Model Checking LTL Using Net Unforldings*

Frank Wallner

Institut für Informatik, Technische Universität München
Arcisstr.21, D-80290 München, Germany
email: wallnerf@in.tum.de

Abstract. Net unfoldings are a well-studied partial order semantics for Petri nets. In this paper, we show that the finite prefix of an unfolding, introduced by McMillan, is suited for model checking linear-time temporal properties. The method is based on the so-called automata-theoretic approach to model checking. We propose a technique to treat this approach within the framework of safe Petri nets, and give an efficient algorithm for detecting the system runs violating a given specification.

1 Introduction

Linear-time Temporal Logic (LTL) is an adequate formalism for specifying behavioural properties of distributed systems, including safety and liveness properties. Deciding whether a given system Σ satisfies a specification φ is called the *model checking problem.*

The *automata-theoretic approach* to model checking translates this problem into an automata-theoretic problem. This approach assumes that Σ can be represented as an automaton A with $L(A)$ being the set of its *runs*. The system satisfies φ iff $L(A)$ is a subset of the language L_φ of words satisfying φ.

Vardi, Wolper et al. [20,22] observed that for every formula φ it is possible to construct a Büchi automaton A_φ that accepts L_φ. Since negation of a formula φ is equivalent to complementing the corresponding language L_φ, the actual problem is to decide if there is a system run accepted by $A_{\neg\varphi}$. Defining an adequate product automaton A_p of A and $A_{\neg\varphi}$ that accepts the intersection of $L(A)$ and $L_{\neg\varphi}$, the problem is finally transformed to an emptiness-problem on automata: the system satisfies φ iff A_p is empty (accepts no word).

Checking emptiness of A_p requires the detection of *accepting cycles,* i.e., cycles containing an accepting state. There exist efficient algorithms for this issue [1,9] with time complexity linear in the size of the product automaton. However, this size is often enormous, due to the well-known state explosion problem: representing concurrency as interleaving may let A, and consequently A_p, grow exponentially in the size of the system. Several partial order methods [10,16–19] have been suggested to palliate this problem by reducing the state space according to the partial order semantics of the system, i.e., by discarding all the states and transitions not relevant for satisfying φ.

* This work was supported by the SFB 342 (subproject A3) of the DFG.

In contrast to these approaches, we will not reduce the state space, but rather directly use a partial-order representation of the behaviour of the distributed system under consideration. We assume Σ to be given as a safe Petri net, and explore its behaviour by unfolding the net to McMillan's *finite prefix* [14,7] of the branching process of Σ. This prefix contains every reachable state of the system. It was already observed by Esparza in [5] that the finite prefix can be used for model checking S4 (the modal logic based on the reachability relation of the global state space), which is strictly less expressive than LTL.

We show in this paper how to construct the product of a given Petri net and a Büchi automaton, yielding *Büchi nets*, i.e. nets with acceptance capabilities. We investigate, for which construction the prefix remains small, in some cases "exponentially compact" compared with the interleaving model. The main contribution, however, is a method for checking emptiness of a Büchi net using its finite prefix.

The paper is structured as follows. Section 2 briefly formalizes the automata-theoretic approach. In Section 3, Petri nets and unfoldings are introduced, and we show how the finite prefix of a Büchi net can be used to decide its emptiness. Section 4 describes an adequate product construction and the entire model checking procedure. Section 5 concludes the paper and refers to related work.

2 The automata-theoretic approach

Let us briefly recall the essential ideas and notions that underlie the automata-theoretic approach to model checking linear-time temporal properties.

Linear-time Temporal Logic. Let Π be a finite set of atomic propositions. The set of *LTL-formulae over Π* is defined inductively as follows: if $\varphi = \pi \in \Pi$ then φ is a formula; if φ and ψ are formulae then $\varphi \wedge \psi$, $\neg\varphi$, $\mathsf{X}\varphi$, and $\varphi \mathsf{U} \psi$ are formulae. The other operators of propositional logic are defined as usual, and we define $\Diamond\varphi := \mathsf{true}\mathsf{U}\varphi$, and $\Box\varphi := \neg\Diamond\neg\varphi$. The set of propositions appearing in φ is written as $\langle\varphi\rangle$.

A formula is interpreted on ω-words ξ over the alphabet 2^Π. An *ω-word* over 2^Π is an infinite sequence $\xi = x_0 x_1 \ldots$ with $x_i \in 2^\Pi$ for all $i \geq 0$. The elements of 2^Π are meant to assign truth values to Π in the obvious manner: the proposition π holds at x_i iff $\pi \in x_i$. We define $\xi(i) := x_i$, and $\xi^{(i)}$ is the suffix of ξ starting at x_i. We write $\xi \models \varphi$ to denote that ξ *satisfies* φ. By L_φ we denote the set of ω-words satisfying φ. The relation $\models$ is inductively defined as follows.

