Implementing LTL Model Checking with Net Unfoldings *

Javier Esparza[1] and Keijo Heljanko[2]

[1] Institut für Informatik, Technische Universität München, Germany
e-mail: esparza@in.tum.de
[2] Lab. for Theoretical Computer Science, Helsinki University of Technology, Finland
e-mail: Keijo.Heljanko@hut.fi

Abstract We report on an implementation of the unfolding approach to model-checking LTL-X recently presented by the authors. Contrary to that work, we consider an state-based version of LTL-X, which is more used in practice. We improve on the checking algorithm; the new version allows to reuse code much more efficiently. We present results on a set of case studies.

1 Introduction

Unfoldings [14,6,5] are a partial-order approach to the automatic verification of concurrent and distributed systems, in which partial-order semantics is used to generate a compact representation of the state space. For systems exhibiting a high degree of concurrency, this representation can be exponentially more succinct than the explicit enumeration of all states or the symbolic representation in terms of a BDD, thus providing a very good solution to the state-explosion problem.

Unfolding-based model-checking techniques for LTL without the next operator (called LTL-X in the sequel) were first proposed in [22]. A new algorithm with better complexity bounds was introduced in [3], in the shape of a tableau system. The approach is based on the automata-theoretic approach to model-checking (see for instance [20]), consisting of the following well-known three steps: (1) translate the negation of the formula to be checked into a Büchi automaton; (2) synchronize the system and the Büchi automaton in an adequate way to yield a composed system, and (3) check emptiness of the language of the composed system, where language is again defined in a suitable way.

In [3] we used an action-based version of LTL-X having an operator $\phi_1 \mathcal{U}^a \phi_2$ for each action a; $\phi_1 \mathcal{U}^a \phi_2$ holds if ϕ_1 holds until action a occurs, and immediately after ϕ_2 holds. Step (2) is very simple for this logic, which allowed us to concentrate on step (3), the most novel contribution of [3]. However, the state-based version of LTL-X is more used in practice. The first contribution of this paper is a solution to step (2) for this case, which turns out to be quite delicate.

* Work partially supported by the Teilprojekt A3 SAM of the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", the Academy of Finland (Projects 47754 and 43963), and the Emil Aaltonen Foundation.

The second contribution of this paper concerns step (3). In [3] we presented a two-phase solution; the first phase requires to construct one tableau, while the second phase requires to construct a possibly large set of tableaux. We propose here a more elegant solution which, loosely speaking, allows to merge all the tableaux of [3] into one while keeping the rules for the tableau construction simple and easy to implement.

The third contribution is an implementation using the `smodels` NP-solver [18], and a report on a set of case studies.

The paper is structured as follows. Section 2 contains basic definitions on Petri nets, which we use as system model. Section 3 describes step (2) above for the state-based version of LTL-X. Readers wishing to skip this section need only read (and believe the proof of) Theorem 1. Section 4 presents some basic definitions about the unfolding method. Section 5 describes the new tableau system for (3), and shows its correctness. Section 6 discusses the tableau generation together with some optimizations. Section 7 reports on the implementation and case studies, and Section 8 contains conclusions.

2 Petri nets

A *net* is a triple (P, T, F), where P and T are disjoint sets of *places* and *transitions*, respectively, and F is a function $(P \times T) \cup (T \times P) \to \{0, 1\}$. Places and transitions are generically called *nodes*. If $F(x, y) = 1$ then we say that there is an *arc* from x to y. The *preset* of a node x, denoted by ${}^\bullet x$, is the set $\{y \in P \cup T \mid F(y, x) = 1\}$. The *postset* of x, denoted by $x^\bullet$, is the set $\{y \in P \cup T \mid F(x, y) = 1\}$. In this paper we consider only nets in which every transition has a nonempty preset *and* a nonempty postset.

A *marking* of a net (P, T, F) is a mapping $P \to \mathbb{N}$ (where $\mathbb{N}$ denotes the natural numbers including 0). We identify a marking M with the multiset containing $M(p)$ copies of p for every $p \in P$. For instance, if $P = \{p_1, p_2\}$ and $M(p_1) = 1$, $M(p_2) = 2$, we write $M = \{p_1, p_2, p_2\}$.

A marking M *enables* a transition t if it marks each place $p \in {}^\bullet t$ with a token, i.e. if $M(p) > 0$ for each $p \in {}^\bullet t$. If t is enabled at M, then it can *fire* or *occur*, and its occurrence *leads to* a new marking M', obtained by removing a token from each place in the preset of t, and adding a token to each place in its postset; formally, $M'(p) = M(p) - F(p, t) + F(t, p)$ for every place p. For each transition t the relation $\xrightarrow{t}$ is defined as follows: $M \xrightarrow{t} M'$ if t is enabled at M and its occurrence leads to M'.

A 4-tuple $\Sigma = (P, T, F, M_0)$ is a *net system* if (P, T, F) is a net and M_0 is a marking of (P, T, F) (called the *initial marking* of Σ). A sequence of transitions $\sigma = t_1 t_2 \dots t_n$ is an *occurrence sequence* if there exist markings M_1, M_2, ... , M_n such that

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots M_{n-1} \xrightarrow{t_n} M_n$$

M_n is the marking reached by the occurrence of σ, which is also denoted by $M_0 \xrightarrow{\sigma} M_n$. A marking M is a *reachable marking* if there exists an occurrence

sequence σ such that $M_0 \xrightarrow{\sigma} M$. An *execution* is an infinite occurrence sequence starting from the initial marking. The *reachability graph* of a net system Σ is the labelled graph having the reachable markings of Σ as nodes, and the $\xrightarrow{t}$ relations (more precisely, their restriction to the set of reachable markings) as edges. In this work we only consider net systems with finite reachability graphs.

A marking M of a net is *n-safe* if $M(p) \leq n$ for every place p. A net system Σ is n-safe if all its reachable markings are n-safe. Fig. 1 shows a 1-safe net system.

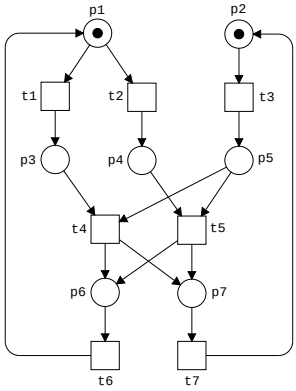

Figure 1. The net system Σ

Labelled nets. Let $\mathcal{L}$ be an alphabet. A *labelled net* is a pair (N, l) (also represented as a 4-tuple (P, T, F, l)), where N is a net and $l\colon P \cup T \to \mathcal{L}$ is a labelling function. Notice that different nodes of the net can carry the same label. We extend l to multisets of $P \cup T$ in the obvious way.

