Parallel LTL-X Model Checking of High-Level Petri Nets Based on Unfoldings

Claus Schröter[1] and Victor Khomenko[2]

[1] Institut für Formale Methoden der Informatik
Universität Stuttgart, Germany
schroeter@fmi.uni-stuttgart.de
[2] School of Computing Science
University of Newcastle upon Tyne, UK
Victor.Khomenko@ncl.ac.uk

Abstract. We present an unfolding-based approach to LTL-X model-checking of high-level Petri nets. It is based on the method proposed by Esparza and Heljanko for low-level nets [4, 5] and a state of the art parallel high-level net unfolder described in [15, 13]. We present experimental results comparing our approach to the one of [4, 5] and the model-checker Spin [12].

1 Introduction

The main drawback of model-checking (see, e.g., [2]) 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 alleviate this problem, a number of techniques have been proposed. They 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 a reduced (though sufficient for a given verification task) representation (e.g., *abstraction* and *partial order reduction* techniques [2]). Among them, a prominent technique is McMillan's (finite prefixes of) Petri net unfoldings (see, e.g., [6, 16, 13, 17]). It relies on the partial order view of concurrent computation, and represents system states implicitly, using an acyclic net. More precisely, given a Petri net Σ, the unfolding technique aims at building a labelled acyclic net Unf_Σ (a *prefix*) satisfying two key properties [6, 16, 13]:

- *Completeness.* Each reachable marking of Σ is represented by at least one 'witness', i.e., a marking of Unf_Σ reachable from its initial marking. Similarly, for each possible firing of a transition at any reachable state of Σ there is a suitable 'witness' event in Unf_Σ.
- *Finiteness.* The prefix is finite and thus can be used as an input to model-checking algorithms, e.g., those searching for deadlocks.

A finite complete prefix satisfying these two properties can be used for model-checking as a condensed (symbolic) representation of the state space of a system.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 109–121, 2004.
© Springer-Verlag Berlin Heidelberg 2004

Indeed, it turns out that often such prefixes are exponentially smaller than the corresponding reachability graphs, especially if the system at hand exhibits a lot of concurrency. At least, they are never larger than the reachability graphs [6, 16, 13].

The unfolding techniques and algorithms described in [6–8, 10, 13, 14, 16–18, 20] help to alleviate the state space explosion problem when model-checking low-level Petri nets. Moreover, the construction of the prefix can be efficiently parallelised [11, 13]. However, the applicability of these techniques is restricted by the fact that low-level Petri nets are a very low-level formalism, and thus inconvenient for practical modelling. Therefore, it is highly desirable to generalise this technique to more expressive formalisms, such as high-level (or 'coloured') Petri nets. This formalism allows one to model in quite a natural way many constructs of high-level specification languages used to describe concurrent systems (see, e.g., [1]). Though it is possible to translate a high-level net into a low-level one and then unfold the latter, it is often the case that the intermediate low-level net is exponentially larger than the resulting prefix. Moreover, such a translation often completely destroys the structure present in the original model. In [15, 13], an approach allowing one to build a prefix directly from a high-level net, thus avoiding a potentially expensive translation into a low-level net, was described. Experiments demonstrated that this method is often superior to the traditional one, involving the explicit construction of an intermediate low-level net.

Petri net unfolding prefixes have been used for verification of simple safety properties, such as deadlock freeness, mutual exclusion and various kinds of reachability analysis [10, 13, 17, 18]. (The LTL model checking algorithm proposed in [22] is quite complicated and a corrected version of the approach requires memory exponential in the size of the prefix in the worst case [4, 5].) Recent work [4, 5, 8] suggested a method for checking LTL-X properties of low-level Petri nets. It uses a particular non-standard way of synchronising a Petri net with a Büchi automaton, which preserves as much concurrency as possible, in order to avoid a blow up in the size of the resulting prefix.

In this paper, we build on the ideas of [4, 5, 11, 13, 15] and propose a parallel algorithm for verification of LTL-X properties based on high-level Petri net unfoldings. To our knowledge, no such an algorithm existed before.

2 Basic Notions

We use *M-nets* [1] as the main high-level Petri net model, as we believe that it is general enough to cover many other existing relevant formalisms. The full description of M-nets can be found in [1]; here we give only an informal introduction, omitting those details which are not directly needed for the purposes of this paper. In particular, [1] devotes a lot of attention to the composition rules for M-nets, which are relevant only at the construction stage of an M-net, but not for model-checking an already constructed one. We assume the reader is familiar with the standard notions of the Petri nets theory, such as *places*, *transitions*, *arcs*, *presets* and *postsets* of places and transitions, *marking* of a Petri net, the *enabledness* and *firing* of a transition and marking *reachability* (see, e.g., [6]).