$$\begin{array}{lll}
\xi \models \pi & \text{iff} & \pi \in \xi(0) \\
\xi \models \neg\varphi & \text{iff} & \xi \not\models \varphi \\
\xi \models \varphi \wedge \psi & \text{iff} & \xi \models \varphi \text{ and } \xi \models \psi \\
\xi \models \mathsf{X}\varphi & \text{iff} & \xi^{(1)} \models \varphi \\
\xi \models \varphi \mathsf{U} \psi & \text{iff} & \exists\, i \geq 0.\ \xi^{(i)} \models \psi \text{ and } \xi^{(j)} \models \varphi \text{ for all } j < i
\end{array}$$

Büchi automata. A Büchi automaton over the alphabet 2^Π is a quadruple $A = (Q, q_0, \delta, \mathcal{F})$, where Q is a finite set of states, including the initial state q_0,

$\delta \subseteq Q \times 2^{\Pi} \times Q$ is the transition relation, and $\mathcal{F} \subseteq Q$ a set of *accepting states.* A *run* of A on an ω-word ξ over 2^{Π} is an infinite sequence $\sigma = q_0 q_1 \ldots$ such that $(q_i, \xi(i), q_{i+1}) \in \delta$ for all $i \geq 0$. A run σ is *accepting* if an accepting state occurs infinitely often in σ, and the automaton A *accepts* the word ξ iff there is an accepting run of A on ξ. $L(A)$ denotes the set of all ω-words accepted by A.

Theorem 1 ([20,22]). *Let φ be an LTL formula. There exists a Büchi automaton A_φ such that $L(A_\varphi) = L_\varphi$.*

Efficient methods for how to build the automaton A_φ from a given formula φ can be found in [21,8].

The automata-theoretic approach assumes the system to be given as an automaton A over an alphabet *Act* of actions, and a valuation v from the transitions of A to subsets of Π. In *action-based* semantics, v is determined by the action associated with the transition, in a *state-based* setting by the state that enables the transition. Given the system A and the automaton $A_{\neg\varphi}$ for the negation of φ, an adequate *product automaton* A_p is defined. The basic idea is that $(\langle s, q\rangle, \langle s', q'\rangle)$ becomes a transition of A_p if there is a transition (s, a, s') of the system evaluated to Π', and (q, Π', q') is a transition of $A_{\neg\varphi}$. The accepting states of A_p are the states $\langle s, q\rangle$ where q ia an accepting state of $A_{\neg\varphi}$. For this construction it holds that the product A_p is empty iff it contains no cycle including an accepting state iff the system automaton A satisfies the property φ.

3 Petri nets and unfoldings

Let us begin with a glance on Petri nets and their unfoldings. We will then show how to use McMillan's prefix for deciding the existence of accepting runs.

Petri nets. Let P and T be disjoint sets of *places* and *transitions.* The elements of $P \cup T$ are called *nodes.* A *net* is a triple $N = (P, T, F)$ with a *flow relation* F, given by its characteristic function $F : (P \times T) \cup (T \times P) \rightarrow \{0, 1\}$.

The *preset* of the node x is defined as ${}^\bullet x := \{y \in P \cup T \mid F(y, x) = 1\}$ and its *postset* as $x^\bullet := \{y \in P \cup T \mid F(x, y) = 1\}$. The preset (postset) of a set X of nodes is given by the union of the presets (postsets) of all nodes in X. By ${}^\circ x$ we denote the set ${}^\bullet x \setminus x^\bullet$, and analogously $x^\circ := x^\bullet \setminus {}^\bullet x$.

A *marking* of a net is a mapping $P \rightarrow \mathbb{N}_0$. We call $\Sigma = (N, M_0)$ a *net system* with initial marking M_0 if N is a net and M_0 a marking of N. A marking M *enables* the transition t if $M(p) \geq 1$ for each $p \in {}^\bullet t$. In this case the transition can *occur*, leading to the new marking M', given by $M'(p) = M(p) + F(t, p) - F(p, t)$ for every place p. We denote this occurrence by $M \xrightarrow{t} M'$. If there exists a chain $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$, the sequence $\gamma = M_0 t_1 M_1 t_2 \ldots t_n M_n$ is called a *computation.* A computation of infinite length is called a *run.* A marking M is *reachable* if there exists a computation γ such that M appears in γ. The reachable markings will also be called the *(reachable) states* of the system.

We will exclusively regard *safe systems,* in which all reachable states map each place to 0 or 1. So every state can be identified with the set of places it maps to 1, i.e., $M \subseteq P$ for every reachable state M. Note that this is no restriction. Often safe nets are used for modelling distributed systems because they can be seen as a composition of several components which are given as finite automata. Furthermore, so-called high-level net systems like coloured or algebraic nets can automatically be transformed into equivalent safe net systems.

Net system semantics for LTL. We define an adequate LTL semantics for safe net systems, distinguishing state and action oriented settings.