For each label $a \in \mathcal{L}$ we define the relation $\xrightarrow{a}$ between markings as follows: $M \xrightarrow{a} M'$ if $M \xrightarrow{t} M'$ for some transition t such that $l(t) = a$. For a finite sequence $w = a_1 a_2 \dots a_n \in \mathcal{L}^*$, $M \xrightarrow{w} M'$ denotes that for some reachable markings $M_1, M_2, \dots, M_{n-1}$ the relation $M \xrightarrow{a_1} M_1 \xrightarrow{a_2} M_2 \dots M_{n-1} \xrightarrow{a_n} M'$ holds. For an infinite sequence $w = a_1 a_2 \dots \in \mathcal{L}^\omega$, $M \xrightarrow{w}$ denotes that $M \xrightarrow{a_1} M_1 \xrightarrow{a_2} M_2 \dots$ holds for some reachable markings $M_1, M_2, \dots$.

The reachability graph of a labelled net system (N, l, M_0) is obtained by applying l to the reachability graph of (N, M_0). In other words, its nodes are the set

$$\{l(M) \mid M \text{ is a reachable marking}\}$$

and its edges are the set

$$\{l(M_1) \xrightarrow{l(t)} l(M_2) \mid M_1 \text{ is reachable and } M_1 \xrightarrow{t} M_2\}\,.$$

3 Automata Theoretic Approach to Model Checking LTL

We show how to modify the automata theoretic approach to model checking LTL [20] to best suit the net unfolding method.

We restrict the logic LTL by removing the next time operator X. We call this stuttering invariant fragment LTL-X. Given a finite set Π of atomic propositions, the abstract syntax of LTL-X is given by:

$$\varphi ::= \pi \in \Pi \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathcal{U} \varphi_2$$

The semantics is a set of ω-words over the alphabet 2^{Π}, defined as usual.

Given a 1-safe net system Σ with initial marking M_0, we identify the atomic propositions Π with a subset $Obs \subseteq P$ of *observable places* of the net system, while the rest of the places are called *hidden*. Each marking M determines a valuation of $\Pi = Obs$ in the following way: $p \in Obs$ is true at M if M puts a token in p. Now, an execution $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots$ of Σ satisfies φ iff the ω-word $M_0 M_1 \ldots$ satisfies φ. The net system Σ satisfies φ, denoted $\Sigma \models \varphi$, if every execution of Σ satisfies φ.

The approach. Let φ be a formula of LTL-X. Using well-known algorithms (see e.g. [8]) we construct a Büchi automaton $\mathcal{A}_{\neg\varphi}$ over the alphabet $2^{\Pi} = 2^{Obs}$ which accepts a word w iff $w \not\models \varphi$.

We define a 1-safe product net system $\Sigma_{\neg\varphi}$ from Σ and $\mathcal{A}_{\neg\varphi}$. $\Sigma_{\neg\varphi}$ can be seen as the result of placing Σ in a suitable environment, i.e., $\Sigma_{\neg\varphi}$ is constructed by connecting Σ to an environment net system through new arcs.

It is easy to construct a product net system with a distinguished set of transitions I such that Σ violates φ iff some execution of the product fires some transition of I infinitely often. We call such an execution an *illegal ω-trace*. However, this product synchronizes $\mathcal{A}_{\neg\varphi}$ with Σ *on all transitions*, which effectively disables all concurrency present in Σ. Since the unfolding approach exploits the concurrency of Σ in order to generate a compact representation of the state space, this product is not suitable, and so we propose a new one.

We define the set V of *visible transitions* of Σ as the set of transitions which change the marking of some observable place of Σ. Only these transitions will synchronize with the automaton. So, for instance, in order to check a property of the form $\Box(p \rightarrow \Diamond q)$, where p and q are places, we will only synchronize with the transitions removing or adding tokens to p and q. This approach is similar but not identical to Valmari's tester approach described in [19]. (In fact, a subtle point in Valmari's construction makes its direct implementation unsuitable for checking state based LTL-X.)

The price to pay for this nicer synchronization is the need to check not only for illegal ω-traces, but also for so-called illegal livelocks. The new product contains a new distinguished set of transitions L (for livelock). An *illegal livelock* is an execution of the form $\sigma_1 t \sigma_2$ such that $t \in L$ and σ_2 does not contain any visible transition. For convenience we use the notation $M_0 \xrightarrow{\sigma} M \xrightarrow{\tau}$ to denote this, and implicitly require that $\sigma = \sigma_1 t$ with $t \in L$ and that τ is an infinite sequence which only contains invisible transitions.

In the rest of the section we define $\Sigma_{\neg\varphi}$. Readers only interested in the definition of the tableau system for LTL model-checking can safely skip it. Only the following theorem, which is proved hand in hand with the definition, is necessary for it. Property (b) is what we win by our new approach: The environment only interferes with the visible transitions of Σ.

Theorem 1. *Let Σ be a 1-safe net system whose reachable markings are pairwise incomparable with respect to set inclusion.*[1] *Let φ be an LTL-X formula over the observable places of Σ. It is possible to construct a net system $\Sigma_{\neg\varphi}$ satisfying the following properties:*

(a) $\Sigma \models \varphi$ iff $\Sigma_{\neg\varphi}$ has neither illegal ω-traces nor illegal livelocks.
(b) The input and output places of the invisible transitions are the same in Σ and $\Sigma_{\neg\varphi}$.

Construction of $\Sigma_{\neg\varphi}$ We describe the synchronization $\Sigma_{\neg\varphi}$ of Σ and $\mathcal{A}_{\neg\varphi}$ in a semiformal but hopefully precise way. Let us start with two preliminaries. First, we identify the Büchi automaton $\mathcal{A}_{\neg\varphi}$ with a net system having a place for each state q, with only the initial state q^0 having a token, and a net transition for each transition (q, x, q'); the input and output places of the transition are q and q', respectively; we keep $\mathcal{A}_{\neg\varphi}$, q and (q, x, q') as names for the net representation, the place and the transition. Second, we split the executions of Σ that violate φ into two classes: *executions of type I*, which contain infinitely many occurrences of visible transitions, and *executions of type II*, which only contain finitely many. We will deal with these two types separately.

$\Sigma_{\neg\varphi}$ is constructed in several steps:

(1) Put Σ and (the net representation of) $\mathcal{A}_{\neg\varphi}$ side by side.
(2) For each observable place p add a *complementary place* (see [17]) $\overline{p}$ to Σ. $\overline{p}$ is marked iff p is not, and so checking that proposition p does not hold is equivalent to checking that the place $\overline{p}$ has a token. A set $x \subseteq \Pi$ can now be seen as a conjunction of literals, where $\overline{p} \in x$ is used to denote $p \in (\Pi \setminus x)$.
(3) Add new arcs to each transition (q, x, q') of $\mathcal{A}_{\neg\varphi}$ so that it "observes" the places in x.
This means that for each literal p ($\overline{p}$) in x we add an arc from p ($\overline{p}$) to (q, x, q') and an arc from (q, x, q') to p ($\overline{p}$). The transition (q, x, q') can only be enabled by markings of Σ satisfying all literals in x.
(4) Add a *scheduler* guaranteeing that:
- Initially $\mathcal{A}_{\neg\varphi}$ can make a move, and all *visible* moves (i.e., the firings of visible transitions) of Σ are disabled.
- After a move of $\mathcal{A}_{\neg\varphi}$, only Σ can make a move.
- After Σ makes a visible move, $\mathcal{A}_{\neg\varphi}$ can make a move and until that happens all visible moves of Σ are disabled.

