Model Checking LTL Properties of High-Level Petri Nets with Fairness Constraints

Timo Latvala*

Laboratory for Theoretical Computer Science
Helsinki University of Technology
P.O. Box 9700, FIN-02015 HUT, Espoo, Finland.
`Timo.Latvala@hut.fi`
`http://www.tcs.hut.fi/Personnel/timo.html`

Abstract Latvala and Heljanko have presented how model checking of linear temporal logic properties of P/T nets with fairness constraints on the transitions can be done efficiently. In this work the procedure is extended to high-level Petri Nets, Coloured Petri Nets in particular. The model checking procedure has been implemented in the MARIA tool. As a case study, a liveness property of a sliding window protocol is model checked. The results indicate that the procedure can cope well with many fairness constraints, which could not have been handled by specifying the constraints as a part of the property to be verified.

Keywords: Model checking, fairness, LTL, high-level Petri Nets.

1 Introduction

Model checking [2, 23] has established itself as one of the most useful methods for reasoning about the temporal behavior of Petri Nets. Currently there are several Petri Net tools which offer model checking of either linear or branching time properties of Petri Nets (see e.g. [29, 37]).

Model checking liveness properties differs from model checking safety properties. In many cases certain unwanted behaviors of the model must be ignored, to facilitate model checking of liveness properties. This is usually done using fairness assumptions [7]. For Petri Nets this means that the behavior of a transition or several transitions is restricted according to the fairness assumption. The only support Petri Net model checkers so far have offered for this is by model checking properties with formulas of the form "*fairness* $\Rightarrow$ *property*". Using many fairness assumptions makes the formula long and therefore quickly makes the model checking intractable, since the model checking problem is PSPACE-complete [30] in the size of the formula.

Latvala and Heljanko presented in [19] how P/T nets with fairness constraints on the transitions can be model checked efficiently by giving semantics to the

* The financial support of TEKES, Nokia Research Center, Nokia Networks, Genera, EKE Electronics and Helsinki Graduate School in Computer Science and Engineering (HeCSE) is gratefully acknowledged.

P/T net with a fair Kripke structure (FKS) [14]. The FKS is then model checked with a procedure similar to the one presented in [14]. It can, however, be difficult to model large systems with P/T nets, and therefore there is a need to extend the procedure to high-level nets.

The main contributions and results of this work are the following. A procedure for model checking high-level Petri Nets with fairness constraints on the transitions is presented. The procedure is a generalization of the procedure in [19]. Included is also the proof of correctness of the construction, which were only briefly touched upon in [19]. The procedure has been implemented in the MARIA analyzer. Using a model of a sliding window protocol, the implementation is tested. A liveness property of the protocol is verified with a window size up to 11, without any reduction methods. This could not have been done with the fairness constraints as part of the property to be verified.

The rest of the paper is structured as follows. In Section 2 Coloured Petri Nets are defined and Fair CPNs are introduced. Section 3 covers the necessary automata theory and presents the model checking procedure. The implementation of the model checker is covered in Section 4. The sliding window protocol is modeled and analyzed in Section 5. Section 6 concludes the paper.

2 Coloured Petri Nets

2.1 Definition of Coloured Petri Nets

The definition follows quite faithfully the definition of Coloured Petri Nets in [12]. CPNs were chosen because they are relatively simple to define while still being high-level Petri nets. Also, the fact that they are well-known contributed to their choice. The results to be presented later can easily be generalized to other high-level Petri net classes.

No concrete syntax and semantics for the net expressions will be given. We will however assume that it exists so that the following concepts are well defined.

- $\mathit{Type}(e)$ - The type of the expression e
- $\mathit{Var}(e)$ - The set of variables in an expression e.
- A *binding* $b(v)$ associates with each variable v a value of the type of the variable.
- $e\langle b\rangle$ - The value obtained by evaluating the expression e with the binding b.

Definition 1. *A tuple $\Sigma = \langle \Pi, P, T, A, N, C, E, G, M_0\rangle$ is a* ***Coloured Petri Net*** *(CPN) [12] where,*

i.) Π *is a finite set of non-empty types or* ***colour sets****.*
ii.) P *is a finite set of* ***places****.*
iii.) T *is a finite set of* ***transitions****, such that* $P \cap T = \emptyset$.
iv.) A *is a finite set of* ***arcs****, such that* $P \cap A = T \cap A = \emptyset$.
v.) $N : A \to (P \times T) \cup (T \times P)$ *is a* ***node function****.*
vi.) $C : P \to \Pi$ *is a* ***colour function****.*

*vii.) E is an **arc expression function**, defined from arcs to expressions, such that $\forall a \in A : E(a)\langle b \rangle$ is a multi-set over the colour of the place component of the arc, for all legal bindings b.*

