteq E'$ such that $\hat{g} \in CO'' \setminus CO' \subseteq E''$, i.e., $\hat{g} \in \mathcal{M}$.

Let $f \in \min_\triangleleft \mathcal{M}$ belongs, without loss of generality, to $CO' \setminus CO''$, and $C \subseteq E'$ be its corresponding configuration. We observe that, for every $e \in C$, $e \triangleleft f$, since $C \triangleleft [f]$ and $[e] \subseteq C$ (recall that $\triangleleft$ refines $\subset$). Let E''_0 and pe''_0 be respectively the set of events of Unf''_Σ and the value of the variable pe, both taken at the point when f is chosen in the main loop of the algorithm during the generation of Pref''_Σ. We now observe that if $\hat{f} \in E''$ is such that $\hat{f} \triangleleft f$, then $\hat{f} \in E''_0$. Indeed, suppose that $\hat{f} \notin E''_0$. Clearly, $\hat{f} \notin pe''_0$ due to the minimality of f. Therefore, $\hat{f}$ has not been generated yet by the algorithm. Since $\hat{f} \in E'' \setminus (E''_0 \cup pe''_0)$, there is $\tilde{f} \in pe''_0$ such that $\tilde{f} \prec \hat{f}$. Consequently, $[\tilde{f}] \subset [\hat{f}]$, and so $\tilde{f} \triangleleft \hat{f} \triangleleft f$, contradicting the minimality of f. Hence $\hat{f} \in E''_0$. Therefore, since $f \notin CO''$, $C \nsubseteq E''$ (otherwise C would be a subset of E''_0, causing the algorithm to designate f as a cut-off event in Pref''_Σ), and so there is $e \in C \setminus E''$. Hence there is $\hat{e} \in [e] \cap CO''$. We then note that $\hat{e} \triangleleft e \triangleleft f$ and $\hat{e} \notin CO'$ (since $\hat{e} \in C$ and C is a correspondent configuration of f in Pref'_Σ, and thus by definition contains no cut-off events), i.e., $\hat{e} \in \mathcal{M}$, contradicting the choice of f. □

The above result is also valid in the case when only local corresponding configurations are allowed.

For efficiency reasons, the call to $\textsc{PotExt}(\mathit{Unf}_\Sigma)$ in the body of the main loop of the algorithm in figure 1 can be replaced by a call $\textsc{UpdatePotExt}(pe, \mathit{Unf}_\Sigma, e)$, which finds

input : $\Sigma = (N, M_0)$ — a bounded 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
$pe \leftarrow \textsc{PotExt}(Unf_\Sigma)$
$cut\_off \leftarrow \emptyset$
while $pe \neq \emptyset$ **do**
  choose $Sl \in \textsc{Slices}(pe)$
  if $\exists e \in Sl : [e] \cap cut\_off = \emptyset$
  then
    for all $e \in Sl$ **do** in any order refining $\lhd$
      if $[e] \cap cut\_off = \emptyset$
      then
        add e and new instances of the places from $h(e)^\bullet$ to Unf_Σ
        if e is a cut-off event of Unf_Σ **then** $cut\_off \leftarrow cut\_off \cup \{e\}$
    $pe \leftarrow \textsc{PotExt}(Unf_\Sigma)$
  else $pe \leftarrow pe \setminus Sl$

Fig. 2. Unfolding with slices.

all (π, e)-extensions and inserts such events into pe according to the $\lhd$ order on their local configurations (see [4–6, 14]).

Almost all the steps of the unfolding algorithm can be implemented quite efficiently. The only hard part is computing the set of possible extensions carried out on each iteration of the main loop of the algorithm (a decision version of this problem is, in fact, NP-complete, see [7, 9]), and in this paper we will focus our attention on its parallelisation.

3 Unfolding with slices

We now present a general idea behind the parallel unfolding algorithm proposed in this paper. After that we explain how it can be implemented in the case when $\lhd$ refines $\lhd_m$, and discuss further improvements aimed at reducing the amount of performed work.

When looking at the algorithm in figure 1, one may observe that a possible way of introducing parallelism would be to insert several events from pe simultaneously, rather than to process them one-by-one. This is done in figure 2, where the main loop of the algorithm has been modified in the following way.

> A set of events $Sl \subseteq \textsc{Slices}(pe)$, called a *slice* of the current set of possible extensions, is chosen on each iteration and processed as a whole, without taking any other events out from pe. It is assumed that for every $Sl \in \textsc{Slices}(pe)$: (i) Sl is a non-empty subset of pe; and (ii) for every $e \in Sl$, if g is an arbitrary event in the unfolding of Σ such that $f \prec g$ for some $f \in pe$, or $g \in pe \setminus Sl$, then $g \not\lhd e$. (*)

In particular, if $f \in pe$ and $f \lhd e$ for some $e \in Sl$, then $f \in Sl$. The algorithm in figure 1 can be seen as a special case of that based on slices, by setting $\textsc{Slices}(pe) \stackrel{\text{df}}{=} \{\{e\} \mid e \in \min_\lhd pe\}$.