[1] This condition is purely technical. Any 1-safe net system can be easily transformed into an equivalent one satisfying it by adding some extra places and arcs; moreover, the condition can be removed at the price of a less nice theory.

This is achieved by introducing two *scheduler places* s_f and s_s [22]. The intuition behind these places is that when s_f (s_s) has a token it is the turn of the Büchi automaton (the system Σ) to make a move. In particular, visible transitions transfer a token from s_s to s_f, and Büchi transitions from s_f to s_s. Because the Büchi automaton needs to observe the initial marking of Σ, we initially put one token in s_f and no tokens on s_s.

(5) Let I be a subset of transitions defined as follows. A transition belongs to I iff its postset contains a final state of $\mathcal{A}_{\neg\varphi}$.

Observe that since only moves of $\mathcal{A}_{\neg\varphi}$ and visible moves of Σ are scheduled, invisible moves can still be concurrently executed.

Let $\Sigma'_{\neg\varphi}$ be the net system we have constructed so far. The following is an immediate consequence of the definitions:

Σ has an execution of type I if and only if $\Sigma'_{\neg\varphi}$ has an illegal ω-trace.

We now extend the construction in order to deal with executions of type II. Let σ be a type II execution of Σ. Take the sequence of markings reached along the execution of σ, and project it onto the observable places. Since σ only contains finitely many occurrences of visible transitions, the result is a sequence of the form $O_0^0 O_0^1 \dots O_0^j O_1^0 O_1^1 \dots O_1^k O_2^0 \dots O_n^0 (O_n)^\omega$. (The moves from O_i to O_{i+1} are caused by the firing of visible transitions.)

We can split σ into two parts: a finite prefix σ_1 ending with the last occurrence of a visible transition (σ_1 is empty if there are no visible transitions), and an infinite suffix σ_2 containing only invisible transitions. Clearly, the projection onto the observable places of the marking reached by the execution of σ_1 is O_n

Since LTL-X is closed under stuttering, $\mathcal{A}_{\neg\varphi}$ has an accepting run

$$r = q_0 \xrightarrow{O_0} q_1 \xrightarrow{O_1} \dots \xrightarrow{O_{n-1}} q_n \xrightarrow{O_n} q_{n+1} \xrightarrow{O_n} q_{n+2} \dots$$

where the notation $q \xrightarrow{O} q'$ means that a transition (q, x, q') is taken such that the literals of x are true at the valuation given by O. We split this run into two parts: a finite prefix $r_1 = q_0 \xrightarrow{O_0} q_1 \dots q_{n-1} \xrightarrow{O_{n-1}} q_n$ and an infinite suffix $r_2 = q_n \xrightarrow{O_n} q_{n+1} \xrightarrow{O_n} q_{n+2} \dots$.

In the net system representation of $\mathcal{A}_{\neg\varphi}$, r_1 and r_2 correspond to occurrence sequences. By construction, the "interleaving" of r_1 and σ_1 yields an occurrence sequence τ_1 of $\Sigma'_{\neg\varphi}$.

Observe that reachable markings of $\Sigma'_{\neg\varphi}$ are of the form (q, s, O, H), meaning that they consist of a token on a state q of $\mathcal{A}_{\neg\varphi}$, a token on one of the places of the scheduler (i.e., $s \in \{s_s, s_f\}$), a marking O of the observable places, and a marking H of the hidden places. Let (q_n, s_f, O_n, H) be the marking of $\Sigma'_{\neg\varphi}$ reached after executing τ_1. (We have $s = s_f$ because the last transition of σ_1 is visible.) The following property holds: With q_n as initial state, the Büchi automaton $\mathcal{A}_{\neg\varphi}$ accepts the sequence O_n^ω. We call any pair (q, O) satisfying this property a *checkpoint* and define $\Sigma_{\neg\varphi}$ as follows:

(6) For each checkpoint (q, O) and for each reachable marking (q, s_f, O, H) of $\Sigma'_{\neg\varphi}$, add a new transition having all the places marked at (q, s_f, O, H) as preset, and all the places marked at O and H as postset. Let L (for *livelocks*) be this set of transitions.

The reader has possibly observed that the set L can be very large, because there can be many hidden markings H for a given marking O (exponentially many in the size of Σ). Apparently, this makes $\Sigma_{\neg\varphi}$ unsuitable for model-checking. In Sect. 6 we show that this is not the case, because $\Sigma_{\neg\varphi}$ need not be explicitly constructed.

Observe that after firing a L-transition no visible transition can occur anymore, because all visible transitions need a token on s_s for firing. We prove:

Σ has an execution of type II if and only if $\Sigma_{\neg\varphi}$ has an *illegal livelock*.

For the only if direction, assume first that σ is a type II execution of Σ. Let τ_1 be the occurrence sequence of $\Sigma_{\neg\varphi}$ defined above (as the "interleaving" of the prefix σ_1 of σ and the prefix r_1 of r). Further, let (q_n, s_f, O_n, H) be the marking reached after the execution of τ_1, and let t be the transition added in (6) for this marking. Define $\rho_1 = \tau_1$ and $\rho_2 = \sigma_2$. It is easy to show that $\rho_1 t \rho_2$ is an execution of $\Sigma_{\neg\varphi}$ and so an illegal livelock. For the if direction, let $\rho_1 t \rho_2$ be an illegal livelock of $\Sigma_{\neg\varphi}$, where t is an L-transition. After the firing of t there are no tokens in the places of the scheduler, and so no visible transition can occur again; hence, no visible transition of Σ occurs in ρ_2. Let σ_1 and σ_2 be the projections of ρ_1 and ρ_2 onto the transitions of Σ. It is easy to see that $\sigma = \sigma_1\sigma_2$ is an execution of Σ. Since σ_2 does not contain any visible transition, σ is an execution of type II.

4 Basic definitions on unfoldings

In this section we briefly introduce the definitions we needed to describe the unfolding approach. More details can be found in [6].