In a *state-based* interpretation, the atomic propositions Π are identified with the set P of places. A proposition p holds at state M iff $M(p) = 1$. Since every state can be expressed as a Boolean combination of marked and unmarked places, any set of propositions on states can be encoded using places as the only propositions. Formulae are interpreted on marking sequences: a run $\gamma = M_0 t_1 M_1 t_2 M_2 \dots$ satisfies φ iff the ω-word $\xi(\gamma) = M_0 M_1 M_2 \dots$ belongs to L_φ.

In an *action-based* interpretation, we assume a valuation $v : T \rightarrow 2^\Pi$, and we interpret formulae on sequences of transition occurrences: a run $\gamma = M_0 t_1 M_1 t_2 M_2 \dots$ satisfies φ iff the ω-word $\sigma(\gamma) = v(t_1)v(t_2)\dots$ belongs to L_φ.

We say that the system Σ satisfies φ iff every run of Σ satisfies φ.

Büchi nets. A *Büchi net* is a net with acceptance capabilities, i.e., a tuple $\Sigma_p = (\Sigma, \mathcal{F})$ where $\Sigma = (P, T, F, M_0)$ is a finite, safe net system and $\mathcal{F} \subseteq P$ a set of accepting places. A run γ of Σ_p is *accepting* if an *accepting transition* $t \in {}^\bullet\mathcal{F}$ appears infinitely often in γ, and Σ_p *is empty* if it has no accepting run. A Büchi net will be the product of a safe net system and a Büchi automaton, defined in the next section.

Net unfoldings. The partial-order representation of the behaviour of safe net systems is based on net unfoldings, also known as branching processes. We briefly recall the main definitions and results of [4].

Two nodes x, x' of the net $N = (P, T, F)$ are *in conflict*, denoted $x \# x'$, if there exist two distinct transitions t, t' with ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ such that (t, x) and (t', x') belong to the reflexive transitive closure of the flow relation F. If $x \# x$, we say x *is in self-conflict.*

An *occurrence net* [15] is a net $N' = (B, E, F)$ where the irreflexive transitive closure of F is well-founded and acyclic (and thus a strict partial order, written as $\prec$), where furtheron $|{}^\bullet b| \leq 1$ for every $b \in B$, and no element $e \in E$ is in self-conflict. The elements of B and E are called *conditions* and *events*, respectively. The reflexive closure $\preceq$ of $\prec$ is a partial order called *causality relation*. By $Min(N')$ we denote the minimal elements of N' w.r.t. $\preceq$.

Given two nets N_1 and N_2, the mapping $h : N_1 \rightarrow N_2$ is a *homomorphism* if $h(P_1) \subseteq P_2$, $h(T_1) \subseteq T_2$, and if for each $t \in T_1$ the restriction of h to ${}^\bullet t$ is a bijection between ${}^\bullet t$ and ${}^\bullet h(t)$, and similarly for $t^\bullet$ and $h(t)^\bullet$.

A *branching process* of a net system $\Sigma = (N, M_0)$ is a pair $\beta = (N', h)$ where $N' = (B, E, F)$ is an occurrence net and $h : N' \rightarrow N$ is a homomorphism that bijectively maps $Min(N')$ onto M_0, and that satisfies: if $h(e) = h(e')$ and ${}^\bullet e = {}^\bullet e'$ then $e = e'$, for all events $e, e' \in E$. In a word, we unfold the net N to an occurrence net N' such that each node x of N' refers to a node $h(x)$ of N.

The branching processes β_1 and β_2 are *isomorphic* if there exists a bijective homomorphism $h : N_1' \to N_2'$, such that the composition $h_2 \circ h$ equals h_1. If h is an injection that bijectively maps $Min(N_1')$ onto $Min(N_2')$, and $B_1 \subseteq B_2$ and $E_1 \subseteq E_2$, we call β_1 a *prefix* of β_2. Notice that a prefix is uniquely determined by its set of events. In [4] it is shown that each net system Σ has a unique maximal branching process up to isomorphism, called *unfolding of* Σ and denoted by $Unf_\Sigma = (N', h)$. Note that N' is infinite iff Σ has infinite computations.

Configurations and Cuts. For the remainder of the section, let $Unf_\Sigma = (N', h)$ and $N' = (B, E, F)$ be fixed. A *configuration* C of N' is a causally downward-closed, conflict-free set of events, i.e., for each $e \in C$: if $e' \preceq e$ then $e' \in C$, and for all $e, e' \in C : \neg(e \# e')$.

Two nodes of N' are *concurrent* if they are neither in conflict nor causally related. A set B' of conditions of N' is called a *co-set* if all elements of B' are pairwise concurrent. A co-set is called a *cut* if it is maximal w.r.t. set inclusion. For a finite configuration C, the set $Cut(C) := (Min(N') \cup C^\bullet) \setminus {}^\bullet C$ of conditions is a cut. The set $h(Cut(C))$ of places is a reachable marking of Σ, called the *marking* $Mark(C)$ *of* C. Conversely, for every reachable state M of Σ there exists a finite configuration C in Unf_Σ such that M is the marking of C. Often, a configuration C is identified with the state $Mark(C)$.