Note that neither any event in $pe \setminus Sl$ nor any causal descendant of an event in pe can be less w.r.t. $\lhd$ than some event in Sl. Therefore, if $e \in Sl$ is a cut-off event then any of its corresponding configurations is in Unf_Σ^{Sl}, where Unf_Σ is the already built part of the prefix. This essentially means that the events from Sl can be inserted into the prefix *in any order* consistent with $\lhd$ (the cut-off events in Sl must be identified while doing so). Such a modification of the unfolding algorithm is correct due to the following result.

Lemma 1. *If Σ is a bounded net system then the algorithm in in figure 2 terminates with a prefix which can be produced by some run of the algorithm in figure 1.*

$pe \leftarrow pe \setminus \{e\}$ for all e in $\sigma_{(1)}$
add e_{i_1} and its postset to the prefix
$pe \leftarrow$ PotExt(Unf_Σ)
if e_{i_1} is a cut-off event of $Unf_\Sigma^{\{e_{i_1}\}}$
  then $cut\_off \leftarrow cut\_off \cup \{e_{i_1}\}$

$\vdots$

$pe \leftarrow pe \setminus \{e\}$ for all e in $\sigma_{(m)}$
add e_{i_m} and its postset to the prefix
$pe \leftarrow$ PotExt($Unf_\Sigma^{\{e_{i_1},\ldots,e_{i_m}\}}$)
if e_{i_m} is a cut-off event of $Unf_\Sigma^{\{e_{i_1},\ldots,e_{i_m}\}}$
  then $cut\_off \leftarrow cut\_off \cup \{e_{i_m}\}$

(**a**)

add e_{i_1} and its postset to the prefix
if e_{i_1} is a cut-off event of $Unf_\Sigma^{\{e_{i_1}\}}$
  then $cut\_off \leftarrow cut\_off \cup \{e_{i_1}\}$

$\vdots$

add e_{i_m} and its postset to the prefix
if e_{i_m} is a cut-off event of $Unf_\Sigma^{\{e_{i_1},\ldots,e_{i_m}\}}$
  then $cut\_off \leftarrow cut\_off \cup \{e_{i_m}\}$
$pe \leftarrow$ PotExt($Unf_\Sigma^{\{e_{i_1},\ldots,e_{i_m}\}}$)

(**b**)

Fig. 3. The sequences of instructions performed **(a)** by the basic algorithm and **(b)** by the slicing algorithm.

Proof. We observe that before the main loops are executed for the first time, the values of the variables Unf_Σ, cut_off and pe in both algorithms are respectively equal. We now will show that if this holds before an l-th iteration of the main loop of the slicing algorithm (l-th *slicing iteration*) and an l'-th iteration of the main loop of the basic algorithm (l'-th *basic iteration*) then we can guide the non-deterministic choice in the next l'' basic iterations in such a way that this condition holds also on completion of the l-th slicing iteration, and the $(l'+l''-1)$-th basic iteration (i.e., one slicing iteration is, in general, equivalent to *several* basic iterations). The underlying idea is to force the **choose** operator in the main loop of the basic algorithm to select (possibly with repetitions) those events from pe which are in the set Sl chosen in the slicing algorithm, so that the sequence of insertions of events into the prefixes being built is the same in both algorithms.

Let $\sigma \overset{\mathrm{df}}{=} e_1 \ldots e_n$ be the enumeration of the events in Sl which was driving the **for all** loop in the slicing iteration we are now considering (this execution order refines $\lhd$). Moreover, let $\widehat{\sigma} \overset{\mathrm{df}}{=} e_{i_1} \ldots e_{i_m}$ be the subsequence of the events of σ which comprises all the e_i's which have been inserted into the prefix (in other words, those which are not post-cut-off events in the prefix being generated), and for every $j \leq m$, let $\sigma_{(j)}$ be the subsequence of σ comprising all the post-cut-off events e_i such that $i < i_j$. We then consider two cases.

Case 1: $m = 0$. Then we take $\sigma' \overset{\mathrm{df}}{=} \sigma$ as the sequence of choices for the **choose** operator in the main loop of the basic algorithm. In this case the base algorithm executes the instruction $pe \leftarrow pe \setminus \{e\}$ for each event e in σ', whereas the slicing algorithm executes the instruction $pe \leftarrow pe \setminus Sl$, which has exactly the same effect.

Case 2: $m > 0$. Then we take $\sigma' \overset{\mathrm{df}}{=} \sigma_{(1)} e_{i_1} \ldots \sigma_{(m)} e_{i_m}$ as the sequence of choices for the **choose** operator in the main loop of the basic algorithm. The sequences of instructions performed by the algorithms in this case is shown in figure 3. One can see that the subsequences of instructions modifying the variables Unf_Σ and cut_off are the same for both sequences. Moreover, the value of pe is determined exclusively by the last call to PotExt, which is also the same. Therefore, the final values of the corresponding variables in the two algorithms are equal.