Occurrence nets. Given two nodes x and y of a net, we say that x is *causally related* to y, denoted by $x \leq y$, if there is a (possibly empty) path of arrows from x to y. We say that x and y are in *conflict*, denoted by $x\#y$, if there is a place z, different from x and y, from which one can reach x and y, exiting z by different arrows. Finally, we say that x and y are *concurrent*, denoted by $x \mathrel{co} y$, if neither $x < y$ nor $y < x$ nor $x\#y$ hold. A *co-set* is a set of nodes X such that $x \mathrel{co} y$ for every $x, y \in X$. *Occurrence nets* are those satisfying the following three properties: the net, seen as a directed graph, has no cycles; every place has at most one input transition; and, no node is in self-conflict, i.e., $x\#x$ holds for no x. A place of an occurrence net is *minimal* if it has no input transitions. The net of Fig. 2 is an infinite occurrence net with minimal places a, b. The *default initial marking* of an occurrence net puts one token on each minimal place an none in the rest.

Branching processes. We associate to Σ a set of *labelled* occurrence nets, called the *branching processes* of Σ. To avoid confusions, we call the places and transitions of branching processes *conditions* and *events*, respectively. The conditions and events of branching processes are labelled with places and transitions of Σ, respectively. The conditions and events of the branching processes are subsets from two sets $\mathcal{B}$ and $\mathcal{E}$, inductively defined as the smallest sets satisfying the following conditions:

- $\bot \in \mathcal{E}$, where $\bot$ is an special symbol;
- if $e \in \mathcal{E}$, then $(p, e) \in \mathcal{B}$ for every $p \in P$;
- if $\emptyset \subset X \subseteq \mathcal{B}$, then $(t, X) \in \mathcal{E}$ for every $t \in T$.

In our definitions of branching process (see below) we make consistent use of these names: The label of a condition (p, e) is p, and its unique input event is e. Conditions $(p, \bot)$ have no input event, i.e., the special symbol $\bot$ is used for the minimal places of the occurrence net. Similarly, the label of an event (t, X) is t, and its set of input conditions is X. The advantage of this scheme is that a branching process is completely determined by its sets of conditions and events. We make use of this and represent a branching process as a pair (B, E).

Definition 1. *The set of* finite branching processes *of a net system Σ with the initial marking $M_0 = \{p_1, \ldots, p_n\}$ is inductively defined as follows:*

- $(\{(p_1, \bot), \ldots, (p_n, \bot)\}, \emptyset)$ *is a branching process of Σ.*
- *If (B, E) is a branching process of Σ, $t \in T$, and $X \subseteq B$ is a co-set labelled by ${}^\bullet t$, then $(\, B \cup \{(p, e) \mid p \in t^\bullet\} \,,\, E \cup \{e\}\,)$ is also a branching process of Σ, where $e = (t, X)$. If $e \notin E$, then e is called a* possible extension *of (B, E).*

The set of branching processes of Σ is obtained by declaring that the union of any finite or infinite set of branching processes is also a branching process, where union of branching processes is defined componentwise on conditions and events. Since branching processes are closed under union, there is a unique maximal branching process, called the *unfolding* of Σ. The unfolding of our running example is an infinite occurrence net. Figure 2 shows an initial part. Events and conditions have been assigned identificators that will be used in the examples. For instance, the event $(t_1, \{(p_1, \bot)\})$ is assigned the identificator 1.

We take as partial order semantics of Σ its unfolding. This is justified, because it can be easily shown the reachability graphs of Σ and of its unfolding coincide. (Notice that the unfolding of Σ is a *labelled* net system, and so its reachability graph is defined as the *image* under the labelling function of the reachability graph of the *unlabelled* system.)

Configurations. A *configuration* of an occurrence net is a set of events C satisfying the two following properties: C is causally closed, i.e., if $e \in C$ and $e' < e$ then $e' \in C$, and C is conflict-free, i.e., no two events of C are in conflict. Given an event e, we call $[e] = \{e' \in E \mid e' \leq e\}$ the *local configuration* of e. Let *Min* denote the set of minimal places of the branching process. A configuration C of the branching process is associated with a marking

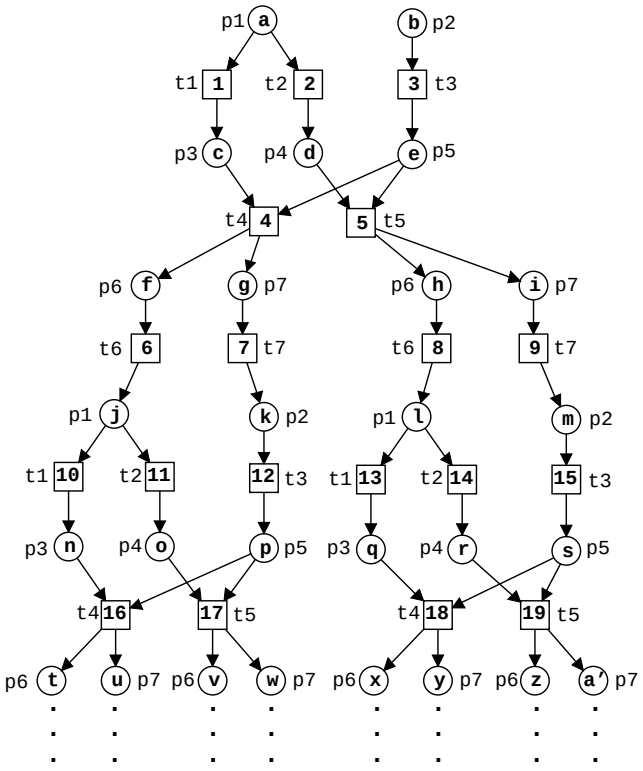

Figure 2. The unfolding of Σ

of Σ denoted by $Mark(C) = l((Min \cup C^\bullet) \setminus {}^\bullet C)$. The corresponding set of conditions associated with a configuration is called a *cut*, and it is defined as $Cut(C) = ((Min \cup C^\bullet) \setminus {}^\bullet C)$.

In Fig. 2, $\{1, 3, 4, 6\}$ is a configuration, and $\{1, 4\}$ (not causally closed) or $\{1, 2\}$ (not conflict-free) are not. A set of events is a configuration if and only if there is one or more firing sequences of the occurrence net (from the default initial marking) containing each event from the set exactly once, and no further events. These firing sequences are called *linearisations*. The configuration $\{1, 3, 4, 6\}$ has two linearisations, namely 1 3 4 6 and 3 1 4 6. All linearisations lead to the same reachable marking. For example, the two sequences above lead to the marking $\{p_1, p_7\}$. By applying the labelling function to a linearisation we obtain a firing sequence of Σ. Abusing of language, we also call this firing sequence a linearisation. In our example we obtain $t_1 t_3 t_4 t_6$ and $t_3 t_1 t_4 t_6$ as linearisations.