An essential observation on configurations is that their *continuations* are determined by their markings: let $\uparrow C \subseteq B \cup E$ be defined as the set of nodes x, such that $x \in \uparrow C$ iff $x \succeq b$ for some $b \in Cut(C)$ and $\neg(b \# x)$ for all $b \in Cut(C)$. By F_C (resp. h_C) we denote the restriction of the flow relation F (resp. of the homomorphism h) of Unf_Σ onto $\uparrow C$. We define the *continuation of* C as the branching process $\beta(C) := (N_C, h_C)$, where $N_C := (\uparrow C \cap B, \uparrow C \cap E, F_C)$. It is easy to see that for two finite configurations C, C' with equal marking it holds that $\beta(C)$ and $\beta(C')$ are isomorphic.

The set of predecessors of each event e is a configuration, called *local configuration of* e, given by $[e] := \{e' \in E \mid e' \preceq e\}$. We call two events e, e' *equivalent* if the markings of their local configurations coincide, i.e., $Mark([e]) = Mark([e'])$.

The finite prefix. In [14], K.L. McMillan defined a finite prefix of the unfolding of a finite-state net system, in which every state is represented by some cut. The idea is that if the prefix contains two equivalent events then the continuations of their local configurations are isomorphic and thus only one of them needs to be explored further, while the other one becomes a *cut-off event*. Formally, an event e is a cut-off event if there exists an event e' equivalent to e such that $|[e']| < |[e]|$. If there are several such events e' for the cut-off e, we fix one of them and refer to it as the *corresponding event* $cor(e)$ of e. By $off(e')$ we denote the set of cut-offs, such that e' is their corresponding event.

The *finite prefix* Fin_Σ is defined as the unique prefix of Unf_Σ with $E_{Fin} \subseteq E$ as set of events, where $e \in E_{Fin}$ iff no event $e' \prec e$ is a cut-off event. Let *Off* (*Cor*) denote the set of all cut-off (corresponding) events of Fin_Σ.

It is easy to prove that Fin_Σ is finite for net systems with finitely many states. Usually, Fin_Σ is much smaller than the state space of the system. However, sometimes it is larger. In [7] it is shown how to construct a *storage-optimal* prefix, essentially by determining cut-offs not by comparison of the size of their local configurations, but another well-founded, *strict* partial order instead. In the prefix constructed by the improved algorithm [7], it is always the case that two non-cut-off events have different markings. Therefore, the number of non-cut-off events never exceeds the number of reachable states of the system, and so Fin_Σ never is larger than the state space (up to a small constant).

Cycle-detection in the prefix. As indicated, the model checking problem requires the detection of a cycle containing an accepting transition. Let $T_a = {}^\bullet\mathcal{F}$ be the set of accepting transitions of the Büchi net $\Sigma_p = (\Sigma, \mathcal{F})$. The goal is to find a run γ such that infinitely often a transition $t \in T_a$ appears in γ.

The problem is solved in two steps: first we will construct a directed graph $G = (V, Edg)$ where $V = Off$ is the set of cut-off events of the prefix, and $Edg \subseteq V \times V$ a set of edges. An edge $e \dashrightarrow e'$ indicates that from state $[e]$ the state $[e']$ is reachable. Some of the edges will be labelled by a. Intuitively, $e \xrightarrow{a} e'$ means that on the partial computation leading from $[e]$ to $[e']$ an accepting transition occurs. Since every (local) configuration of the prefix is reachable, every node in G can be seen as being initial. The graph G is constructed by the algorithm given in Fig. 1, with T_a as the input parameter.

The second step is to apply a standard algorithm on G for detecting a strongly connected component [1] or a cycle [9] containing an a-labelled edge.

The key idea of the algorithm for constructing the graph G is as follows. Let e_1, e_1^0 be a cut-off and its corresponding event. Since $\beta([e_1])$ and $\beta([e_1^0])$ are isomorphic, every state that is reachable from $[e_1^0]$ is also reachable from $[e_1]$. Thus, if $e_1^0 \prec e_2$ for some other cut-off e_2, an edge $e_1 \dashrightarrow e_2$ is added to G. If $[e_2] \setminus [e_1^0]$ contains an accepting event then the edge is labelled by a.

The other case is a bit more involved. Let e_2^0 be the corresponding event of e_2, and assume $e_1^0 \preceq e_2^0$. This means that from state $[e_1^0]$ (equivalent to $[e_1]$) the state $[e_2^0]$ (equivalent to $[e_2]$) is reachable, and so an edge $e_1 \dashrightarrow e_2$ is added. But when and how has such an edge to be labelled? Clearly, if $[e_2^0] \setminus [e_1^0]$ contains an accepting event, the edge must be labelled. However, this is not the only case. Additionally, there may exist a state of the system (possibly not corresponding to a local configuration) where *concurrently* $[e_1^0]$ and $[e_2]$ are reachable. In this case, we have to consider the set $E_1^2 := [e_2] \setminus [e_1^0]$. If e_1^0 and e_2 are concurrent, and the set E_1^2 contains an accepting event, we label the edge $e_1 \dashrightarrow e_2$ with a. In the algorithm, let $\Gamma(e_1^0, e_2)$ denote the function computing the set of these events: $\Gamma(e_1^0, e_2) := E_1^2$, if e_1^0 and e_2 are concurrent, and the empty set else.