This holds for each slicing iteration and the corresponding (several) basic iterations. The basic algorithm always terminates ([6]); thus both algorithms eventually generate isomorphic prefixes (note that the number of slicing iterations is never greater than the number of basic ones, so the slicing algorithm terminates when the basic one does).

The only thing which still needs to be shown is that the condition (*) imposed on SLICES(pe) ensures that this choice can be guided in the way given by σ', without violating the requirement to choose minimal w.r.t. $\lhd$ events from pe.

Suppose that the basic algorithm, having processed the first $k-1$ elements $\sigma'_1, \ldots, \sigma'_{k-1}$ of σ', has found out that σ'_k cannot be chosen by the **choose** operator. This cannot happen because $\sigma'_k \notin pe'_1$, where pe'_1 is the value of the variable pe when we would like the algorithm to choose σ'_k. Indeed, $\sigma'_k \in Sl \subseteq pe'_0$, where pe'_0 is the value of the variable pe at the beginning of the l'-th iteration. Moreover, σ' contains no repetitions of the events from $\hat{\sigma}$, the only ones from σ' which are being inserted into the prefix. This, and the fact that for all $S \subseteq$ POTEXT(Unf_Σ), POTEXT(Unf_Σ) $\setminus S \subseteq$ POTEXT(Unf^S_Σ), guarantees that σ'_k remains in pe while $\sigma'_1, \ldots, \sigma'_{k-1}$ are being processed. Therefore, the only reason why σ'_k cannot be chosen is that it is not minimal, i.e., there is an event $g \in pe'_1$ such that $g \lhd \sigma'_k$.

Suppose that $g \notin Sl$, i.e., either g was in pe at the time of choosing Sl in the slicing iteration being considered, but was not included in Sl, or it was generated and added to pe after the slice Sl had been chosen. In the latter case, there must be an event $f \in [g]$ which was present in pe at the moment of choosing Sl. In any case, according to (*), we have that $g \not\lhd e$, a contradiction. Therefore, $g \in Sl$. We now consider the four possible cases.

- Neither g nor σ'_k is a post-cut-off event. But the subsequence of non-post-cut-off events of σ' is arranged according to an order refining $\lhd$ and contains no repetitions. Therefore, since $g \lhd \sigma'_k$, g is to the left of σ'_k in σ'. After g had been processed and removed from pe, it was inserted into the prefix, so it could not have been generated again by a call to POTEXT. Hence $g \notin pe'_1$, a contradiction.
- $g = e_{i_j}$ is not a post-cut-off event and σ'_k is a post-cut-off event. Note that g occurs only once in σ'. Moreover, σ'_k is not in $\sigma'_{(j')}$ for all $j' \leq j$, since $g = e_{i_j} \lhd \sigma'_k$. Therefore, this occurrence of g is to the left of σ'_k. As in the previous case, after g had been processed and removed from pe, it was inserted into the prefix, so it could not have been generated again by a call to POTEXT. Hence $g \notin pe'_1$, a contradiction.
- g is a post-cut-off event, and $\sigma'_k = e_{i_j}$ is not a post-cut-off event. Then g occurs in $\sigma_{(j)}$, i.e., there is an occurrence of g at the left of σ'_k, and the algorithm does not call POTEXT between this occurrence of g has been removed from pe and the time σ'_k is being processed. Hence $g \notin pe'_1$, a contradiction.
- Both g and σ'_k are post-cut-off events. Then σ'_k is in some sequence $\sigma_{(j)}$. Since $g \lhd \sigma'_k$, it follows that g precedes σ'_k in $\sigma_{(j)}$, and the algorithm does not call POTEXT between this occurrence of g has been removed from pe and the time σ'_k is being processed. Again, this means that $g \notin pe'_1$, a contradiction.

Therefore, σ' is a sequence of choices which does not violate the requirement to choose a minimal event from pe, which completes the proof. ☐

Corollary 1 (Correctness). *If Σ is a bounded net system, then the algorithm in figure 2 always terminates and produces a finite and complete prefix of the unfolding of Σ.*

Proof. Follows directly from the correctness of the basic algorithm ([5, 6]) and lemma 1. ☐

Although the result given by lemma 1 is sufficient to prove the correctness of our algorithm, a somewhat stronger result, in fact, holds.

Theorem 2. *Let $Pref'_\Sigma$ and $Pref''_\Sigma$ be the prefixes of the unfolding of a bounded net system Σ, produced by arbitrary runs of the basic and slicing algorithms respectively. Then $Pref'_\Sigma$ and $Pref''_\Sigma$ are isomorphic.*

Proof. Follows directly from theorem 1 and lemma 1. ☐

Similarly as for the basic algorithm, the call to PotExt in the body of the main loop of the slicing algorithm can be replaced by a call UpdatePotExt(pe, Unf_{Σ}, Sl), which finds all events f such that f is an (Unf_{Σ}, e)-extension for some $e \in Sl$. The slicing version of the unfolding algorithm provides a basis for subsequent parallelisation, since now possible extensions are derived not from a single event, but rather from a set of events Sl; it turns out that computing UpdatePotExt(pe, Unf_{Σ}, Sl) can be effectively split into non-overlapping parts and distributed among several processors. Of course, for such scheme to work, we need to ensure that the sets in Slices(pe) do satisfy the condition (*) formulated at the beginning of this section.