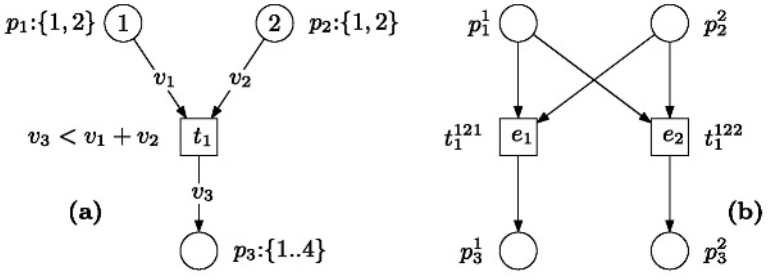

Fig. 1. An M-net system (a) and its unfolding (b).

2.1 High-Level Petri Nets

Let *Tok* be a (finite or infinite) set of elements (or 'colours') and *VAR* be a set of variable names, such that $Tok \cap VAR = \emptyset$. An *M-net* is a Petri net such that:

- Each of its places has a *type*, which is a subset of *Tok* and indicates the colours this place can contain. We assume that the types of all places are finite[1]. Valid markings of an M-net can be obtained by placing in each of its places a (possibly empty) multiset of tokens consistent with the type of the place.
- Each transition is labelled by a *guard*, which is a well-formed Boolean expression over $Tok \cup VAR$.
- Each arc is labelled with a multiset of variables from *VAR*.

For a transition t, we will denote by $VAR(t)$ the set of variables which appear in its guard and its incident arcs[2].

The enabling and firing rules of M-nets are as follows: when tokens flow along the incoming arcs of a transition t, they become bound to variables labelling those arcs, forming a (partial) mapping $\sigma : VAR(t) \to Tok$. If this mapping can be extended to a total mapping σ' (such an extension can be non-unique) in such a way that the guard of t evaluates to **true** and the values of the variables on the outgoing arcs are consistent with the types of the places these arcs point to, then t is called *enabled* and σ' is called a *firing mode* of t. An enabled transition can *fire*, consuming the tokens from its preset and producing tokens in places in its postset, in accordance with the values of the variables on the appropriate arcs given by σ'.

As an example, consider the M-net system shown in Figure 1(a). At the initial marking, t_1 can fire with the firing modes $\sigma' \stackrel{\text{df}}{=} \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 1\}$ or $\sigma'' \stackrel{\text{df}}{=} \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 2\}$, consuming the tokens from p_1 and p_2 and producing respectively the token 1 or 2 in p_3.

[1] In general, allowing infinite types yields a Turing-powerful model. Nevertheless, this restriction can be omitted in certain important cases [13, 15].

[2] If some variable appears in the guard of t but not on its incident arcs, it must be explicitly given a finite type.

2.2 Translation into Low-Level Nets

For each M-net system it is possible to build an 'equivalent' low-level one (the construction is given in [1, 15, 13]). Such a transformation is called 'unfolding' in [1], but since we use this term in a different meaning, we adopt the term 'expansion' instead. One can show that the reachability graphs generated by an M-net and its expansion are isomorphic, i.e., the expansion faithfully models the original M-net. This means that LTL properties of high-level nets can be verified by building its expansion. However, the disadvantage of this transformation is that it typically yields a very large net. Moreover, the resulting Petri net is often *unnecessarily* large, in the sense that it contains many unreachable places and many dead transitions. This is so because the place types are usually overapproximations, and the transitions of the original M-net system may have many firing modes, only few of which are realised when executing the M-net from its initial marking. Therefore, though the M-net expansion is a neat theoretical construction, it is often impractical.

2.3 Petri Net Unfoldings

The *finite complete prefix* of a low-level Petri net Σ is a finite acyclic net which implicitly represents all the reachable states of Σ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Σ, by successive firings of transition, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever Σ has at least one infinite run; however, if Σ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated without loss of information, yielding a finite and complete prefix.

One can show that the number of events in the complete prefix can never exceed the number of reachable states of Σ, for a precise statement see [6, 13, 16]. However, complete prefixes are often exponentially smaller than the corresponding reachability graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in reachability graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the reachability graph will be a 100-dimensional hypercube with 2^{100} nodes, whereas the complete prefix will coincide with the net itself.

In [13, 15] unfoldings of high-level Petri nets were defined, and the parallel unfolding algorithm proposed in [11, 13] was generalised to high-level nets. It turns out that the unfolding of a high-level net is isomorphic to the unfolding of its low-level expansion. (However, it can be constructed directly from the high-level net, without building this expansion.) Thus this approach is conservative in the sense that all the verification tools using traditional unfoldings as input can be reused with high-level ones. Figure 1(b) shows a finite and complete prefix of the M-net in Figure 1(a), coinciding with the finite and complete prefix of the expansion.