*viii.) G is a **guard function**, defined from transitions to expressions, such that $G(t)\langle b \rangle \in \{true, false\}$ for any legal binding b and $t \in T$.*

*ix.) M_0 is an **initial marking**.*

We define the following notations:

- $A(x) = \{a \in A \mid \exists x' \in P \cup T : [N(a) = (x, x') \vee N(a) = (x', x)]\}$
- $A(x_1, x_2) = \{a \in A \mid N(a) = (x_1, x_2)\}$
- $\forall t \in T : Var(t) = \{v \mid v \in Var(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$.
- $\forall (x_1, x_2) \in (P \times T \cup T \times P) : E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$.

$A(x)$ returns the set of surrounding arcs, i.e. the arcs that have x as a source or a destination, for a given node x. $A(x_1, x_2)$ return the arcs which are between the nodes x_1 and x_2. $Var(t)$ is the set of variables of t, while $E(x_1, x_2)$ is the expression of (x_1, x_2) and returns the multi-set sum of all expressions connected to the arcs which have x_1 and x_2 as nodes.

Definition 2. *A **token element** is a pair $(p, c) \in P \times C(p)$. The set of all token elements is denoted by $\mathcal{TE}$. A **marking** is a multi-set over $\mathcal{TE}$.*

Because each marking defines a unique function $M(p)$, which maps each place to a multi-set over the colour set of the place, a marking is usually presented as a function on P.

Definition 3. *A **binding** of a transition $t \in T$ is a binding function on $Var(t)$ such that $\forall v \in Var(t) : b(v) \in Type(v)$ and $G(t)\langle b \rangle = true$. We denote the binding $t\langle b \rangle$ and call $t\langle b \rangle$ an **instance** of t. A transition instance $t\langle b \rangle$ is **enabled** in a marking M iff $\forall p \in P : E(p, t)\langle b \rangle \leq M(p)$.*

The function $en(M)$ returns the transition instances which are enabled in the marking M. If a transition instance $t\langle b \rangle \in en(M)$ it can **occur** changing M into another marking M' which is given by

$$\forall p \in P : M'(p) = M(p) - E(p, t)\langle b \rangle + E(t, p)\langle b \rangle$$

Hence M' is *reachable* from M, which we denote by $M \overset{t\langle b \rangle}{\to} M'$.

The behavior of the net is given by the Kripke structure of the net.

Definition 4. *The **Kripke structure** of a CPN Σ is a triple $K = \langle S, \rho, s_0 \rangle$, where S is the set of markings, ρ is the transition relation, and s_0 is the initial marking. S and ρ are defined inductively as follows.*

1. *$s_0 = M_0 \in S$*
2. *If $M \in S$ and $M \to M'$ then $M' \in S$ and $(M, M') \in \rho$. If $M \in S$ and $en(M) = \emptyset$ then $(M, M) \in \rho$.*
3. *S and ρ have no other elements.*

The executions of the net are infinite sequences of markings $\xi = M_0 M_1 M_2 \ldots$, for which $(M_i, M_{i+1}) \in \rho$ for all $i \geq 0$.

2.2 Petri Nets and Fairness

Fairness can be a useful abstraction, when one does not want to model the details of the scheduling required in the implementation of the system. Using *fairness assumptions* for the transitions is perhaps the most convenient way one can restrict the set of legal executions to the desired ones when verifying liveness properties. Fairness does not affect safety properties. The most common fairness assumptions are known as *weak fairness* and *strong fairness*. In [7] they are defined using the familiar concepts of *enabledness* and *occurrence* of the relevant events. An event is *weakly fair* when continuous enabledness implies that the event occurs infinitely often. For some situations, however, weak fairness is not enough. In this situation *strong fairness* might be appropriate. Strong fairness assumes that if an event is infinitely often enabled then it will occur infinitely often.

There is some related work which has combined model checking and fairness constraints. Latvala and Heljanko presented how P/T nets can be extended with fairness constraints and model checked. In [6] Emerson and Lei presented how to cope with strong fairness constraints when model checking CTL properties of a Kripke structure. A similar method was used to design a BDD based algorithm when the property was given as an automaton in [10] while in [14] a procedure for model checking LTL properties using BDDs with both weak and strong fairness constraints was presented.