3.1 The case of an adequate order refining $\lhd_m$

When $\lhd$ refines $\lhd_m$ (this is the case for $\lhd_{erv}$ and for most other orders proposed in literature), there is a simple scheme for choosing an appropriate set Slices(pe), by setting it to contain all non-empty closed w.r.t. $\lhd$ sets of events from pe whose local configurations have the minimal size. Then the condition (*) holds. Indeed, suppose that $e \in Sl \in$ Slices(pe) and g be an event in the unfolding of Σ. If $f \prec g$ for some $f \in pe$ then it is the case that $|[g]| > |[e]|$. Hence, since $\lhd$ refines $\lhd_m$, $g \not\lhd e$. Moreover, if $g \in pe \setminus Sl$ then $g \not\lhd e$ as Sl is a closed w.r.t. $\lhd$ set of events from pe.

Notice that in order to achieve better parallelisation, it is advantageous to choose large slices. Therefore, we can simply choose as a slice the set of *all* events from pe, whose size of the local configuration is minimal (note that this set is closed w.r.t. $\lhd$, and, therefore, is in Slices(pe)). With this scheme, we may simply consider pe as a sequence $Sl_1, Sl_2, \ldots$ of sets of events such that Sl_i contains the events whose local configurations have the size i (clearly, in each step of the algorithm there is only a finite number of non-empty Sl_i's). Thus inserting an event e into the queue is reduced to adding it into the set $Sl_{|[e]|}$, and choosing a slice in the main loop of the algorithm can be replaced by a call to *Front*(pe), returning the first non-empty set Sl_i in pe. Now all the required operations with the queue can be performed without comparisons of configurations at all.

The resulting algorithm is shown in figure 4. It uses the strategy of cut-offs 'in advance' outlined in [13], i.e., it checks the cut-off criterion as soon as a new possible extension is computed. This guarantees that at the beginning of each iteration of the main loop there are no cut-off events in *Front*(pe), and thus the restriction that the events from Sl must be processed in an order consistent with $\lhd$ can be safely left out. What is more, this strategy allows one to move the code computing the cut-off criterion into UpdatePotExt — the part of the algorithm which is executed in parallel.

When $\lhd$ is a total adequate order, each time two configurations are compared w.r.t. $\lhd$, one of the events becomes a cut-off event, i.e., the number of the performed comparisons is exactly $|E_{cut}|$ (rather than $O(|E| \log |E|)$ as in former implementations), and the algorithm achieves noticeable speedup even when only one processor is available (see section 4). One can reduce the number of comparisons even further, using the fact that the local configurations of the events which are already in the prefix are always less than those of newly computed possible extensions. But this would provide almost no speedup, since in this case the sizes of local configurations to be compared always differ, and so the comparisons are fast (we may assume that the size of the local configuration is attached to an event).

3.2 Parallelising the unfolding algorithm

As it was already mentioned, the events in Sl can be processed in any order. This leads to possibility of parallelising the unfolding algorithm when $|Sl| > 1$. There are only two kinds of dependencies between the events in Sl. First, the cut-off events must be handled properly; this part of the algorithm was explained in the previous section. Second, the (Unf_{Σ}, f)-extensions for $f \in Sl$ may have in their presets conditions produced by other events from Sl, inserted into

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

$Unf_\Sigma \leftarrow$ the empty branching process
$pe \leftarrow \{\perp\}$
$cut\_off \leftarrow \emptyset$
while $pe \neq \emptyset$ **do**
    $Sl \leftarrow Front(pe)$
    $pe \leftarrow pe \setminus Sl$
    for all $e \in Sl$ **do**
        add e and new instances of the places from $h(e)^\bullet$ to Unf_Σ
    for all $e \in Sl$ **do parallel**
        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

procedure UPDATEPOTEXT(pe, Unf_Σ, e)
$Ignore \leftarrow$ the set of events added into Unf_Σ after e
$Unf_\Sigma^{\lceil e \rceil} \leftarrow Unf_\Sigma$ with f and $f^\bullet$ removed, for all $f \in Ignore$
for all $(Unf_\Sigma^{\lceil e \rceil}, e)$-extensions g **do**
    if $\exists g' \in Unf_\Sigma \cup pe$ **such that** $Mark([g]) = Mark([g'])$ and $g' \lhd g$
    then $cut\_off \leftarrow cut\_off \cup \{g\}$
    else
        $pe \leftarrow pe \cup \{g\}$
        if $\exists g' \in Unf_\Sigma \cup pe$ **such that** $Mark([g]) = Mark([g'])$ and $g \lhd g'$
        then
            $cut\_off \leftarrow cut\_off \cup \{g'\}$
            $pe \leftarrow pe \setminus \{g'\}$

Fig. 4. A parallel algorithm for unfolding Petri nets.

the prefix before f. This can be dealt with by inserting all the events from Sl into Unf_Σ before the loop for computing possible extensions starts, and ignoring some of the inserted events in UPDATEPOTEXT (see figure 4).