Proposition 2. *$T_a \subseteq T$ contains a transition that infinitely often can occur in Σ iff there exists a cycle in the graph G, containing an a-labelled edge.*

4 The automata-theoretic approach for Petri nets

We now want to lift the automata-theoretic approach to the framework of safe net systems. We show two different methods for constructing a product Büchi

BuildGraph(T_a)

```
V := Off;  Edg := ∅;  $E_a := \{e \in E_{Fin} \mid h(e) \in T_a\}$;
forall $e_1^0 \in Cor$ do
  $X := \{e' \in Off \mid e' \succeq e_1^0\}$; $Y := \{e' \in Cor \mid e' \succeq e_1^0\}$;
  forall $e_2 \in X$ do
    forall $e_1 \in off(e_1^0)$ do
      if $G$ contains no edge $e_1 \rightarrow e_2$ then add $e_1 \rightarrow e_2$ to $Edg$;
      if $([e_2] \setminus [e_1^0]) \cap E_a \neq \emptyset$ then label $e_1 \rightarrow e_2$ with $a$;
    enddo
  enddo
  forall $e_2^0 \in Y$ do
    forall $(e_1, e_2) \in off(e_1^0) \times off(e_2^0)$ do
      if $G$ contains no edge $e_1 \rightarrow e_2$ then add $e_1 \rightarrow e_2$ to $Edg$;
      if ( $([e_2^0] \setminus [e_1^0]) \cup \Gamma(e_1^0, e_2)$ ) $\cap E_a \neq \emptyset$ then label $e_1 \rightarrow e_2$ with $a$;
    enddo
  enddo
enddo
```

Fig. 1. Algorithm for constructing the graph G.

net, corresponding to product automata of Section 2. This product is constructed as a *synchronization* or an *observation* on the net level.

In the entire section, we assume the system net under consideration to be *deadlock-free*, i.e., all of its computations are infinite.

Synchronization. We will first assume an action-based interpretation. In this case, the product net Σ_p of the automaton $A_{\neg\varphi}$ and of the system Σ under consideration is obtained by synchronizing the transitions according to the valuation v. Let $A_{\neg\varphi} = (Q, q_0, \delta, \mathcal{F})$ be fixed.

The product net Σ_p is an extension of Σ in the following sense: the states Q of the automaton are added to the set P of places of Σ, and initially $M_0 \cup \{q_0\}$ is marked. The accepting states $\mathcal{F}$ become the accepting places, and for each transition $(q, \Pi', q') \in \delta$, we add q to the preset and q' to the postset of every transition t of the system, with $v(t) = \Pi'$.

Proposition 3. *Let Σ_p be the synchronized product of Σ and $A_{\neg\varphi}$ as defined above. The system Σ satisfies φ in action-based semantics iff Σ_p is empty.*

Observation. In a state-based interpretation, the automaton $A_{\neg\varphi}$ can be seen as a process observing the marking sequences of Σ. Clearly, it suffices to observe only the places that appear as atomic propositions in φ as stated by Lemma 4 below.

Let $\xi = M_0 M_1 \ldots$ be the infinite marking sequence corresponding to a run γ, and $Q \subseteq P$ a set of places. By $M_i^Q := M_i \cap Q$ we denote the restriction of M_i onto the places in Q, and we define $\xi^Q := M_0^Q M_1^Q \ldots$.

Lemma 4. *If φ is an LTL formula and $Q \subseteq P$ a set of propositions such that $\langle\varphi\rangle \subseteq Q$, then $\xi \models \varphi$ iff $\xi^Q \models \varphi$ for every ω-word ξ over 2^P.*

We will construct Σ_p in such a way that the automaton and the system alternate their moves. Intuitively, if (q, P', q') is a transition of $A_{\neg\varphi}$, the automaton tests if the current marking is P'. In this case it moves from q to q' and enables Σ to make a move, which makes its move and again enables the automaton to observe the current marking. The mutual enabling is implemented using two "scheduler" places s_f, s_s. If s_f (s_s) is marked, then the automaton (the system) has to move next. The automaton must observe M_0, so initially s_f is marked.

The testing of a marking is done by connecting the relevant places with the transitions of the automaton. If $d = (q, P', q')$ is a transition of $A_{\neg\varphi}$, we add all the places in $P' \subseteq P$ to the preset and to the postset of d. Thus, d can occur if all places in P' are marked, and after d occurred, again P' is marked. In general, however, this is insufficient: the automaton changes from q to q' only if the current marking is *equal* to P', in particular, if no proposition in $P \setminus P'$ belongs to the marking. By simply adding P' to $^\bullet d$, the transition d can occur, no matter whether any place $p \notin P'$ is marked or not. Therefore, we have to presuppose some *complementary places* in Σ.