The traditional way of incorporating fairness when model checking Petri nets exploits the fact that fairness is expressible by LTL. First, one usually has to add places and transitions to the model so that the occurrence of transitions is explicitly visible in the reachable markings. These modifications in a sense model a scheduler and have to be made because the state based version of LTL can only express properties of markings and thus cannot express properties of the transitions unless they are explicitly visible in the Kripke structure. The model is then verified by checking the formula "$fairness \Rightarrow property$". This approach has several drawbacks. The two most obvious ones are that adding places and transitions might increase the size of the state space, and that the size of the Büchi automaton representing the property can grow exponentially in the number of fairness constraints (see e.g. [8]). A more subtle drawback is that adding the scheduler often reduces the concurrency in the model, which may affect the performance of some partial order methods (see e.g. [36]).

Let $\xi = M_0 \stackrel{t_0\langle b_0\rangle}{\rightarrow} M_1 \stackrel{t_1\langle b_1\rangle}{\rightarrow} \cdots$, with $M_i \stackrel{t_i\langle b_i\rangle}{\rightarrow} M_{i+1}$, be an execution of a CPN. A fairness function F is a function from transitions to boolean valued expressions. We define $EN_{F,i}(\xi) = \mathit{true}$ if $\exists t\langle b\rangle \in en(M_i) : F(t)\langle b\rangle = \mathit{true}$; otherwise $EN_{F,i}(\xi) = \mathit{false}$. Also let $OC_{F,i}(\xi) = \mathit{true}$ if $F(t_i)\langle b_i\rangle = \mathit{true}$; otherwise $OC_{F,i}(\xi) = \mathit{false}$. These two functions return true if a transition instance is enabled ($EN_{F,i}(\xi)$) or has occured ($OC_{F,i}(\xi)$) w.r.t. a fairness function. Denote the quantifier "there exist infinitely many" by $\exists^\omega$ and then let $InfEN_F(\xi)$ and $InfOC_F(\xi)$ be defined in the following way:

$$InfEN_F(\xi) = \begin{cases} true & \text{if } \exists^\omega i:\ EN_{F,i}(\xi) = true \\ false & \text{otherwise.} \end{cases}$$

$$InfOC_F(\xi) = \begin{cases} true & \text{if } \exists^\omega i:\ OC_{F,i}(\xi) = true \\ false & \text{otherwise.} \end{cases}$$

Now strong and weak fairness can be defined w.r.t. a fairness function. An execution is strongly fair if the set of transitions instances, defined by a fairness function F, are infinitely often enabled implies that they occur infinitely often [12].

$$InfEN_F(\xi) \Rightarrow InfOC_F(\xi).$$

The definition for weak fairness is that persistent enabling implies an occurrence [12].

$$\forall i \in \mathbb{N} : EN_{F,i}(\xi) \Rightarrow \exists k \geq i : [\neg EN_{F,k}(\xi) \vee OC_{F,k}(\xi)]$$

The semantics of our fairness constraints are equivalent to those presented by Jensen in [12], but the notation is a little different.

We now extend CPNs with fairness constraints on the transitions.

Definition 5. *A* ***fair CPN*** *(FCPN) is a triple $\Sigma_F = \langle \Sigma, WF, SF \rangle$ where Σ is a CPN and $WF = \{wf_1, \ldots wf_k\}$ a set of weak fairness functions, where wf_i is a function from the set of transitions to expressions such that $Var(wf_i(t)) \subseteq Var(t)$ for all $t \in T$ and $wf_i(t)\langle b \rangle \in \{true, false\}$ for any legal binding b of the expression. $SF = \{sf_1, \ldots, sf_m\}$ is the corresponding set of strong fairness functions with similar restrictions. An execution is an infinite sequence of markings and transition instances $\xi = M_0 \overset{t_0\langle b_0 \rangle}{\rightarrow} M_1 \overset{t_1\langle b_1 \rangle}{\rightarrow} \cdots$, such that for all $i \geq 0$ there exists $t_i\langle b_i \rangle$ for which $M_i \overset{t_i\langle b_i \rangle}{\rightarrow} M_{i+1}$ and ξ obeys the fairness constraints defined previously for all weak and strong fairness functions in WF and SF.*

The expression of a fairness function is true for all instances, which should be treated as equivalent; if one of the instances is fair, the fairness requirement has been satisfied.

In the following discussion we only consider finite state and finitely branching net systems. For analysis purposes these execution based semantics are not very convenient. Something similar to a Kripke structure is needed. However, the behavior of a FCPN cannot be described accurately by a Kripke structure because the fairness constraints are not taken into account in any way. Some mechanism is needed so that unfair executions can be rejected, and only those which conform to the fairness constraints are accepted. One way of doing this is by extending the definition of a Kripke structure to a fair Kripke structure. It could have been possible to extend a labeled transition system (LTS), but as they in [14] use a state based approach and the fastest available Streett emptiness algorithm [24] is state based, extending Kripke structures seemed more appropriate.