Since UPDATEPOTEXT is the most time-consuming part of the algorithm, this strategy usually provides quite good parallelisation. In the most of our experiments, there were less then 200 iterations of the main loop, so the time spent on executing the sequential parts of the algorithm was neglible (this fact was confirmed by profiling the program). The first and the last few iterations usually allowed to execute 5–20 UPDATEPOTEXT's in parallel (which is already enough to provide quite good parallelism for most existing shared memory architectures), whereas the middle ones were highly parallel (from several hundreds up to several thousands tasks could potentially be executed in parallel). Thus the scalability of the algorithm is usually very good.

Of course, bad examples do exist, in particular those having 'long and narrow' unfoldings, e.g., the BUF100 net (section 4). But such examples are very rare in practice. Intuitively, they have only a small number of different partial order executions of the same length. This means that they have a very small number of conflicts and a low degree of concurrency (as for the BUF100 example, it has no conflicts at all and allows only few transitions to be executed concurrently). Our experiments show that as soon as first conflicts are encountered and added into the prefix being built, the number of events in $Front(pe)$ grows very quickly from step to step.

We implemented our algorithm on the shared memory architecture. It should not be hard to implement it on the distributed memory or even hybrid architecture, consisting of a network of multiprocessors. In that case, each node keeps a local copy of the built part of the prefix and synchronises it with the master node at the beginning of each iteration of the main loop. The master node is responsible for maintaining the queue of possible extensions, checking the cut-off criterion, and for distributing the work between the slaves; the slaves compute possible extensions and send them to the master.

3.3 Optimising computation of possible extensions

The idea of slicing the queue may result in developing a more efficient sequential algorithm for computing possible extensions due to merging common parts of the work in the spirit of the preset trees construction developed in [13, 14]. This direction is still to be investigated; here we present a simple improvement taking advantage of this idea, and leading to the final version of the algorithm.

Let $\mathcal{C} = \{f_1, \ldots, f_k\} \subseteq Sl$ be (Unf_Σ, e)-extensions for some event e in Unf_Σ. In order to find $(Unf_\Sigma^{\mathcal{C}}, f_i)$-extensions, the algorithm has to find the set of conditions in the already built part of the prefix, which are concurrent to f_i (note that only such conditions, together with those from $f_i^\bullet$, can be in the presets of (Unf_Σ, f_i)-extensions). This can be done by marking the conditions which are f_i's causal predecessors, or are in conflict with it, as unusable (the only nodes in the built part of the prefix which are the causal successors of f_i are the conditions in $f_i^\bullet$). Now, one can observe that the local configurations of all the f_i's include $[e]$ as a subset. Therefore, for any condition c, $c \prec e \Rightarrow c \prec f_i$, and $c \# e \Rightarrow c \# f_i$, i.e., the set of conditions which are the causal predecessors of e, or are in conflict with it, is a common subset of unusable conditions for all the f_i's, and so needs to be marked only once. Afterwards, this set can be extended to the sets of unusable conditions for each f_i separately. This allows one to generate possible extensions in the following way. As soon as we have a 'cluster' $\mathcal{C} \subseteq Sl$ of events which are among (Unf_Σ, e)-extensions for some e, we pass it as a whole to UPDATEPOTEXT (its interface must now be changed appropriately), which computes (Unf_Σ, f_i)-extensions for all $f_i \in \mathcal{C}$. The resulting algorithm is shown in figure 5, where it is assumed that Σ is a safe net system and $\lhd$ is an adequate total order.

This way of computing possible extensions is fully compatible with postset trees construction proposed in [13, 14] and, for some examples, it reduced the time needed for generating a complete prefix by more than 30%.

4 Experimental results

We used the unfolding algorithm described in [13, 14] as the basis for our parallel implementation and for the comparison. In order to experimentally confirm the correctness of the developed parallel implementation, we checked that the produced prefixes are isomorphic to those generated by the sequential version of the algorithm.[2] For this, a special utility for 'sorting' prefixes was developed, so that if two prefixes were isomorphic then after 'sorting' they become equal. It works in the following way:

1. Separate cut-off events, pushing them to the end.
2. Sort non-cut-off events according to $\lhd_{erv}$.
3. Separate post-cut-off conditions, pushing them to the end.
4. Sort non-post-cut-off conditions according to the following ordering: $c' \lessdot c''$ if $e' \lhd_{erv} e''$, or $e' = e''$ and $h(c') \ll h(c'')$, 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 e and e' are non-cut-off events, and that the of non-cut-off events of the prefix have already been sorted according to $\lhd_{erv}$ by this step.

[2] Note that due to theorem 1, two algorithms using the same adequate order produce isomorphic prefixes (provided that the implementations are correct).