Given a configuration C, we denote by $\uparrow C$ the set of events of the unfolding $\{e \mid e \notin C \ \wedge \ \forall e' \in C : \neg(e \# e')\}$. Intuitively, $\uparrow C$ corresponds to the behavior of Σ from the marking reached after executing any of the linearisations of C. We call $\uparrow C$ the *continuation* after C of the unfolding of Σ. If C_1 and C_2 are two finite configurations leading to the same marking, i.e. $Mark(C_1) = M = Mark(C_2)$,

then $\uparrow C_1$ and $\uparrow C_2$ are isomorphic, i.e., there is a bijection between them which preserves the labelling of events and the causal, conflict, and concurrency relations (see [6]).

Adequate orders. To implement a net unfolding algorithm we need the notion of *adequate order* on configurations [6]. Given a configuration C of the unfolding of Σ, we denote by $C \oplus E$ the set $C \cup E$, under the condition that $C \cup E$ is a configuration satisfying $C \cap E = \emptyset$. We say that $C \oplus E$ is an *extension* of C. Now, let C_1 and C_2 be two finite configurations leading to the same marking. Then $\uparrow C_1$ and $\uparrow C_2$ are isomorphic. This isomorphism, say f, induces a mapping from the extensions of C_1 onto the extensions of C_2; the image of $C_1 \oplus E$ under this mapping is $C_2 \oplus f(E)$.

Definition 2. *A partial order $\prec$ on the finite configurations of the unfolding of a net system is an* adequate order *if:*

- *$\prec$ is well-founded,*
- *$C_1 \subset C_2$ implies $C_1 \prec C_2$, and*
- *$\prec$ is preserved by finite extensions; if $C_1 \prec C_2$ and $Mark(C_1) = Mark(C_2)$, then the isomorphism f from above satisfies $C_1 \oplus E \prec C_2 \oplus f(E)$ for all finite extensions $C_1 \oplus E$ of C_1.*

Total adequate orders for 1-safe Petri nets and for synchronous products of transition systems have been presented in [6,5].

5 Tableau System

We showed in Section 3 that the model checking problem for LTL-X can be solved by checking the existence of illegal ω-traces and illegal livelocks in $\Sigma_{\neg\varphi}$. In [3] these problems are solved using tableau techniques. A branching process can be seen as a "distributed" tableau, in which conditions are "facts" and events represent "inferences". For two conditions b and b', $b \; co \; b'$ models that the facts represented by b and b' can be simultaneously true. A tableau is constructed by adding new events (inferences) one by one following an adequate order; some events are declared as "terminals", and the construction of the tableau terminates when no new event can be added having no terminals among its predecessors.

The tableau systems of [3] require to construct a possibly large set of branching processes. Here we present a new tableau system consisting of one single branching process.[2]

[2] For the reader familiar with [3]: the L-transitions in the net system $\Sigma_{\neg\varphi}$ act as glue to connect a set of branching processes (the tableau components of [3]) together into one larger tableau.

An Adequate Order for LTL. We simplify the implementation of the tableau system by selecting a special adequate order. We use $\prec$ to denote the total adequate order defined for 1-safe Petri nets in [6]. We call an event corresponding to an L-transition an *L-event*. We define for a set of events C the function *before L-event* as $BL(C) = \{e \in C \mid [e] \setminus \{e\} \text{ contains no } L\text{-events}\}$. The function *after L-event* is defined correspondingly as $AL(C) = (C \setminus BL(C))$. We can now define our new adequate order.

Definition 3. *Let C_1 and C_2 be two finite configurations of the unfolding of the product net system $\Sigma_{\neg\varphi}$. $C_1 \prec_{LTL} C_2$ holds if*

- *$BL(C_1) \prec BL(C_2)$, or*
- *$BL(C_1) = BL(C_2)$ and $C_1 \prec C_2$.*

The adequate order $\prec_{LTL}$ is application specific in the sense that it is not an adequate order for an arbitrary net system Σ, but needs some special properties of the net system $\Sigma_{\neg\varphi}$. We have the following result.

Theorem 2. *The order $\prec_{LTL}$ is a total adequate order for finite configurations of the unfolding of $\Sigma_{\neg\varphi}$.*

See [4] for the proof.

New Tableau System. We first divide the unfolding of $\Sigma_{\neg\varphi}$ into two disjoint sets of events. Intuitively, the first set is used for the ω-trace detection part, and the second for the illegal livelock detection part. We define *part-I* to be the set of events e such that $[e]$ does not contain an L-event and *part-II* as the set of events which are not in part-I.

Definition 4. *An event e of the unfolding $\Sigma_{\neg\varphi}$ is a* terminal, *if there exists another event e' such that $Mark([e']) = Mark([e])$, $[e'] \prec_{LTL} [e]$, and one of the following two mutually exclusive cases holds:*

(I) $e \in$ part-I, and either
  (a) $e' < e$, or
  (b) $\neg(e' < e)$ and $\#_I[e'] \geq \#_I[e]$, where $\#_I C$ denotes the number of I-events in C.

(II) $e \in$ part-II, and either
  (a) $BL([e']) \prec_{LTL} BL([e])$, or
  (b) $BL([e']) = BL([e])$ and $\neg(e' \# e)$, or
  (c) $BL([e']) = BL([e])$, $e' \# e$, and $|[e']| \geq |[e]|$.

A tableau *$\mathcal{T}$ is a branching process (B, E) of $\Sigma_{\neg\varphi}$ such that for every possible extension e of (B, E) at least one of the immediate predecessors of e is a terminal. A terminal is* successful *if it is type (I)(a) and $[e] \setminus [e']$ contains an I-event, or it is of type (II)(b). All other terminals are* unsuccessful. *A tableau $\mathcal{T}$ is* successful *if it contains a successful terminal, otherwise it is* unsuccessful.

Loosely speaking, a tableau is a branching process which cannot be extended without adding a causal successor to a terminal.

We have the following result:

Theorem 3. *Let $\mathcal{T}$ be a tableau for $\Sigma_{\neg\varphi}$.*

- *$\Sigma_{\neg\varphi}$ has an illegal ω-trace iff $\mathcal{T}$ has a successful terminal of type I.*
- *$\Sigma_{\neg\varphi}$ has an illegal livelock iff $\mathcal{T}$ has a successful terminal of type II.*
- *$\mathcal{T}$ contains at most K^2 non-terminal events, where K is the number of reachable markings of $\Sigma_{\neg\varphi}$.*

See [4] for the proof.

6 Generating the Tableau

We describe an implementation of the tableau system of Sect. 5. The main goal is to keep the tableau generation as similar as possible to a conventional prefix generation algorithm [6]. In this way any prefix generation algorithm can be easily adapted to also perform LTL model checking.

The tableau generation algorithm (Algorithm 1) is almost identical to the main routine of a prefix generation algorithm. The changes are: an additional block of code devoted to generating the L-events dynamically; a different but easy to implement adequate order; a new cut-off detection subroutine. The main feature of the implementation is the efficient handling of L-transitions, which we discuss next.