Let $\rho = s_0 s_1 s_2 \ldots \in S^\omega$ be an infinite sequence of states. The set of states occuring infinitely often in the sequence is given by:

$$Inf(\rho) = \{s \in S^\omega \mid \exists^\omega :\ \rho(i) = s\}$$

Definition 6. *A **fair Kripke structure (FKS)** [14] is a quintuple $K_F = \langle S, \rho, s_0, \mathcal{W}, \mathcal{S}\rangle$, where S is a set of states, $\rho \subseteq S \times S$ is a transition relation and $s_0 \in S$ is the initial state. The fairness requirements are defined by a set of weak fairness requirements*[1] *$\mathcal{W} = \{J_1, J_2, \ldots, J_k\}$ where $J_i \subseteq S$, and a set of strong fairness requirements, $\mathcal{S} = \{\langle L_1, U_1\rangle, \ldots, \langle L_m, U_m\rangle\}$ where $L_i, U_i \subseteq S$. An execution is an infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots \in S^\omega$, where s_0 is the initial state, and for all $i \geq 0$, $(s_i, s_{i+1}) \in \rho$. Computations, i.e. fair executions of the system, are sequences that obey the fairness requirements $\bigwedge_{i=1}^{k} Inf(\sigma) \cap J_i \neq \emptyset$ and $\bigwedge_{i=1}^{m} (Inf(\sigma) \cap L_i = \emptyset \vee Inf(\sigma) \cap U_i \neq \emptyset)$.*

An execution σ is a computation if both the weak and strong fairness requirements are satisfied. From each weak fairness set, at least one state occurs infinitely often in the execution and for each strong fairness if a state from a set L_i occurs infinitely often, also state from U_i must occur infinitely often in the execution.

Using the fairness requirements of the FKS, it is possible to only accept the computations which adhere to the fairness constraints on the transitions for FCPN. However, generating a FKS from a FCPN is not completely straightforward. For the same reason that a normal CPN must sometimes be modified in order for the LTL formulas to be able to express the fairness assumptions, a FKS cannot simply be a normal Kripke structure where we have added some fairness sets. The occurrence of transition instances must be made explicit in the FKS. Here, this is done by adding an intermediate state for each occurrence of a transition instance in the FKS. We make the transition "visible" by adding the intermediate state. For instance, if in the normal Kripke structure the marking M_j is followed by M_{j+1} when taking the transition instance $t\langle b\rangle$, in the FKS this sequence will have an intermediate state. If the intermediate state is denoted by $M_{t\langle b\rangle}$ the sequence will be $M_j M_{t\langle b\rangle} M_{j+1}$. With the intermediate states added it is now possible to use the weak and strong fairness sets to ensure that only executions which obey the fairness constraints on the transitions are considered legal.

The states of the FKS are defined as pairs $\langle M, t\langle b\rangle\rangle$ so that the intermediate states can be distinguished from "normal" states. The special symbol $\perp$ replaces the transition instance if the state is not an intermediate state. Hence, to obtain a FKS $K_F = \langle S, \rho, s_0, \mathcal{W}, \mathcal{S}\rangle$ from a FCPN $\Sigma_F = \langle \Sigma, WF, SF\rangle$, we define S and ρ inductively as follows:

1. $s_0 = \langle M_0, \perp\rangle \in S$.

[1] In order to have consistent terminology weak and strong fairness are used instead of justice and compassion as in [14].

2. If $\langle M, \perp\rangle \in S$ and $M \stackrel{t\langle b\rangle}{\rightarrow} M'$ then, $\langle M', t\langle b\rangle\rangle \in S$, $\langle M', \perp\rangle \in S$ and $(\langle M, \perp\rangle, \langle M', t\langle b\rangle\rangle) \in \rho$, $(\langle M', t\langle b\rangle\rangle, \langle M', \perp\rangle) \in \rho$. If $\langle M, \perp\rangle \in S$ and $en(M) = \emptyset$ then $(\langle M, \perp\rangle, \langle M, \perp\rangle) \in \rho$.
3. S and ρ have no other elements.

The weak fairness sets and the strong fairness sets are defined as:

1. For each $wf_i \in WF$ the weak fairness set is
   - $J_i = \{\langle M, \perp\rangle \in S \mid \forall t\langle b\rangle \in en(M) : wf_i(t)\langle b\rangle = false\} \cup \{\langle M', t\langle b\rangle\rangle \in S \mid wf_i(t)\langle b\rangle = true\}$.
2. For each $sf_i \in SF$ the strong fairness sets are
   - $L_i = \{\langle M, \perp\rangle \in S \mid \exists t\langle b\rangle : t\langle b\rangle \in en(M) \wedge sf_i(t)\langle b\rangle = true\}$ and
   - $U_i = \{\langle M', t\langle b\rangle\rangle \in S \mid sf_i(t)\langle b\rangle = true\}$.