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
$pe \leftarrow \{\bot\}$
$cut\_off \leftarrow \emptyset$
while $pe \neq \emptyset$ **do**
  /* pe is implemented as a sequence of sets */
  $Sl \leftarrow Front(pe)$
  $pe \leftarrow pe \setminus Sl$
  for all $e \in Sl$ **do**
    add e and new instances of the places from $h(e)^\bullet$ to Unf_Σ
  /* partitioning of Sl; the events in $\mathcal{C}_i$ are (Unf_Σ, e_i)-extensions of some event e_i */
  $SP \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$
  for all $\mathcal{C} \in SP$ **do parallel**
    UPDATEPOTEXT(pe, Unf_Σ, $\mathcal{C}$)
for all $e \in cut\_off$ **do**
  add e and new instances of the places from $h(e)^\bullet$ to Unf_Σ

procedure UPDATEPOTEXT(pe, Unf_Σ, $\mathcal{C}$)
$e \leftarrow$ the event whose extensions are in $\mathcal{C}$
$Cond \leftarrow \{c$ is a condition of $Unf_\Sigma \mid \neg(c \prec e \vee c \# e)\}$
for all $f \in \mathcal{C}$ **do**
  $Ignore \leftarrow$ the set of events added into Unf_Σ after f
  for all $t \in T$ **such that** an instance of t can be inserted after f **do**
    for all co-sets $C \subseteq Cond \setminus Ignore^\bullet$ **such that** f co $C \wedge h(C) \cup (h(f)^\bullet \cap {}^\bullet t) = {}^\bullet t$ **do**
      $g \leftarrow$ a new t-labelled event with the preset $C \cup \{c \in f^\bullet \mid h(c) \in {}^\bullet t\}$
      /* The following test can be implemented as one lookup in a hash table */
      if $\exists g' \in Unf_\Sigma \cup pe$ **such that** $Mark([g]) = Mark([g'])$
      then
        if $[g'] \lhd [g]$ **then** $cut\_off \leftarrow cut\_off \cup \{g\}$
        else
          $cut\_off \leftarrow cut\_off \cup \{g'\}$
          $pe \leftarrow (pe \cup \{g\}) \setminus \{g'\}$
      else $pe \leftarrow pe \cup \{g\}$

Fig. 5. A parallel algorithm (with clusters) for unfolding safe Petri nets ($\lhd$ is an adequate total order).

5. Sort the presets of the events (including the cut-offs) according to $\lessdot$.
6. Sort the cut-off events according to the following ordering: $e' \lessdot e''$ if ${}^\bullet e' \lessdot_{sl} {}^\bullet e''$, or ${}^\bullet e' = {}^\bullet e''$ and $h(e') \ll h(e'')$, 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 have already been sorted by this step.
7. Sort post-cut-off conditions according to $\lessdot$.
   Note that all events have already been sorted by this step.
8. Sort the postsets of the events (including the cut-offs) according to the $\lessdot$ ordering.
   Note that all conditions have already been sorted by this step.

This is an enhanced version of the approach described in [13, 14], the only difference is that we can no longer assume that the non-cut-off events in prefixes produced by our algorithm are sorted according to $\lhd_{erv}$, and therefore have to explicitly sort them (step 2).

Test cases The popular set of benchmark examples, collected by J.C. Corbett ([2]), K. McMillan, S. Melzer, and S. Römer was attempted[3] (this set was also used in [4, 8–11, 13, 14, 17]). Also we used the RND(m, n), SPA(n), and SPA(m, n) series described in [13, 14]. The experiments were conducted on a workstation with four *Pentium*^{TM} III/500MHz processors and 512M RAM.

The results of our experiments are summarised in table 1. The meanings of the columns are 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 sequential unfolder described in [13, 14]; the time spent by the parallel unfolder with different number N of working threads; the average/maximal number of independent tasks which may be performed in parallel on each iteration of the main loop (this coincides with the number of 'clusters' in SP, see figure 5). Although, due to the limited number of processors, we could not exploit all the arising parallelism in our experiments, this data shows the potential scalability of the problem.

It is interesting to note that the new algorithm with only one working thread ($N = 1$) works faster than the sequential unfolder described in [13, 14]. This is so because it performs much less comparisons of configurations (see section 3.1) and due to the improvement described in section 3.3.

One can see that our algorithm does not achieve linear speedup. This was a surprising discovery, since the potential parallelism (the last column in the table) is usually *very high*. Profiling shows that the program spends more than 95% of time in a function which neither acquires locks, nor performs system calls, so that the contention on locks cannot be the reason for such a slowdown. The only rational explanation we could think of is the bus contention: the mentioned function tries to find co-sets forming presets of possible extensions, exploring the build part of the prefix. It is a fairly large pointer-linked structure, and the processors have to intensively access the memory in a quite unsystematic way, so that the processors' caches often have to redirect the access to the RAM. Therefore, the processors are forced to content for the bus, and the program slows down. Since this explanation might seem superficial, we decided to establish that bus contention does reveal itself in practice, and the following experiment was performed. Several processors intensively read random locations in a large array and performed some fake computation with the fetched values. The total number of fetches was fixed and evenly distributed among them. In the absence of bus contention, the time spent by such a program would decrease linearly in the number of used processors, but we observed the degradation of speed similar to that shown by our unfolding algorithm. We expect that future generations of hardware will alleviate this problem, e.g., by increasing the bus frequency or by introducing a separate bus for each processor.