Let $p, \overline{p} \in P$. The place $\overline{p}$ is the complement of p, if $\overline{p}^\bullet = {}^\circ p$, $^\bullet\overline{p} = p^\circ$, and $M_0(\overline{p}) = 1 - M_0(p)$. Thus, $p \in M$ iff $\overline{p} \notin M$ for every reachable state M. Due to Lemma 4, only the propositions (places, here) that appear in φ are relevant. So, we have to extend Σ by a complementary place for every place in $\langle\varphi\rangle$. Note that this extension has no influence on the system's behaviour. Let us denote by $Obs(\varphi) := \{p, \overline{p} \mid p \in \langle\varphi\rangle\}$ the set of *observed* places.

The Büchi net Σ_p then is defined as follows: the places are $P \cup Q \cup \{s_s, s_f\}$, the transitions are $T \cup \delta$, the initial marking is $M_0 \cup \{q_0\} \cup \{s_f\}$, the accepting places are $\mathcal{F}$, and the flow relation is F, extended by

- $(s_s, t), (t, s_f)$ for all $t \in T$, and $(s_f, d), (d, s_s)$ for all $d \in \delta$;
- $(q, d), (d, q')$ for every $d = (q, P', q') \in \delta$, as well as $(p, d), (d, p)$ and $(\overline{r}, d), (d, \overline{r})$ for all $p \in P'$ and $r \in \langle\varphi\rangle \setminus P'$.

The construction is sketched in Fig. 2 for a transition $d = (q, \{p_1\}, q')$ of the automaton $A_{\neg\varphi}$ where $\langle\varphi\rangle = \{p_1, p_2\}$.

Proposition 5. *Let Σ_p be the observation product of Σ and $A_{\neg\varphi}$ as defined above. The system Σ satisfies φ in state-based semantics iff Σ_p is empty.*

Relaxing the observation. Since $A_{\neg\varphi}$ behaves strictly sequentially, each observation introduces causal dependency on observed transitions, which ruins the benefits of any partial-order representation. However, restricting ourselves to *stutter-invariant properties*, it is sufficient to observe only all the *visible* transitions [18]. A transition t is visible iff $^\circ t$ or t° contains some place of $Obs(\varphi)$.

In [13] it has been shown that stutter-invariant properties are expressed by the "next-free" fragment of LTL, i.e., LTL without the next step operator X. In this fragment one cannot distinguish between the ω-word $\xi = x_0 x_1 \ldots$ and an ω-word ξ' similar to ξ except that any of the x_is are repeated finitely often. Let $\mu(\xi)$ denote the ω-word where every maximal finite subsequence $xx \ldots x$ in ξ is substituted by x. Two ω-words ξ, ξ' are *stutter-equivalent* if $\mu(\xi) = \mu(\xi')$.

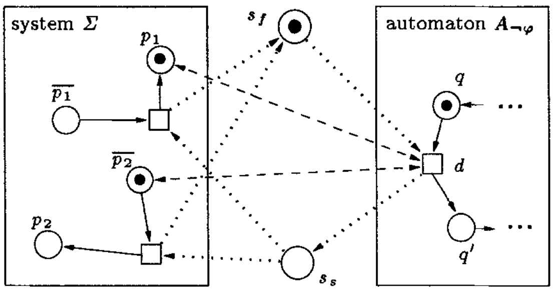

Fig. 2. The observation construction.

Lemma 6 ([13]). *If φ is a next-free LTL formula and ξ, ξ' are stutter-equivalent ω-words then $\xi \models \varphi$ iff $\xi' \models \varphi$.*

In the construction of the product this means, that *only the visible* system transitions and *all transitions of the automaton* are strictly alternating, while all concurrency among the non-visible transitions is preserved. The *reduced product* Σ_r thus is defined like Σ_p, except that for every non-visible transition t, the arcs (s_s, t) and (t, s_f) are discarded.

Unfortunately, with this construction it is possible that some run γ satisfies $\neg\varphi$, but it is *not* accepting. This is the case if only finitely many visible transitions occur in γ. Then the place s_s remains marked forever and thus no transition of the automaton will occur anymore. However, we have:

Proposition 7. *Let $\gamma = M_0 t_1 M_1 t_2 \ldots$ be a run of Σ_r, and t_{i_j} the jth occurrence of a transition of Σ in γ. For each state M_j of γ, let $P_j := M_j \cap P$ the restriction of M_j onto system places. The* projection *of γ onto system nodes then is defined as $Proj(\gamma) := P_0 t_{i_1} P_{i_1} t_{i_2} P_{i_2} \ldots$*