We are now ready prove that the construction of the FKS is correct, in the sense that the semantics of the fairness constraints are as we wanted.

Theorem 1. *Let Σ_F be a FCPN. $\xi = M_0 \stackrel{t_0\langle b_0\rangle}{\rightarrow} M_1 \stackrel{t_1\langle b_1\rangle}{\rightarrow} \cdots$ is a fair execution of Σ_F, if and only if the the FKS of the FCPN has a computation ξ'. (Under the assumption that Σ_F is finite state and finitely branching.)*

Proof. See [18].

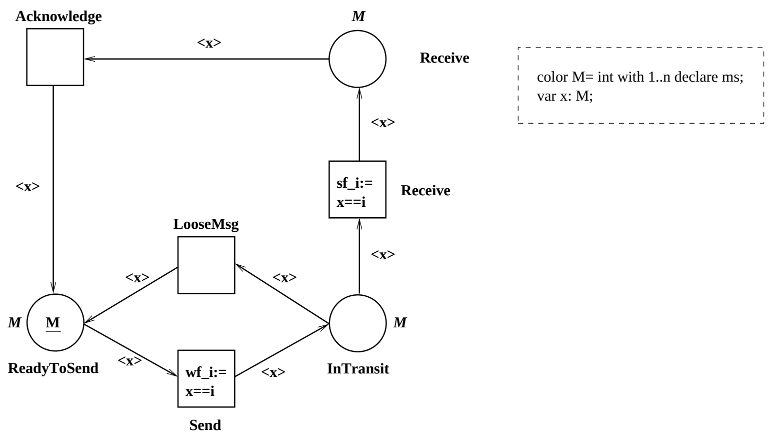

Figure 1. FCPN model of the sender-receiver system.

Example. Consider a simple system consisting of a sender and a receiver. The sender can send n different messages to the receiver and the receiver acknowledges the messages. The communication could be modeled so that a sent message can be lost, while acknowledgements always come through. A Petri Net model checker would declare that if a message is ready to be sent, it would not always be received. A closer study of the counterexamples reveals two reasons.

A message may not be sent because only one type of messages are sent even if though all types are ready to be sent, or the channel looses all messages. This could be remedied by including scheduler in the model, however, a simpler solution would be to use some fairness constraints on the transitions. By giving the transition "Send" a weak fairness constraint w.r.t the message type and the transition "Receive" a strong fairness constraint w.r.t the message type, the model checker would now report that the property holds. The weak fairness constraint ensures progress, so that no message type is left unsent when it is ready. The strong fairness constraint ensures that a message gets through eventually if it is resent. A FCPN model of the fair system can be found in Figure 1.

3 Model Checking LTL

3.1 ω-Automata

The close connection between automata on infinite words and LTL is used by many model checking procedures. Here the necessary automata theory is introduced and the most important terms are defined.

Büchi automata are the basic theoretical construction for every LTL model checker which uses the automata theoretic approach.

Definition 7. *A* ***labeled generalized Büchi automaton*** *(LGBA) [3] is a tuple $\mathcal{A} = \langle Q, \Delta, I, \mathcal{F}, \mathcal{D}, \mathcal{P} \rangle$, where Q is a finite set of states, $\Delta \subseteq Q \times Q$ is the transition relation, I is a set of initial states, $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ with $F_i \subseteq Q$ is a finite set of acceptance sets, $\mathcal{D}$ some finite domain (in LTL model checking $\mathcal{D} = 2^{AP}$ for some finite set of atomic propositions AP) and $\mathcal{P} : Q \mapsto 2^{\mathcal{D}}$ is a labeling function. A run of $\mathcal{A}$ is an infinite sequence of states $\rho = q_0 q_1 q_2 \ldots$ such that $q_0 \in I$ and for each $i \geq 0$, $(q_i, q_{i+1}) \in \Delta$.*

Let the operator $Inf(\rho)$ be defined similarly for a run as for an execution. A run ρ is accepting if for each acceptance set $F_i \in \mathcal{F}$ there exists at least one state $q \in F_i$ that appears infinitely often in ρ, i.e. $Inf(\rho) \cap F_i \neq \emptyset$ for each $F_i \in \mathcal{F}$. An infinite word $\xi = x_0 x_1 x_2 \ldots \in \mathcal{D}^\omega$ is accepted iff there exists an accepting run $\rho = q_0 q_1 q_2 \ldots$ of $\mathcal{A}$ such that for each $i \geq 0$, $x_i \in \mathcal{P}(q_i)$. If $\mathcal{F} = \{F_1\}$ the LGBA corresponds to an ordinary Büchi automaton.