5 Conclusions

Experimental results indicate that the algorithm we proposed in this paper can achieve significant speedups, at least in theory. But this is still not enough for practical size problems, because the number of processors in shared memory multiprocessors is usually quite small. Therefore, generating unfoldings is still a bottleneck for the unfolding based verification of Petri nets. Our future research will aim at developing an effective implementation of this algorithm for the distributed-memory or hybrid architecture. Another promising area is the approach allowing non-local correspondent configurations, proposed in [8]. It sometimes allows to significantly reduce the size of complete prefixes. We plan to investigate if this idea can be efficiently implemented.

[3] We chose only those examples from this set whose unfolding time was large enough to be of some interest.

Acknowledgements

This research was supported by an ORS Awards Scheme grant ORS/C20/4 and by an EPSRC grant GR/M99293. The financial support of Academy of Finland (Projects 43963, 47754) and Tekniikan Edistämissäätiö is also gratefully acknowledged.

References

1. 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.
2. J. C. Corbett: *Evaluating Deadlock Detection Methods.* University of Hawaii at Manoa (1994).
3. J. Engelfriet: Branching processes of Petri Nets. *Acta Informatica* 28 (1991) 575–591.
4. J. Esparza and S. Römer: An Unfolding Algorithm for Synchronous Products of Transition Systems. Proc. of *CONCUR'99*, Springer-Verlag, Lecture Notes in Computer Science 1664 (1999) 2–20.
5. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *TACAS'96*, Margaria T., Steffen B. (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 87–106.
6. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* (2001) to appear.
7. J. Esparza and C. Schröter: Reachability Analysis Using Net Unfoldings. Proc. of *Workshop of Concurrency, Specification & Programming 2000 (CS&P'2000)*, H. D. Burkhard, L. Czaja, A. Skowron, and P. Starke, (Eds.). Informatik-Bericht 140, vol. 2. Humboldt-Universitat zu Berlin (2000) 255–270.
8. K. Heljanko: Minimizing Finite Complete Prefixes. Proc. of *Workshop Concurrency, Specification and Programming 1999 (CS&P'99)*, (1999) 83–95.
9. K. Heljanko: Deadlock and Reachability Checking with Finite Complete Prefixes. Technical Report A56, Laboratory for Theoretical Computer Science, HUT, Espoo, Finland (1999).
10. K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fundamentae Informaticae* 37 (1999) 247–268.
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. V. Khomenko and M. Koutny: LP Deadlock Checking Using Partial Order Dependencies. Proc. of *CONCUR'2000*, Palamidessi C. (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1877 (2000) 410–425.
13. V. Khomenko and M. Koutny: An Efficient Algorithm for Unfolding Petri Nets. Technical Report CS-TR-726, Department of Computing Science, University of Newcastle (2001).
14. V. Khomenko and M. Koutny: Towards An Efficient Algorithm for Unfolding Petri Nets. Proc. of *CONCUR'2001*, Larsen P.G., Nielsen M. (Eds.). Springer-Verlag, Lecture Notes in Computer Science (2001) to appear.
15. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *4th CAV*, Springer-Verlag, Lecture Notes in Computer Science 663 (1992) 164–174.
16. K. L. McMillan: *Symbolic Model Checking.* PhD thesis, CMU-CS-92-131 (1992).
17. 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]					
	$\|S\|$	$\|T\|$	$a/m\ \|^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*Seq*	*N=1*	*N=2*	*N=3*	*N=4*	$a/m\ \|SP\|$
Buf(100)	200	101	1.98/2	10101	5051	1	31	18	13	13	13	1.94/9