3 An Unfolding Approach to LTL-X Model-Checking

In [4, 5], an approach for unfolding based LTL-X model-checking of safe low-level Petri nets was proposed. It makes use of the automata-theoretic approach to model-checking [21]. A synchronised net system is constructed as the product of the original net system and a Büchi automaton accepting the negation of the property to be checked. Then the model-checking problem is reduced to the problem of detecting illegal ω-traces and illegal livelocks in the synchronised net system. Both problems are solved by exploiting finite complete prefixes of the unfolded synchronised net system. The main advantage of this approach over Wallner's [22] is its efficiency. Wallner first calculates a complete finite prefix and then constructs a graph, but the definition of the graph is non-trivial, and the graph grows exponentially in the size of the prefix [4, 5]. The approach of Esparza and Heljanko [4, 5] avoids the construction of the graph, but builds a larger prefix.

In this paper we follow the approach of [4, 5], but we are using *strictly safe* M-nets (i.e., ones which cannot put more than one token in a place) instead of safe low-level Petri nets. We will explain our approach by means of the example shown in Figure 2(a) (and in the following denoted by Υ). It is an M-net model of a buffer which can store at most 2 items (in places p_2 and p_4). To distinguish these two items they are represented by coloured tokens 0 and 1. The M-net contains 4 places and 3 transitions whereas its expanded low-level net contains 8 places and 8 transitions. One can see that if this example is scaled then its expansion grows exponentially in the size of the buffer and the cardinalities of place types, e.g., if the places have types $\{0..150\}$ then the expansion contains 604 places and 23103 transitions, whereas the prefix of such an M-net has just 10 conditions and 7 events (in this particular example place types do not influence the size of the prefix). Thus the prefix is often much smaller than the low-level expansion because the latter can contain many dead transitions and unreachable places which disappear in the prefix. Furthermore, for larger systems the expansion cannot be constructed within a reasonable time. The main advantage of our approach over the one of [4, 5] is that we unfold the M-net directly and do not have to build its expansion.

Let us assume that we want to check the property $\varphi \stackrel{\text{df}}{=} \Diamond\Box(p_2 \neq 0)$, i.e., "eventually the item 0 is never stored in the buffer cell p_2 again." First of all, we have to construct a Büchi automaton accepting $\neg\varphi$. This means that the property φ is violated by a system run in which $\neg(p_2 \neq 0)$ is **true** over and over again. The corresponding Büchi automaton $A_{\neg\varphi}$ is shown in Figure 2(b). We identify it with an M-net system (also denoted by $A_{\neg\varphi}$) which has a place for each state q of the automaton, with only the initial state q_0 having a token. (The type of all its places is $\{\bullet\}$.) For each transition (q, x, q') of $A_{\neg\varphi}$ the M-net has a transition (q, x, q'), where q and q' are the input and output places of the transition, and x is its guard. This M-net is shown in Figure 2(c), and in the following its places and transitions are called *Büchi places* and *Büchi transitions*, respectively.

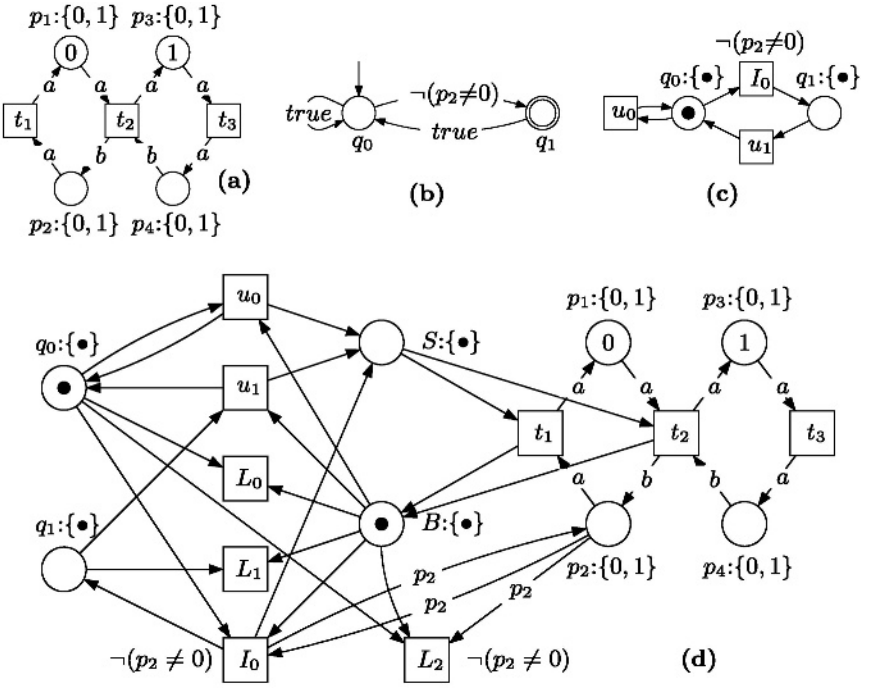

Fig. 2. M-net Υ: buffer of capacity 2 (a), Büchi automaton $A_{\neg\varphi}$ constructed for the property $\varphi \stackrel{\text{df}}{=} \Diamond\Box(p_2 \neq 0)$ (b), the M-net corresponding to $A_{\neg\varphi}$ (c), and the product net $\Upsilon_{\neg\varphi}$ (d).