Generating the L-transitions Dynamically. Recall that in the synchronization $\Sigma_{\neg\varphi}$ we can for each Büchi state q have as many L-transitions as there are reachable markings of the form (q, s_f, O, H) in the net system $\Sigma_{\neg\varphi}$. Clearly we can not explicitly generate them all due to efficiency reasons. Instead we generate a net system $\Sigma^s_{\neg\varphi}$ (s stands for *static*) in which this set of L-transitions (added by step (6) of the synchronization procedure in Section 3) is replaced by:

(6') Add for each Büchi transition $t = (q, x, q')$ in the net system $\Sigma'_{\neg\varphi}$ (i.e., the synchronization after steps (1)-(5) as defined in Sect. 3) a new transition t'. The preset of t' is equivalent to the preset of t and the postset of t' is empty. Let L (for *livelocks*) be this set of transitions.

We can now dynamically generate any of the (enabled) L-transitions of $\Sigma_{\neg\varphi}$. Namely, for a transition t corresponding to a reachable marking $M = (q, s_f, O, H)$ to be enabled in $\Sigma_{\neg\varphi}$, a transition t' (for some (q, x, q')) must be enabled in $\Sigma^s_{\neg\varphi}$ and the Büchi automaton must accept O^ω when q is given as the initial state. Loosely speaking we test the first label of the sequence using the transition t', and if this test succeeds we check whether O can be infinitely stuttered. (Using this construction it is easy to implement "no-care values" for selected atomic propositions by leaving them out of the preset of t'.) Now generating the postset of t from M is trivial.

Optimizations in Dynamic Creation. We can thus dynamically generate L-transitions for each reachable marking M as required. However, we can do better by using the net unfolding method. The main idea is to generate the unfolding of $\Sigma_{\neg\varphi}$ by using $\Sigma^s_{\neg\varphi}$ to find "candidate" L-events. Assume we have found an event e^s corresponding to a transition t' in the unfolding of $\Sigma^s_{\neg\varphi}$ and the stuttering check described above passes for the marking $M = \mathit{Mark}([e^s])$. Then we add an event e into the unfolding of $\Sigma_{\neg\varphi}$ corresponding to the effect of the transition t in the marking M. If we would directly use the construction above we would also add an event e' to the unfolding of $\Sigma_{\neg\varphi}$ for each marking $M' = (q, s_f, O, H')$ which is reachable from M using only invisible transitions. We now show that adding only the event e suffices: Let E be an extension of $[e]$. If there is an illegal livelock starting from $M' = \mathit{Mark}([e] \oplus E)$ then there is also an illegal livelock starting from M. This can be easily seen to be the case because all extensions E contain only invisible events and thus the set of observable places in both M and M' is O. Algorithm 1 uses the property described above to add the required L-events dynamically. Another optimization used is the fact that only the places in the presets of invisible transitions (denoted *InvisPre*) need to be added to the postset of an L-transition.

Algorithm 2 is the cut-off detection subroutine. It handles events in *part-I* and *part-II* differently. This is one example implementation, and it closely follows the definition of the tableau. It sets the global boolean variable *success* to *true* and calls the counterexample generation subroutine (Algorithm 3) if it finds a counterexample.

The implementation of the check whether $\mathcal{A}^q_{\neg\varphi}$ accepts O^ω in Algorithm 1 can be done in linear time in the size of the automaton $\mathcal{A}_{\neg\varphi}$ as follows. First restrict $\mathcal{A}_{\neg\varphi}$ to transitions satisfying O, and then use a linear time emptiness checking algorithm (see e.g. [2]) to check whether an accepting loop can be reached starting from q in this restricted automaton. Because $\mathcal{A}_{\neg\varphi}$ is usually quite small compared to the size of the model checked system this should not be a limiting factor. Caching of these check results can also be used if necessary.

The adequate order $\prec_{LTL}$ can also be quite efficiently implemented. We can prove that if a configuration C contains an L-event e, then $BL(C) = [e]$. Now it is also the case that each configuration only includes at most one L-event. By using these two facts a simple and efficient implementation can be devised.

Each time our algorithm adds a non-terminal L-event, it first finds out whether a livelock counterexample can be generated from its future. Only if no counterexample is found, it continues to look for illegal ω-traces and further L-events. Thus we use the adequate order $\prec_{LTL}$ to force a search order similar to that used by Valmari in [19] which detects divergence counterexamples in interleaved state spaces. However, our algorithm is "breadth-first style" and it also does illegal ω-trace detection, a part which is not included in [19].

Algorithm 1 *The tableau generation algorithm*

input: The product net system $\Sigma^s_{\neg\varphi} = (P, T, F, M_0)$, where $M_0 = \{p_1, \ldots, p_n\}$.
output: *true* if there is a counterexample, *false* otherwise.
global variables: *success*
begin
$Fin := \{(p_1, \bot), \ldots, (p_n, \bot)\}$;
cut-off $:= \emptyset$;
$pe := PE(Fin)$; /* Compute the set of possible extensions */
success := *false*;
while $pe \neq \emptyset$ and *success* = *false* **do**
    choose an event $e = (t, X)$ in pe such that $[e]$ is minimal
    with respect to $\prec_{LTL}$;
    $Y := t^\bullet$; /* Remember the postset of t */

    /* Create the required L-events dynamically */
    if t is a L-transition **then**
        $M := Mark([e] \setminus \{e\})$; /* The marking $M = (q, s_f, O, H)$ */
        $q := M \cap Q$; /* Extract the Büchi state q */
        /* (Büchi emptiness checking algorithm can be used here) */
        if $\mathcal{A}^q_{\neg\varphi} = (\Gamma, Q, q, \rho, F)$ does not accept O^ω **then**
            continue; /* Discard e because (q, O) is not a checkpoint */
        endif
        $X := Cut([e] \setminus \{e\})$; /* Extend the preset to also remove tokens from H */
        $e := (t, X)$; /* Rename e (i.e., add arcs from all preset conditions to e) */
        $Y := (M \cap InvisPre)$; /* Project M on invisible transition presets */
    endif

    if $[e] \cap$ *cut-off* $= \emptyset$ **then**
        append to Fin the event e and a condition (p, e)
            for every place $p \in Y$;
        $pe := PE(Fin)$; /* Compute the set of possible extensions */
        if *is_cutoff*(e) **then**
            cut-off := *cut-off* $\cup \{e\}$;
        endif
    else
        $pe := pe \setminus \{e\}$;
    endif
enddo
return *success*;
end