1. *If γ is a run of Σ_r then its projection $Proj(\gamma)$ onto system nodes is a run of Σ, and $\gamma \models \neg\varphi$ iff $Proj(\gamma) \models \neg\varphi$.*
2. *For every run γ' of Σ satisfying $\neg\varphi$, there exists a run γ of Σ_r, such that $\gamma' = Proj(\gamma)$ is the projection of γ onto system nodes, and $\gamma \models \neg\varphi$.*
3. *If γ is a run of Σ_r containing infinitely many visible transitions then the projection $Proj(\gamma)$ of γ satisfies $\neg\varphi$ iff γ is accepting.*

Model checking LTL. The fact that acceptance requires infinitely many visible transition occurrences may look like a drawback. To cope with this problem, we have to apply a *2-phase model checking* procedure ("mc_unf"):

Phase 1. We construct the reduced product Σ_r of Σ and $A_{\neg\varphi}$, and compute its finite prefix Fin_{Σ_r}. Now we build the graph G, applying BuildGraph(T_a) (Fig. 1), where T_a is the set of accepting transitions of $A_{\neg\varphi}$. *Additionally* we apply BuildGraph(T_b), where T_b is the set of *all* the automaton transitions, not only the accepting ones. The edges of the graph now may be labelled with *a and/or b*. Now, Tarjan's depthfirst search algorithm [1] determines the maximal strongly connected components (scc) of G. Each scc containing an a-labelled edge represents an accepting run. If such an scc is found then we reconstruct the corresponding violating run of the system and stop, else consider Phase 2.

Phase 2. We discard each scc containing an a- or b- or a, b-labelled edge. The *remaining* sccs correspond to infinite runs of Σ_r containing only finitely many occurrences of automaton (and, consequently, of *visible*) transitions. This means, that after finitely many steps, $M(p)$ remains unchanged for all observed places $p \in Obs(\varphi)$. Since each cut-off e in a remaining scc can be considered as being initial in G, each e in scc refers to a certain set of runs. All these e-runs have a unique, *last* reached automaton state q_e in common, determined by $q_e = Mark([e]) \cap Q$. That is, in all possible e-runs of Σ_r, the automaton get stuck in state q_e. Further note that the set $P_e := Mark([e]) \cap Obs(\varphi)$ of observed places is the last, forever-unchanged, "relevant" submarking in all these runs.

Thus, for each cut-off e in all the remaining sccs, we have to determine q_e and P_e, and have then to investigate if P_e allows an accepting cycle of the automaton starting at q_e. If so, the corresponding violating run of Σ can be reconstructed. Note that the Phase 2 needs only a fraction of the verification effort since $A_{\neg\varphi}$ is usually small compared with the size of the product.

Experimental results. A prototype implementation of the proposed method, using the very efficient unfolding procedure of [7], yielded promising results. Mainly, we observed that even large systems, synchronized with small automata, e.g. for (the negation of) the usual liveness property $\Box(p \Rightarrow \Diamond q)$, result in reasonably small prefixes. Liveness (resp. non-liveness) of Peterson's, Dekker's and Lamport's mutex-algorithms were checked within less than two seconds.

We also considered (the reactive version of) a leader election algorithm for a ring topology, described in [3]. The modelling is due to Stephan Melzer [6].

Essentially, the system consists of n processes, connected via a token ring. Each of the processes can be identified by its unique process number. The algorithm (in the reactive version) strives for repeated determination of a designated process, i.e., the one with the maximal number. We considered the liveness property $\Box\Diamond(\mathit{elected} = \mathsf{true})$, expressing that infinitely often a designated process is found. The results are presented in Table 1. They are extremely positive, since for all n, the prefix contained only one cut-off event. The complementary property $\Diamond\Box(\mathit{elected} = \mathsf{false})$ was shown to be not valid both by Spin [11] and our implementation in a second. All experiments were done within the PEP-tool [2].

n	Σ		Σ_r		Fin		time (sec.)	
	$\|P\|$	$\|T\|$	$\|P\|$	$\|T\|$	$\|B\|$	$\|E\|$	mc_unf	Spin
5	88	84	94	88	179	93	0.8	2.2
6	110	106	116	110	215	113	0.9	9.3
7	142	138	148	142	251	133	0.9	39.4
8	160	156	166	160	287	153	0.9	—[1]

Table 1. Results and comparison with SPIN for leader election.

[1] 64 MB main memory are exceeded.

5 Conclusion

Discussion. We have presented a method for model checking LTL in the framework of safe Petri nets, adopting the well-known automata-theoretic approach. We have shown how the finite prefix can be used for detecting the emptiness of a net with acceptance capabilities, and how to construct the product net of a given net system and a Büchi automaton, exhibiting enough concurrency to take profit from the partial order representation of behaviour.

How efficient is the proposed method? If one suppose a setting where only a small fraction of the behaviour influences the specification (i.e. there are few visible transitions), the "degree" of concurrency will remain high enough to exploit the advantages of net unfoldings, which are in some cases exponentially smaller than the global state space.