In the next step a *synchronised M-net system* $\Upsilon_{\neg\varphi}$ is constructed as the product of Υ and $A_{\neg\varphi}$. A standard approach would synchronise the system and the Büchi automaton on all transitions. A drawback of such a synchronisation is that it completely sequentialises the system. Since a strength of the unfolding technique is to exploit concurrency, such a product net would not be very suitable for unfolding based verification techniques. Therefore, we use the synchronisation construction of [4, 5] generalised to high-level nets. In order to exploit concurrency, Υ and $A_{\neg\varphi}$ are only synchronised on those transitions of Υ which 'touch' the places that appear in the atomic propositions of φ. In the following such transitions are called *visible*. For our example this means that Υ and $A_{\neg\varphi}$ only synchronise on the visible transitions t_1 and t_2, because they touch the place p_2 which is the only place referred to in φ.

The resulting product net $\Upsilon_{\neg\varphi}$ is shown in Figure 2(d). Unlabelled arcs denote that only • can flow along them.

Construction of $\Upsilon_{\neg\varphi}$:

1. Put Υ and $A_{\neg\varphi}$ side by side.
2. Connect each Büchi transition with Υ in such a way that it 'observes' the places whose names appear in the guard of the transition. Let (q, x, q') be

a Büchi transition. For each place p which appears in the guard x add arcs from p to (q, x, q') and back; both arcs are labelled with "p". In our example this rule only applies to I_0 because it is the only Büchi transition whose guard refers to p_2. Therefore I_0 and p_2 are connected by arcs which are labelled with p_2.

3. Add *scheduler places* B and S (forming a flip-flop) in such a way that:
   - initially $A_{\neg\varphi}$ can make a step, and all the *visible* transitions of Υ are disabled.
   - after a step of $A_{\neg\varphi}$ only Υ can make a step.
   - after Υ has made a *visible* step, $A_{\neg\varphi}$ can make a step, and until that all the visible transitions of Υ are disabled.

   The intuition behind B and S is that $A_{\neg\varphi}$ (respectively, Υ) can make a step if there is a token on B (respectively, S). The types of these places are $\{\bullet\}$. B is connected by arcs going from B to every Büchi transition (u_0, u_1 and I_0), and going from every visible transition (t_1 and t_2) to B. S is connected by arcs going from S to every visible transition (t_1 and t_2), and going from every Büchi transition (u_0, u_1 and I_0) to S. Only $\bullet$ can flow along the new arcs. Initially, B contains $\bullet$ and S is empty, which allows $A_{\neg\varphi}$ to observe the initial marking of Υ.

Taking a look at the construction of $\Upsilon_{\neg\varphi}$ so far (see Figure 2(d) without transitions L_0, L_1, and L_2) it is clear that an infinite transition sequence that touches the accepting Büchi place q_1 infinitely often violates the property φ because in this case p_2 stores 0 over and over again. To detect such system runs we introduce the set $I \stackrel{\text{df}}{=} \{t \mid t^\bullet \text{ contains an accepting place of } A_{\neg\varphi}\}$ of all the transitions putting a token into an accepting Büchi place.

Now we claim that an infinite transition sequence of $\Upsilon_{\neg\varphi}$ which is fireable from the initial marking and contains infinitely many occurrences of I-transitions violates φ. In the following such sequences are called *illegal ω-traces*.

Unfortunately, not every sequence that violates the property φ is detected by illegal ω-traces because we synchronise just on the *visible* transitions. Consider a system that contains amongst others a "loop" of *invisible* transitions. Since we just synchronise on the visible transitions this "loop" would not take part in the synchronisation at all. Suppose the system reaches a marking M from which it is possible to fire a loop of invisible transitions. That is, the system would fire an infinite sequence of transitions without $A_{\neg\varphi}$ being able to observe it. Thus one can imagine a scenario where φ is violated but $A_{\neg\varphi}$ cannot detect this. Therefore, such a situation must be dealt with separately.

Since the firing of invisible transitions cannot change the marking of the *visible* places (i.e., those appearing in the atomic propositions of φ), as well as the places of the scheduler and $A_{\neg\varphi}$, we simply have to check for each marking enabling an infinite sequence of invisible transitions the existence of an accepting run of $A_{\neg\varphi}$ restricted only to those Büchi transitions which are enabled by this marking. (In such a case φ is violated.)