With Streett automata it is possible to extend the LTL model checking procedure to also cope with strong fairness in an efficient manner.

Definition 8. *A* ***Streett automaton*** *(see [35] for an arc labeled version) is a tuple $\mathcal{A} = \langle Q, \Delta, I, \Omega, \mathcal{D}, \mathcal{P} \rangle$, where Q, Δ, I, $\mathcal{D}$ and $\mathcal{P}$ have the same meanings as above. $\Omega = \{(L_1, U_1), \ldots, (L_k, U_k)\}$ with $L_i, U_i \subseteq Q$ is a set of pairs of acceptance sets. A run of a Streett automaton is defined in the same way as for an LGBA. The Streett automaton accepts a run $\rho = q_0 q_1 q_2 \ldots$ if $\bigwedge_{i=1}^{k} (Inf(\rho) \cap L_i = \emptyset \vee Inf(\rho) \cap U_i \neq \emptyset)$.*

We can read the acceptance condition as that the automaton accepts when "for each i, if some state in L_i is visited infinitely often, then some state in U_i is visited infinitely often". We define the set of infinite words accepted by $\mathcal{A}$ analogously to the LGBA case, using the new acceptance condition Ω.

Streett automata and generalized Büchi automata both accept the class of ω-regular languages, however, there is no polynomial translation from a Streett automaton to a Büchi automaton (see e.g. [28]). The converse can easily be done by letting $L_i = Q$ and $U_i = F_i$.

The set of ω-words the automaton $\mathcal{A}$ accepts is denoted by $\mathcal{L}(\mathcal{A})$, and it is called the language of $\mathcal{A}$. $\mathcal{L}(\mathcal{A}) = \emptyset$ denotes that the language accepted by $\mathcal{A}$ is empty. Determining whether $\mathcal{L}(\mathcal{A}) = \emptyset$ is referred to as performing an emptiness check.

3.2 LTL Definitions

Linear temporal logic (LTL) [22] is commonly used for specifying properties of reactive systems. LTL is interpreted over infinite executions which makes it appropriate to specifying properties of the executions of a Kripke structure.

Given a finite non-empty set of atomic propositions AP, LTL formulas are defined inductively as follows:

1. Every member $p \in AP$ is a LTL formula.
2. If φ and ψ are LTL formulas then so are $\neg\varphi, \varphi \vee \psi, X\ \varphi$ and $\varphi\ U\ \psi$.
3. There are no other LTL formulas.

An interpretation for a LTL formula is an infinite word $\xi = x_0 x_1 x_2 \ldots$ over the alphabet 2^{AP}, i.e. a mapping $\xi : \mathbb{N} \to 2^{AP}$. The mapping is interpreted to give the propositions which are true; elements not in the set are interpreted as being false. With ξ_i we mean the suffix starting at index i, namely $x_i x_{i+1} x_{i+2} \ldots$. The semantics of LTL are given by the following:

- $\xi \models p$ if $p \in x_0$, the first index of ξ, for $p \in AP$.
- $\xi \models \neg\varphi$ if $\xi \not\models \varphi$.
- $\xi \models \varphi \vee \psi$ if $\xi \models \varphi$ or $\xi \models \psi$.
- $\xi \models X\ \varphi$ if $\xi_1 \models \varphi$.
- $\xi \models \varphi\ U\ \psi$ if there exists an $i \geq 0$ such that $\xi_i \models \psi$ and $\xi_j \models \varphi$ for all $0 \leq j < i$.

The constants $\mathbf{T} = p \vee \neg p$, for an arbitrary $p \in AP$, and $\mathbf{F} = \neg\mathbf{T}$ denote atomic propositions which are always true and respectively false. Commonly used abbreviations are $\Diamond\varphi = \mathbf{T}\ U\ \varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$ and the usual boolean abbreviations $\wedge$, $\Rightarrow$ and $\Leftrightarrow$.

By mapping the execution of a CPN or a FCPN to a sequence ξ, it is then possible to use LTL to specify properties of the executions.

Definition 9. *The corresponding sequence ξ_K of an execution $\zeta_K = M_0 M_1 \ldots$ of the Kripke structure of a CPN is the sequence generated by the function $seq(\zeta_K)$ which maps each marking in the execution to the corresponding subset of atomic propositions by evaluating the atomic propositions in the marking. The corresponding sequence ξ_F of a computation $\zeta_F = s_0 s_1 s_2 \ldots$ of the FKS of* *a FCPN is the sequence generated by the function* $seq_F(\zeta_F)$ *which maps every second state in the computation to the corresponding subset of atomic propositions by evaluating the atomic proposition in the marking component of every second state in the sequence, starting from* s_0.