However, until now there is no better way to handle fairness constraints than including them into the formula, i.e., checking "$\varphi_{fair} \Rightarrow \varphi$". This may increase the number of visible transitions, and so possibly a larger part of the behaviour will be sequentialized. A more efficient method for treating fairness is desirable.

For a fast detection of violating runs when dealing with systems under development, we want to investigate an on-the-fly construction of the graph G: whenever a new cut-off event is detected during the unfolding procedure, the (partial) graph has to be "updated" and searched for an accepting cycle. If such a cycle is found, the unfolding needs not to be constructed further.

Related work. A closely related approach recently has been investigated in [6], also considering a product of a given safe net system and a Büchi automaton. There, a semidecision test is considered, that is a procedure which may answer "yes", in which case Σ satisfies the specification, or "don't know". The procedure works without ever constructing the state space, but uses structural net theory.

The common idea of the partial order methods proposed so far [10,16–19], bases on the observation that the order of execution of *concurrent* actions in many cases is irrelevant for the checked property φ. Intuitively, when several concurrent actions are enabled at a state, only some of them are selected, such that certain computation sequences and states may be discarded, yielding a *reduced system.* Since φ may be sensitive to certain interleavings of *visible* actions [18], all concurrent visible actions are considered to be causally dependent.

So-called *on-the-fly* methods are investigated in [10,17,19]. There, the Büchi automaton is incorporated in the construction of a *reduced product.* That is, instead of first reducing the system and then building the product, the Büchi automaton is used to guide the further exploration of concurrently enabled transition sets. In some cases, it is possible to detect accepting cycles during the construction, and thus not the entire product needs to be built.

In [12] it was shown that the visibility of actions (and thus the need to consider them dependent) may diminish during the construction of the reduced product, sometimes resulting in even better reduction.

Acknowledgments. I'd like to thank Stephan Melzer and Javier Esparza for many fruitful discussions. Special thanks to Stephan Melzer for the support with the implementation and the experiments.

References

1. A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. E. Best, H. Fleischhack (eds.). PEP: Programming Environment based on Petri nets. Technical report, University of Hildesheim, 1995.
3. E. Chang, R. Roberts. An Inproved Algorithm for Decentralised Extrema-finding in Circular Distributed Systems. *Communication of the ACM*, 22(5):281–283, 1979.
4. J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.
5. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.
6. J. Esparza, S. Melzer. Model Checking LTL Using Constraint Programming. In *Proc. of 18th Int. Conf. on Application and Theory of Petri Nets*, LNCS 1248, pp. 1–20, 1997.
7. J. Esparza, S. Römer, W. Vogler. An Improvement of McMillan's Unfolding Algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, LNCS 1055, pp. 87–106, 1996.
8. R. Gerth, D. Peled, M. Vardi, P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Protocol Specification, Testing, and Verification PSTV'95*, pp. 3–18, 1995.
9. P. Godefroid, G.J. Holzmann. On the Verification of Temporal Properties. In *Protocol Specification, Testing, and Verification PSTV'93*, 1993.
10. P. Godefroid, P. Wolper. A Partial Approach to Model Checking. In *Proc. of 6th IEEE Symp. on Logic in Computer Science*, pp. 406–415, 1991.
11. G.J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
12. I. Kokkarinen, D. Peled, A. Valmari. Relaxed Visibility Enhances Partial Order Reduction. In *Proc. of 9th Computer-Aided Verification CAV'97*, LNCS 1254, pp. 328–339, 1997.
13. L. Lamport. What good is temporal logic? *Information Processing 83*, pp. 657–668, 1983.
14. K.L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. 4th Workshop on Computer-Aided Verification*, LNCS 663, pp. 164–174, 1992.
15. M. Nielsen, G. Plotkin, G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, 13(1):85–108, 1980.
16. D. Peled. All from one, one for all: on model checking using representatives. In *Proc. of 5th Computer-Aided Verification CAV'93*, LNCS 697, pp. 409–423, 1993.
17. D. Peled. Combining partial order reductions with on-the-fly model checking. In *Proc. of 6th Computer-Aided Verification CAV'94*, LNCS 818, pp. 377–390, 1994.
18. A. Valmari. A Stubborn Attack on State Explosion. *Formal Methods in System Design*, 1:297–322, 1992.
19. A. Valmari. On-the-fly Verification with Stubborn Sets. In *Proc. of 5th Computer-Aided Verification CAV'93*, LNCS 697, pp. 397–408, 1993.
20. M.Y. Vardi, P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of 1st IEEE Symp. on Logic in Computer Science*, pp. 322–331, 1986.
21. P. Wolper. On the relations on programs and computations to models of temporal logic. In *Proc. of Temporal Logic in Specification*, LNCS 398, pp. 75–123, 1989.
22. P. Wolper, M.Y. Vardi, A.P. Sistla. Reasoning about infinite computation paths. In *Proc. of 24th IEEE Symp. on Foundations of Computer Science*, pp. 185–194, 1983.