Formally, we augment the product net $\Upsilon_{\neg\varphi}$ with a set L of *livelock monitors* in such way that a new transition $t \in L$ iff there exists a place q in $A_{\neg\varphi}$ such that

$q \in t^\bullet$ and, starting from q, the $A_{\neg\varphi}$ accepts an infinite sequence of transitions. Then the situation described above can be formalised by an *illegal livelock*, which is an infinite firing sequence of the form $\tau t^\sigma \tau'$ such that $M_0 \xrightarrow{\tau t^\sigma} M \xrightarrow{\tau'}$, $t \in L$ and τ' contains only invisible transitions. A formal description for adding L-transitions to $\Upsilon_{\neg\varphi}$ is given in [5].

Theorem 1 (LTL-X Model-Checking of M-Nets). *Let Υ be a strictly safe M-net system whose reachable markings are pairwise incomparable with respect to set inclusion*[3]*, and φ be an LTL-X formula. It is possible to construct a product net system $\Upsilon_{\neg\varphi}$ satisfying the following properties:*

- *Υ satisfies φ iff $\Upsilon_{\neg\varphi}$ has neither illegal ω-traces nor illegal livelocks.*
- *The input and output places of the invisible transitions are the same in Υ and $\Upsilon_{\neg\varphi}$* [4]*.*

From [4, 5] we know that the number of L-transitions can grow exponentially in the size of the M-net system Υ, and so inserting them explicitly into the product net $\Upsilon_{\neg\varphi}$ would seriously hamper this approach. Therefore we follow the trick of [4, 5] and generate such transitions on-the-fly during the unfolding procedure. In order to do so we extend the construction of the product net $\Upsilon_{\neg\varphi}$ by the following step:

4. For each Büchi transition, add a copy of it with exactly the same guard and preset, but empty postset. (In our example, these new copies of u_0, u_1 and I_0 are L_0, L_1 and L_2, respectively.) All these new copies form a subset of transitions which are *candidates* to become livelock monitors (*L-transitions*) during the unfolding procedure.

The unfolding procedure makes use of these candidates in such a way that every time when an L-labelled event e could be inserted into the prefix it is checked whether starting from the marking M which is reached by firing e $A_{\neg\varphi}$ accepts an infinite run. If this is the case then the L-event is inserted into the prefix (otherwise it is not), and its preset is expanded to the whole marking M, and its postset is a copy of the preset with instances of the visible, scheduler and Büchi places removed. This guarantees that from this marking only invisible transitions can be fired in the future.

The check whether $A_{\neg\varphi}$ can start an accepting run from M restricted to the Büchi places can be done as follows. Let q be the Büchi place marked under M. We remove from $A_{\neg\varphi}$ all the transitions whose guards are evaluated to **false** under M (and thus cannot fire). Now, in order to check whether there exists an accepting run of $A_{\neg\varphi}$ it is enough to search for a strongly connected component in the obtained graph which is reachable from q and contains at least one I-transition.

[3] The latter condition is technical and can be removed [5].
[4] And thus much concurrency is preserved.

4 Tableaux System

We showed in Section 3 that the model-checking problem for LTL-X can be reduced to the problem of finding illegal ω-traces and illegal livelocks. In [4, 5] these problems are solved by considering the prefix as a "distributed" tableaux, in which the conditions are "facts" and the events represent "inferences", with the cut-off events being the *terminals* of the tableaux. We follow this approach, and informally describe how illegal ω-traces and illegal livelocks are detected.

The tableaux $\mathcal{T}$ for $\Upsilon_{\neg\varphi}$ is shown in Figure 3. It contains three terminals, e_{15}, e_{16}, and e_{17}, but for the moment let us concentrate on e_{17}. Since it is a cut-off event of the prefix, there exists a partially ordered execution (po-execution) C ($C = \{e_2\}$ in our example) with the same marking $\{q_0, S, p_1^0, p_3^1\}$ of $\Upsilon_{\neg\varphi}$ as the po-execution $[e_{17}] = \{e_1, \ldots, e_8, e_{12}, e_{14}, e_{17}\}$, where $[e]$ is defined as the minimal (w.r.t. $\subset$) causally closed set of events containing e. Since $C \subset [e_{17}]$, the execution $[e_{17}] \setminus C = \{e_1, e_3, \ldots, e_8, e_{12}, e_{14}, e_{17}\}$ starts and ends at the same marking $\{q_0, S, p_1^0, p_3^1\}$ of $\Upsilon_{\neg\varphi}$, i.e., it can be repeated arbitrarily many times. Moreover, it contains an I-event e_{12}, and thus an illegal ω-trace $\tau\tau'^{\omega}$ is detected, where $\tau \stackrel{\text{df}}{=} u_0$ corresponds to C and $\tau' \stackrel{\text{df}}{=} t_3t_2t_3u_0t_1u_0t_2I_0t_1u_1$ corresponds to $[e_{17}] \setminus C$.