Thus it is now possible to define two problems related to model checking of LTL, which are especially interesting for verification of systems modeled with Petri nets.
Model Checking Problem: Given a CPN Σ, and an LTL formula φ, does $seq(\zeta_K) \models \varphi$ hold for every execution ζ_K of Σ.
Fair Model Checking Problem: Given a FCPN Σ_F and an LTL formula ψ, does $seq_F(\zeta_F) \models \psi$ hold for every fair execution ζ_F of Σ_F.

3.3 Model Checking

The automata theoretic approach to model checking utilizes the close relationship between LTL and automata on infinite words. Several procedures [8, 4, 5, 32] have been suggested which construct a LGBA that recognizes all the models of a given LTL formula. Most model checking procedures are designed for ordinary Büchi automata but in our case this is not a problem as they are a special case of the LGBA.

Given a LTL property φ and a corresponding Büchi automaton, model checking a system is now possible by interpreting the Kripke structure as a Büchi automaton. This Büchi automaton represents all the possible executions of the system. If this system automaton is intersected with the property automaton, the result is an automaton which accepts all executions which are common to the two automata. Intersecting the system automaton with an automaton corresponding to the negation of the property yields an automaton which has no accepting executions if and only if the system is a model of the LTL property.

Hence, the steps performed to verify that a system has a property given by a LTL formula φ and solve the model checking problem are the following [3, 17]:

1. Construct a generalized Büchi automaton $\mathcal{A}_{\neg\varphi}$ corresponding to the negation of the property φ.
2. Generate the Kripke structure of the system and interpret it as a LGBA $\mathcal{K}$, with $\mathcal{F} = \emptyset$.
3. Form the product automaton $\mathcal{B} = \mathcal{A}_{\neg\varphi} \times \mathcal{K}$.
4. Check if $\mathcal{L}(\mathcal{B}) = \emptyset$.

If $\mathcal{L}(\mathcal{B}) = \emptyset$ the model of the system has the desired property. Combining several of these steps into a single algorithm and performing them in an interleaving manner is referred to as "on-the-fly" model checking [3, 17]. Naturally the procedure can also be done with a simple Büchi automaton, if the property LGBA is further expanded to a simple Büchi automaton.

The afore mentioned procedure is not appropriate for model checking a FKS, from an efficiency point of view, and thus solving the fair model checking problem. What is needed is a procedure which can handle both generalized Büchi

```
proc Check(formula φ, System K) ≡
    LGBA-Automaton A := to-automaton(¬φ);              Step 1.
    LGBA-Automaton B := product(A, K);                 Step 2.
    Streett-Automaton S;
    Component mscc;
    forall mscc ∈ MSCC(B) do                           Step 3.
        if (¬modelcheck(mscc)) then ;                  Step 4.
            continue;
        fi
        if (hasWF(mscc)) AND
            ¬wf-modelcheck(mscc)) then                 Step 5.
            continue;
        fi
        S = ToStreett(mscc);
        if (hasSF(mscc) AND
            ¬sf-modelcheck(S)) then                    Step 6.
            continue;
        fi
        counterexample(S);                             Step 7.
        return true;
    od
    return false;
.
```

Figure 2. The fair model checking procedure

acceptance sets and Streett acceptance sets. Of course, the procedure should also avoid using the more time consuming (see e.g. [24]) Streett emptiness checking procedure if possible.

To solve the fair model checking problem the new procedure for high-level Petri Nets, shown in Figure 2, does the following.

1. Constructs a generalized Büchi automaton $\mathcal{A}_{\neg\varphi}$.
2. The *Kripke* structure of the FCPN model is constructed, interpreted as a LGBA with $\mathcal{F} = \emptyset$, and simultaneously the product with $\mathcal{A}_{\neg\varphi}$ is computed.
3. Tarjan's algorithm is used to compute a *maximal strongly connected component* (MSCC) of the product. In graph theoretic terms a MSCC is a maximal subset of vertices C of a directed graph, such that for all $v_1, v_2 \in C$, the vertex v_1 is reachable from v_2 and vice versa. The set is maximal in the sense that if any state is added to this set, it ceases to be a SCC.
4. When a MSCC of the product automaton has been calculated, we check for generalized Büchi acceptance, i.e. whether there is *any* execution which violates the given property. There cannot exist a fair counterexample if there is no failing execution. Hence, if the component does not contain a state from each Büchi acceptance set (LGBA acceptance condition), we return to step 3.
5. If a component is accepted, the component is checked if it is weakly fair. This can be done without generating any intermediate states, which is of course desirable. The memberships of the fairness sets are assigned in the following