Byz(1,4)	504	409	3.33/30	42276	14724	752	246	183	110	84	78	135.84/896
Dme(7)	470	343	3.24/5	9542	2737	49	7	5	2	2	1	42.02/56
Dme(8)	537	392	3.24/5	13465	3896	64	16	12	6	5	4	55.54/72
Dme(9)	604	441	3.24/5	18316	5337	81	33	26	14	11	10	71.03/90
Dme(10)	671	490	3.24/5	24191	7090	100	61	49	28	21	19	88.47/110
Dme(11)	738	539	3.24/5	31186	9185	121	105	86	50	39	35	107.89/132
Dph(6)	57	92	1.98/2	14590	7289	3407	10	7	3	3	2	62.05/127
Dph(7)	66	121	1.98/2	74558	37272	19207	286	211	126	97	90	219.96/509
Elev(4)	736	1939	1.99/2	32354	16935	7337	73	42	25	19	17	204.58/964
Ftp(1)	176	529	1.98/2	178085	89046	35197	2820	1609	975	761	714	915.57/3249
Furn(3)	53	99	1.75/2	30820	18563	12207	30	15	9	7	5	91.83/264
Gasnq(4)	258	465	1.98/2	15928	7965	2876	19	11	6	5	4	110.94/284
Gasnq(5)	428	841	1.99/2	100527	50265	18751	884	553	334	259	243	529.16/1400
Gasq(4)	1428	2705	2/2	19864	9933	4060	30	18	11	7	6	138.25/493
Key(3)	129	133	1.98/2	13941	6968	2911	10	7	4	3	2	57.91/145
Key(4)	164	174	1.98/2	135914	67954	32049	935	806	485	379	354	427.27/1224
Mmgt(3)	122	172	1.95/2	11575	5841	2529	6	4	2	1	1	96.17/328
Mmgt(4)	158	232	1.95/2	92940	46902	20957	556	339	205	159	150	567.77/1992
Q(1)	163	194	1.89/2	16123	8417	1188	41	25	15	11	10	84.03/344
Rw(12)	63	313	2/2	98378	49177	45069	15	6	3	2	2	157.62/462
Sync(3)	106	270	2.21/4	28138	15401	5210	79	62	36	27	24	116.27/343
Rnd(5,8)	40	540	4.70/5	235600	56691	46559	68	51	29	22	19	386.81/1344
Rnd(5,9)	45	545	4.67/5	304656	72895	59840	113	90	53	41	37	447.62/1519
Rnd(5,10)	50	550	4.64/5	419946	98477	82279	175	144	85	66	61	474.97/1712
Rnd(5,11)	55	555	4.60/5	573697	132344	112310	267	227	134	104	99	526.03/1853
Rnd(5,12)	60	560	4.57/5	627303	145378	122465	351	297	178	140	131	557.76/1872
Rnd(5,13)	65	565	4.54/5	718762	166093	140147	453	382	232	183	172	539.40/1881
Rnd(5,14)	70	570	4.51/5	802907	185094	156417	546	471	284	225	215	584.35/1970
Rnd(5,15)	75	575	4.48/5	842181	195228	163722	665	567	345	274	259	605.35/1971
Rnd(5,16)	80	580	4.45/5	886158	206265	171957	787	674	413	329	312	623.24/2013
Rnd(5,17)	85	585	4.42/5	987605	229284	191576	942	822	503	404	382	607.82/2066
Rnd(5,18)	90	590	4.39/5	1025166	239069	198524	1091	956	584	469	448	614.02/2114
Rnd(10,3)	30	530	9.49/10	1415681	153628	144548	84	46	26	19	17	633.79/2095
Rnd(10,4)	40	540	9.33/10	2344821	252320	237000	216	137	80	61	55	720.00/2415
Rnd(10,5)	50	550	9.18/10	2485903	271083	250600	354	236	140	108	101	751.15/2406
Rnd(10,6)	60	560	9.04/10	2535070	280560	255010	526	360	216	168	159	746.97/2343
Rnd(10,7)	70	570	8.89/10	2537646	285323	254767	724	510	306	242	229	707.14/2323
Rnd(10,8)	80	580	8.76/10	2534970	289550	254000	953	681	411	327	312	786.64/2116
Rnd(15,2)	30	530	14.21/15	1836868	135307	128358	70	17	9	6	5	664.40/1979
Rnd(15,3)	45	545	13.84/15	3750719	271074	255560	270	128	74	56	49	895.59/2141
Rnd(15,4)	60	560	13.50/15	3787575	280560	257515	487	277	162	128	117	874.85/2301
Rnd(15,5)	75	575	13.17/15	3795090	288075	257515	776	480	286	228	214	819.19/2472
Rnd(20,2)	40	540	18.59/20	4744587	256197	245750	176	42	21	14	11	841.25/2797
Rnd(20,3)	60	560	17.96/20	5040080	280560	260020	447	203	118	90	82	842.21/2237
Rnd(20,4)	80	580	17.38/20	5050100	290580	260020	825	456	271	213	201	865.03/2510
Spa(7)	167	241	5.38/8	52516	18712	9937	81	48	28	21	19	169.27/629
Spa(8)	190	385	6.82/9	216772	76181	45774	1005	603	362	280	264	480.21/2002
Spa(9)	213	657	8.35/10	920270	320582	209449	13512	8066	4854	3750	3537	1669.04/6953
Spa(2,3)	144	161	4.20/7	15690	5682	2512	8	4	2	2	1	71.11/232
Spa(2,4)	190	385	6.82/9	253219	88944	52826	1412	872	524	406	382	614.64/2455
Spa(3,2)	144	161	4.20/7	15690	5682	2512	8	4	2	2	1	71.11/232
Spa(3,3)	213	657	8.35/10	1142214	398850	256600	22011	13565	8171	6317	5943	2166.84/8928

Table 1. Experimental results.