The way of detecting illegal livelocks is quite similar, but in this case a terminal e occurring after an L-event e_L is considered. This means that there exists a po-execution $C \subset [e]$ (which also contains e_L) such that the execution $[e] \setminus C$ starts and ends at the same marking. Now we can use the same argument as above to find an infinite trace $\tau\tau'^{\omega}$, but τ' will contain only transitions of $\Upsilon_{\neg\varphi}$ corresponding to the events occurring after e_L. Since an L-transition empties the scheduler places of tokens, and they are never put back, no visible or Büchi transition of $\Upsilon_{\neg\varphi}$ can be enabled after firing an L-transition. Thus τ' consists only of invisible transitions, i.e., an illegal livelock is detected. (See [4, 5, 10] for more details.)

5 Experimental Results

In this section we present experimental results for the verification of LTL-X properties. We compare our implementation (in the following denoted by PUNF) with the unfolding based LTL-X model-checker for low-level nets UNFSMODELS [5], and with the model-checker SPIN [12]. All experiments are performed on a Linux PC with a 2.4 GHz Intel(R) Xeon(TM) processor and 4 GB of RAM. All times are measured in seconds.

We used UNFSMODELS version 0.9 of 22nd of October 2003, and invoked it with the option *-l* (for LTL-X model-checking).

We used SPIN version 4.0.7 of 1st of August 2003, and invoked it with the options *-DMEMCNT=32* (to allow the use of all 4 GB of memory) and *-DNOFAIR* (as no fairness assumptions were needed). SPIN's partial order reductions were used in all cases (they are enabled by default).

We used PUNF version 7.03 (parallel). In order to have a fair competition against the other tools we invoked PUNF in all examples with the option *-N=1*

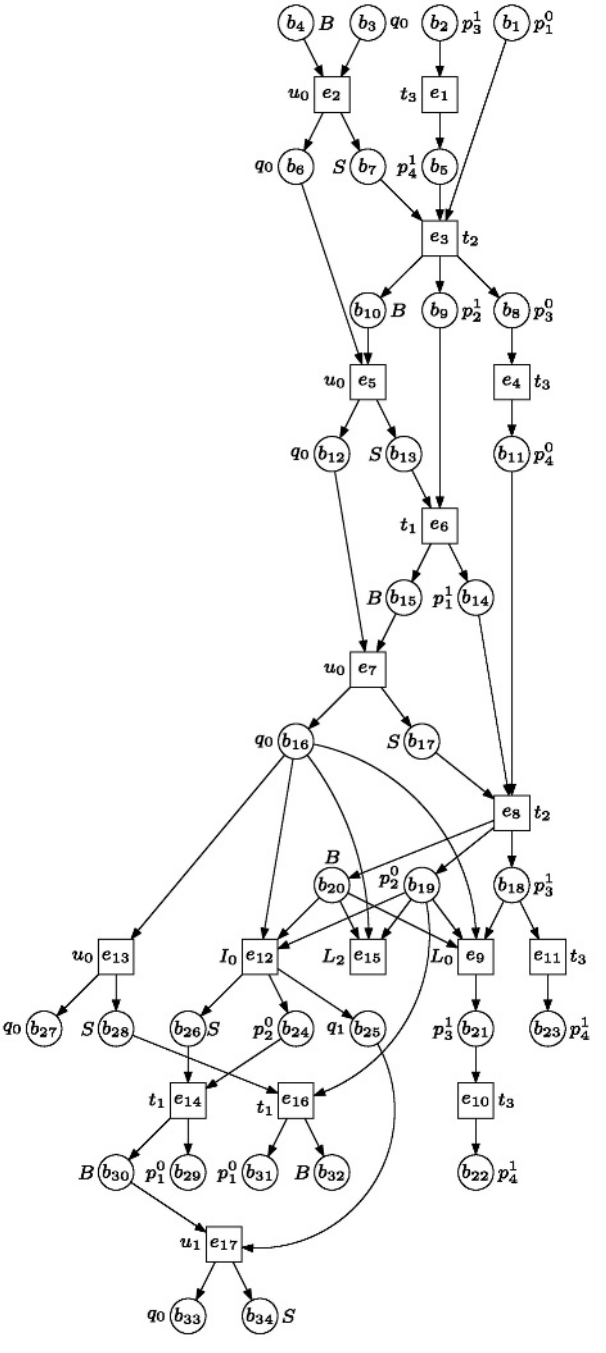

Fig. 3. The tableaux $\mathcal{T}$ for the product net $\Upsilon_{\neg\varphi}$. The colours of conditions are not shown when they are •. The firing modes of events are not shown as they can be derived from their pre- and postsets.