Algorithm 2 *The is_cutoff subroutine*

input: An event e.
output: *true* if e is a terminal of the tableau, *false* otherwise.
begin
foreach e' such that $Mark([e']) = Mark([e])$ **do** /* $[e'] \prec_{LTL} [e]$ holds */
  if $e \in$ *part-I* **then** /* case (I) */
    if $e' < e$ **then**
      if $[e] \setminus [e']$ contains an I-event **then**
        $success := true$; /* Counterexample found! */
        $counterexample(e, e')$;
      endif
      return *true*;
    else if $\#_I[e'] \geq \#_I[e]$ **then**
      return *true*;
    endif
  else /* case (II) */
    if $BL([e']) \prec_{LTL} BL([e])$ **then**
      return *true*;
    else if $\neg(e' \# e)$ **then** /* $BL([e']) = BL([e])$ holds */
      $success := true$; /* Counterexample found! */
      $counterexample(e, e')$;
      return *true*;
    else if $|[e']| \geq |[e]|$ **then** /* $BL([e']) = BL([e])$ holds */
      return *true*;
    endif
  endif
enddo
return *false*;
end

Algorithm 3 *The counterexample subroutine*

input: A successful event e with the corresponding event e'.
begin
$C_1 := [e] \cap [e']$;
$C_2 := [e] \setminus C_1$;
/* C_1 contains the prefix and C_2 the accepting loop */
print_linearisation(C_1);
print_linearisation(C_2);
end

7 Experimental Results

We have implemented a prototype of the LTL model checking procedure called `unfsmodels`. We use the SPIN tool [12] version 3.4.3 to generate the Büchi automaton $\mathcal{A}_{\neg\varphi}$ and a tool by F. Wallner [22] to generate the synchronization $\Sigma'_{\neg\varphi}$ which is given to the prototype tool as input.

The `smodels` tool [18] is used to calculate the set of possible extensions of a branching process. It is a NP-solver which uses logic programs with stable model semantics as the input language. Calculating the possible extensions is a quite demanding combinatorial problem. Actually a decision version of the problem can be show to be NP-complete in the general case [10]. However if the maximum preset size of the transitions $|{}^\bullet t|$ is bounded the problem becomes polynomial [7]. (The problem is closely related to the *clique* problem which has a similar characteristic, for a longer discussion see [7].)

We chose to use `smodels` to solve this combinatorial problem instead of implementing a dedicated algorithm. That choice allowed us to concentrate on other parts of the implementation. The translation employs constructs similar to those presented for the submarking reachability problem in [11], however it differs in several technical details. The translation is linear in the sizes of both the net and the prefix, however we will not present it here due to space restrictions.

For benchmarks we used a set of LTL model checking examples collected by C. Schröter. The experimental results are collected in Fig. 3. The 1-safe net systems used in the experiments are as follows:

- BRUIJN(2), DIJKST(2), and KNUTH(2): Mutex algorithms modeled by S. Melzer.
- BYZA4_0B and BYZA4_0B: Byzantine agreement algorithm versions modeled by S. Merkel [16].
- RW1W1R, RW1W3R and RW2W1R: Readers and writers synchronization modeled by S. Melzer and S. Römer [15].
- PLATE(5): A production cell example from [13], modeled by M. Heiner and P. Deussen [9].
- EBAHN: A train model by K. Schmidt.
- ELEV(3) and ELEV(4): Elevator models by J. C. Corbett [1], converted to nets by S. Melzer and S. Römer [15].
- RRR(xx): Dining philosophers with xx philosophers, modeled by C. Schröter.

The reported running times only include `unfsmodels 0.9` running times, as the Büchi automata generation and the synchronization with the original net system took insignificant amount of time. All the running times are reported as the sum of system and user times as reported by the `/usr/bin/time` command when run on a PC with an AMD Athlon 1GHz processor, 512MB RAM, using `gcc 2.95.2` and Linux 2.2.17. The times are all averaged over 5 runs.

The `unfsmodels` tool in an on-the-fly tool in the sense that it stops the prefix (tableau) generation if it finds a counterexample during the unfolding. The

Problem	B_{LTL}	E_{LTL}	$\#c_{LTL}$	Cex	B_{Fin}	E_{Fin}	$\#c_{Fin}$	States	Sec_{LTL}	Sec_{Fin}
BRUIJN(2)	2874	1336	327	N	2676	1269	318	5183	13.1	11.0
DIJKST(2)	1856	968	230	N	1700	921	228	2724	4.8	3.8
KNUTH(2)	2234	1044	251	N	2117	1009	251	4483	7.1	6.1
BYZA4_0B	1642	590	82	N	1630	587	82	>2000000	7.0	6.9
BYZA4_2A	401	125	4	N	396	124	4	>2500000	0.3	0.3
RW1W1R	568	296	32	N	563	295	32	2118	0.5	0.5
RW1W3R	28143	15402	5210	N	28138	15401	5210	165272	1863.4	1862.2
RW2W1R	18280	9242	1334	N	18275	9241	1334	127132	1109.6	1108.2
PLATE(5)	1803	810	12	N	1619	768	12	1657242	14.0	11.8
EBAHN	151	62	21	Y	1419	673	383	7776	0.0	0.7
ELEV(3)	124	64	10	Y	7398	3895	1629	7276	0.1	91.7
ELEV(4)	154	80	13	Y	32354	16935	7337	48217	0.1	1706.2
RRR(10)	88	42	5	Y	85	45	19	14985	0.0	0.0
RRR(20)	167	81	8	Y	161	81	32	>10000000	0.1	0.0
RRR(30)	240	114	9	Y	230	110	41	>10000000	0.2	0.1
RRR(50)	407	201	18	Y	388	188	70	>10000000	0.7	0.5

Figure 3. Experimental results.

reported prefix sizes in this case are the partial prefix at the time the counterexample was found. The tool can also be instructed to generate a *conventional prefix* using the prefix generation algorithm described in [6] for comparison.

In Fig. 3 the columns of the table have the following meanings:

- Problem: The name of the problem with the size of the instance.
- B_{LTL}, E_{LTL}, and $\#c_{LTL}$: The number of conditions, events, and the number of events which are terminals in the LTL prefix, respectively.
- Cex: N - There was no counterexample, the formula holds. Y - There was a counterexample, the formula does not hold.
- B_{Fin}, E_{Fin}, and $\#c_{Fin}$: The size of different parts of the finite complete prefix as above but for the original net system Σ using the conventional prefix generation algorithm described in [6].
- States: The number of states n in the reachability graph of the original net system Σ obtained using the `PROD` tool [21], or a lower bound $> n$.
- Sec_{LTL}: The time used by `unfsmodels` in seconds needed to find a counterexample or to show that there is none.
- Sec_{Fin}: The time used by `unfsmodels` in seconds needed to generate a finite complete prefix of the original net system Σ.