Table 1. Experimental results for LTL-X model-checking.

(a)

Net	Formula	Result	UnfSmdls	Spin	Punf
Abp	$\Box(p \to \Diamond q)$	True	0.19	0.01	0.08
Bds	$\Box(p \to \Diamond q)$	True	199	0.71	8.47
Dpd(7)	$\Diamond\Box\neg(p \wedge q \wedge r)$	True	507	2.14	7.25
Furnace(3)	$\Diamond\Box p$	True	1057	1.00	26.90
Gasnq(4)	$\Diamond\Box p$	True	240	0.14	8.46
Rw(12)	$\Box(p \to \Diamond q)$	True	2770	0.44	47.67
Ftp	$\Diamond\Box p$	True	>12000	3.99	836
Over(5)	$\Diamond\Box p$	True	66.01	0.44	0.12
Cyclic(12)	$\Box(p \to \Diamond q)$	True	0.38	11.25	0.08
Ring(9)	$\Diamond\Box p$	True	2.13	1.64	0.13
Dp(12)	$\Diamond\Box\neg(p \wedge q \wedge r)$	True	13.05	117	0.36
Ph(12)	$\Diamond\Box\neg(p \wedge q \wedge r)$	True	0.04	0.61	0.02
Com(15,0)	$\Box(p \to \neg\Diamond q)$	True	—	3.11	0.02
Par(5,10)	$\Box(p \to \neg\Diamond q)$	True	—	3.60	0.02

(b)

Net	Spin	Punf
Cyclic(15)	168	0.08
Cyclic(16)	478	0.07
Cyclic(17)	1601	0.10
Ring(12)	75.38	0.30
Ring(13)	274	0.50
Ring(14)	1267	0.85
Dp(13)	559	0.53
Dp(14)	2123	0.75
Ph(15)	16.69	0.01
Ph(18)	1570	0.01
Com(20,0)	232	0.02
Com(21,0)	686	0.03
Com(22,0)	2279	0.02
Par(6,10)	161	0.02
Par(7,10)	*mem*	0.04

Table 2. Experimental results for the parallel mode.

Net	Spin	Punf(1)	Punf(2)
Com(20,3)	*mem*	8.58	6.01
Com(22,3)	*mem*	11.51	8.51
Com(25,3)	*mem*	17.29	12.84
Par(20,100)	*mem*	8.60	4.84
Par(20,150)	*mem*	31.98	18.28
buf(20)	—	22.70	16.95
buf(25)	—	142.72	89.40

which means that the unfolding procedure is not parallelised, and Punf uses only one CPU.

The benchmarks (except Ph(n), Com(n, m) and Par(n, m)) are taken from J.C. Corbett [3]. We used only *deadlock-free* examples because our semantics for LTL-X for deadlocking systems is different from Spin's semantics and thus the results would not be comparable. These benchmarks together with the corresponding result files are available at
`http://www.fmi.uni-stuttgart.de/szs/people/schroeter/CAV/cav.tgz`.

In order to have a fair contest against Spin all systems were modelled in *Promela*, all of them (except Ph(n), Com(n, m) and Par(n, m)) by J.C. Corbett [3], but we scaled up some of the benchmarks. The M-nets used as the input for Punf were automatically derived (using a routine implemented by one of the authors) from Spin's automata representations of the *Promela* models. The low-level nets used as the input for UnfSmodels were obtained from the M-nets with the help of the hl2ll utility of the Pep tool [9].

Table 1(a) confirms that our approach outperforms UnfSmodels on all examples, with an increase of speed up to 550 times (Over(5) example). But we should mention that as noted in [5] UnfSmodels is more or less an academic prototype, whereas Punf is a high-performance unfolding engine. Com(15,0) and Par(5,10) could not been verified with UnfSmodels because either the Büchi automaton corresponding to the LTL-X formula for the low-level net (in the former case) or the low level net itself (in the latter case) could not be generated within reasonable time. Comparing Punf and Spin one can see that Spin

performs better on the examples over the line. In contrast, PUNF outperforms SPIN on the examples under the line. We wanted to investigate this further and therefore we scaled up these systems and checked them again.

The results are shown in Table 1(b). They seem to confirm that SPIN's verification time grows exponentially with the size of the benchmark, whereas the verification time of PUNF remains less than a second in all cases. All these systems have a high degree of concurrency, and the results show that our partial order technique handles these systems quite well. In contrast, SPIN's partial order reductions are not very efficient on these examples. In our current theory this seems to be related to a reduction condition for LTL-X model-checking which is known as the *reduction proviso* (see, e.g., [2, 19] for further details).

As it was already mentioned, our unfolding routine supports multiple threads running in parallel. To demonstrate this we performed some experiments on a two processor machine.