At this point there are a couple of observations to be made. First of all, on this set of example nets and formulas, the speed of computing a LTL prefix is almost identical to the speed of computing a conventional prefix (of comparable size). The main reason for this is that the time needed to compute the possible extensions dominates the computation time in our prototype. Thus the (slightly) more complicated algorithm needed for the cut-off detection do not contribute in

a major way to the running time of the tool. Secondly, on all of the experiments, the size of the LTL prefix is of the same order of magnitude as the conventional prefix. Thus in this set of examples the quadratic worst-case blow-up (possible according to Theorem 3) does not materialize. We expect this to be the case also in other examples when the used LTL formulas are short and the properties to be checked are local, in the sense that the product net system preserves most of the concurrency present in the original net system.

Problem	B_I	E_I	$\#c_I$	B_{II}	E_{II}	$\#c_{II}$	Cpt	Formula type
BRUIJN(2)	2874	1336	327	0	0	0	0	$\Box\neg(p_1 \wedge p_2)$
DIJKST(2)	1856	968	230	0	0	0	0	$\Box\neg(p_1 \wedge p_2)$
KNUTH(2)	2234	1044	251	0	0	0	0	$\Box\neg(p_1 \wedge p_2)$
BYZA4_0B	1642	590	82	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
BYZA4_2A	401	125	4	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
RW1W1R	568	296	32	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
RW1W3R	28143	15402	5210	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
RW2W1R	18280	9242	1334	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
PLATE(5)	1803	810	12	0	0	0	0	$\Box((p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge p_3))$
EBAHN	113	48	20	38	14	1	1	$\Box\neg(p_1 \wedge p_2)$
ELEV(3)	22	10	0	102	54	10	1	$\Box(p_1 \rightarrow \Diamond p_2)$
ELEV(4)	25	12	0	129	68	13	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(10)	40	14	0	48	28	5	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(20)	73	27	0	94	54	8	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(30)	104	38	0	136	76	9	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(50)	173	67	0	234	134	18	1	$\Box(p_1 \rightarrow \Diamond p_2)$

Figure 4. Detailed LTL tableau statistics.

In Fig. 4 a detailed breakdown of the different components of the LTL prefix is given. The subscripts I and II denote the part of the prefix used for ω-trace and livelock checking, respectively (i.e., events in *part-I* and *part-II*). Column *Cpt* contains the number of checkpoints, i.e. how many of the L-events are checkpoints. Finally *Formula type* gives the type of the formula being checked.

In Fig. 4 we can also see that in the cases a counterexample was found it was found after only a small amount of the prefix was generated. Actually in all the experiments the counterexample was a livelock counterexample, and the livelock was found from the first checkpoint found during the prefix generation. This allowed the LTL model checking procedure to terminate quite early with a counterexample in many case, see e.g. the ELEV(4) example.

The net systems used in experiments and `unfsmodels 0.9` are available at <`http://www.tcs.hut.fi/~kepa/experiments/spin2001/`>.

8 Conclusions

We have presented an implementation of the tableau system of [3]. We have been able to merge the possibly large set of tableaux of [3] into a single one. In this way, the algorithm for model checking LTL with unfoldings remains conceptually similar to the algorithms used to generate prefixes of the unfolding containing all reachable states [6,5]: We just need more sophisticated adequate orders and cut-off events.

The division of the tableau into *part-I* and *part-II* events is the price to pay for a partial-order approach to model checking. Other partial-order techniques, like the one introduced by Valmari [19], also require a special treatment of divergences or livelocks. [3] We have shown that the conditions for checking if *part-I* or *part-II* events are terminals remain very simple.

In our tableau system the size of a tableau may grow quadratically in the number of reachable states of the system. We have not been able to construct an example showing that this bound can be reached, although it probably exists. In all experiments conducted so far the number of events of the tableau is always smaller than the number of reachable states. In examples with a high degree of concurrency we obtain exponential compression factors.

The prototype implementation was created mainly for investigating the sizes of the generated tableau. Implementing this procedure in a high performance prefix generator such as the one described in [5] is left for further work.

Acknowledgements

We would like to thank Claus Schröter for collecting the set of LTL model checking benchmarks used in this work.

References

1. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. Technical report, Department of Information and Computer Science, University of Hawaii at Manoa, 1995.
2. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
3. J. Esparza and K. Heljanko. A new unfolding approach to LTL model checking. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, pages 475–486, July 2000. LNCS 1853.
4. J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. Research Report A68, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, March 2001. Available at <`http://www.tcs.hut.fi/Publications/reports/A68abstract.html`>.

[3] The idea of dynamically checking which L-transitions are checkpoints could also be used with the approach of [19] to implement state based LTL-X model checking.

5. J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *Proceedings of the 10th International Conference on Concurrency Theory (Concur'99)*, pages 2–20, 1999. LNCS 1664.
6. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proceedings of 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 87–106, 1996. LNCS 1055.
7. J. Esparza and C. Schröter. Reachability analysis using net unfoldings. In *Proceeding of the Workshop Concurrency, Specification & Programming 2000, volume II of Informatik-Bericht 140*, pages 255–270. Humboldt-Universität zu Berlin, 2000.
8. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of 15th Workshop Protocol Specification, Testing, and Verification*, pages 3–18, 1995.
9. M. Heiner and P. Deussen. Petri net based qualitative analysis - A case study. Technical Report Technical Report I-08/1995, Brandenburg Technische Universität Cottbus, Cottbus, Germany, December 1995.
10. K. Heljanko. Deadlock and reachability checking with finite complete prefixes. Research Report A56, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 1999. Licentiate's Thesis. Available at <`http://www.tcs.hut.fi/Publications/reports/A56abstract.html`>.
11. K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.
12. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
13. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell*. Springer-Verlag, 1995. LNCS 891.
14. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
15. S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV '97)*, pages 352–363, 1997. LNCS 1254.
16. S. Merkel. Verification of fault tolerant algorithms using PEP. Technical Report TUM-19734, SFB-Bericht Nr. 342/23/97 A, Technische Universität München, München, Germany, 1997.
17. W. Reisig. *Petri Nets, An Introduction*. Springer-Verlag, 1985.
18. P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Laboratory for Theoretical Computer Science, April 2000. Also available on the Internet at <`http://www.tcs.hut.fi/Publications/reports/A58abstract.html`>.
19. A. Valmari. On-the-fly verification with stubborn sets. In *Proceeding of 5th International Conference on Computer Aided Verification (CAV'93)*, pages 397–408, 1993. LNCS 697.
20. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, pages 238–265, 1996. LNCS 1043.
21. K. Varpaaniemi, K. Heljanko, and J. Lilius. PROD 3.2 - An advanced tool for efficient reachability analysis. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 472–475, 1997. LNCS 1254.
22. F. Wallner. Model checking LTL using net unfoldings. In *Proceeding of 10th International Conference on Computer Aided Verification (CAV'98)*, pages 207–218, 1998. LNCS 1427.