The results are shown in Table 2. PUNF(n) means that PUNF makes use of n processors. The results confirm that the process of LTL-X model checking can be parallelised quite efficiently. This speeds up the verification, in the best case up to n times. Also the results show that these examples are not verifiable with SPIN because it runs out of memory. (We did not verify the BUF(n) system with SPIN because it was modelled directly as an M-net).

6 Conclusions

We have presented an efficient algorithm for verification of LTL-X properties of high-level Petri nets. We followed an approach proposed by Esparza and Heljanko in [4, 5] for low-level nets, generalised it to M-nets, and implemented it using a state of the art parallel unfolding engine for high-level nets described in [15, 13]. Finally, we presented results showing that our implementation outperforms UNFSMODELS on all examples, and beats SPIN on some examples, mainly those having a high degree of concurrency. To the best of our knowledge, this is the first parallel algorithm for verification of LTL-X properties of high-level Petri nets based on unfoldings.

Acknowledgements

We would like to thank Javier Esparza and Keijo Heljanko for valuable discussions and comments on this topic. This research was supported by EPSRC grants GR/R64322 (AUTOMATIC SYNTHESIS OF DISTRIBUTED SYSTEMS) and GR/R16754 (BESST), and EC IST grant 511599 (RODIN).

References

1. E. Best, H. Fleischhack, W. Fraczak, R. Hopkins, H. Klaudel and E. Pelz: An M-net Semantics of $B(PN)^2$. Proc. of *STRICT'1995*, Berlin (1995) 85–100.
2. E. M. Clarke, O. Grumberg and D. Peled: *Model Checking*. MIT Press (1999).

3. J. C. Corbett: Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering* 22(3) (1996) 161–180.
4. J. Esparza and K. Heljanko: A New Unfolding Approach to LTL Model Checking. Proc. of *ICALP'2000*, LNCS 1853 (2000) 475–486.
5. J. Esparza and K. Heljanko: Implementing LTL Model Checking with Net Unfoldings. Proc. of *SPIN'2001*, LNCS 2057 (2001) 37–56.
6. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *TACAS'1996*, LNCS 1055 (1996) 87–106. Full version: FMSD 20(3) (2002) 285–310.
7. J. Esparza and C. Schröter: Net Reductions for LTL Model-Checking. Proc. of *Correct Hardware Design and Verification Methods*, LNCS 2144 (2001) 310–324.
8. J. Esparza and C. Schröter: Unfolding Based Algorithms for the Reachability Problem. *Fundamenta Informaticae* 47:(3,4) (2001) 231–245.
9. B. Grahlmann, S. Römer, T. Thielke, B. Graves, M. Damm, R. Riemann, L. Jenner, S. Melzer and A. Gronewold: PEP: Programming Environment Based on Petri Nets. Technical Report 14, Universität Hildesheim (1995).
10. K. Heljanko: *Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets*. PhD thesis, Helsinki University of Technology (2002).
11. K. Heljanko, V. Khomenko and M. Koutny: Parallelization of the Petri Net Unfolding Algorithm. Proc. of *TACAS'2002*, LNCS 2280 (2002) 371–385.
12. G. J. Holzmann: *The* SPIN *Model Checker*. Addison-Wesley (2003).
13. V. Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD Thesis, School of Computing Science, University of Newcastle upon Tyne (2003).
14. V. Khomenko and M. Koutny: Towards An Efficient Algorithm for Unfolding Petri Nets. Proc. of *CONCUR'2001*, LNCS 2154 (2001) 366–380.
15. V. Khomenko and M. Koutny: Branching Processes of High-Level Petri Nets. Proc. of *TACAS'2003*, LNCS 2619 (2003) 458-472.
16. V. Khomenko, M. Koutny and V. Vogler: Canonical Prefixes of Petri Net Unfoldings. Proc. of *CAV'2002*, LNCS 2404 (2002) 582–595. Full version: Acta Informatica 40(2) (2003) 95–118.
17. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *CAV'1992*, LNCS 663 (1992) 164–174.
18. S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. of *CAV'1997*, LNCS 1254 (1997) 352–363.
19. D. Peled: All from One, One for All — on Model Checking Using Representatives. Proc. of *CAV'1993*, LNCS 697 (1993) 409–423.
20. S. Römer: *Entwicklung und Implementierung von Verifikationstechniken auf der Basis von Netzentfaltungen*. PhD thesis, Technische Universität München (2000).
21. M. Y. Vardi and P. Wolper: An Automata Theoretic Approach to Automatic Program Verification. Proc. of *LICS'1986*, Cambridge (1986) 322–331.
22. F. Wallner: Model Checking LTL Using Net Unfoldings. Proc. of *CAV'1998*, LNCS 1427 (1998) 207–218.