manner. Let the MSCC be denoted by C. For a state $s = \langle M, P\rangle$, where M is the corresponding marking in the Kripke structure and P the corresponding state in the formula automaton. Then, for all $s \in C$, s is member of F_i if:
- $\forall t\langle b\rangle \in en(M): \; wf_i(t)\langle b\rangle = \mathit{false}$, or
- $\exists t\langle b\rangle \in en(M), s' \in S: \; wf_i(t)\langle b\rangle = \mathit{true}$ and $(s, s') \in \Delta, M \overset{t\langle b\rangle}{\to} M', s' = \langle M', P'\rangle$, such that $s' \in C$.

See Theorem 2 why this works. If the component is accepted, i.e. it contains all weak fairness sets, and has no strong fairness constraints, the intermediate states are added according to the definition of a FKS. However, the generalized Büchi sets are interpreted as Streett acceptance sets U_i with each L_i set initialized to the universal set. The intermediate states are needed for the correctness of the counterexample. The counterexample algorithm must be able to identify fair transitions instances which can occur infinitely often. Now we can generate a counterexample at step 7.

6. We now know that the MSCC contains a weakly fair counterexample. To ensure that there is also a counterexample which is both strongly and weakly fair, we will use a Streett emptiness checking algorithm on this MSCC. (Using the Streett emptiness checking to handle strong fairness constraints goes back to at least [6, 20].) However, we cannot yet ignore the property sets and the weak fairness sets. Therefore the weak fairness sets are computed according to the definition of the FKS and both the property sets and the weak fairness sets are again simulated with Streett acceptance sets, using the technique given in step 5. Before the component is given to the Streett emptiness algorithm, also the strong fairness sets L_i and U_i must be computed. These are also computed according to the definition of a FKS. The correctness of this step is proven in Theorem 3. We simulate the FKS with a Streett automaton and if no weakly and strongly fair counterexample is found, we continue from step 3 with the next MSCC of the product automaton.
7. A counterexample is generated using the subset of vertices of the MSCC (the Streett emptiness algorithm possibly deletes some states and edges), which the emptiness checking algorithm gives to the counterexample algorithm.

Theorem 2. *Let C be a MSCC of the product automaton. The component contains a weakly fair counterexample if and only if the component interpreted as an automaton A, using the set assignments done in step 5, is non-empty.*

Proof. If C contains a counterexample which is weakly fair, then by step 3 of the procedure, the sets representing the property must be present in the automata. For all weak fairness functions there are transition instances in such a way that the implicit set generated by the fairness function is infinitely often disabled or can occur infinitely often. A state in A belongs to a weak fairness set, i.e. an acceptance set of the automaton, if no transition of the set is enabled in the state or a transition of the set occurs and the resulting state is also in the component. The first condition handles cases where the weakly fair transition instances never are enabled while the second condition handles the cases where the transition instances occur infinitely often. Thus all weak fairness sets are present if there is a weakly fair counterexample, and thus A is non-empty.

If A is non-empty, we know from step 3 of the procedure that the component contains a counterexample. Any execution respecting the acceptance sets of the property is a counterexample. As all sets are present in the component, and all states are reachable from each other, there must exist an execution which goes through both the property sets and weak fairness sets (a trivial example is an execution which visits all states of the component infinitely often). This execution is a weakly fair counterexample.

Theorem 3. *Let C be a MSCC of the product automaton. The component contains a strongly fair counterexample if and only if the Streett automaton A resulting from transforming C according to the definition of a FKS and simulating the property and weak fairness set with the Streett sets (consider the CPN marking of each product state only, ignoring the product automaton state) is non-empty.*

Proof. If C contains a counterexample which is weakly and strongly fair, then by step 3 of the procedure the property sets must be present in A, as they are simulated by some Streett sets. From Theorem 1 we know that the acceptance sets simulating the weak fairness sets will also be present as the counterexample is weakly fair. As the counterexample is also strongly fair for each strong fairness function there are transition instances, such that if the instances are infinitely often enabled, some instances occur infinitely often. In A each state which has a transition instance enabled that makes a strong fairness function true, belongs to the corresponding L set, thus marking all possible states where some strongly fair transition instances defined by a fairness function are enabled. An intermediate state is generated which will belong to the corresponding U set, which will be belong to the MSCC if the occurrence of the transition instance results in a state which is in the component. Thus the U set marks all the states making it possible for transition instances defined by a fairness function to occur. Remembering that the Streett acceptance condition is similar to the strong fairness constraint, clearly the Streett acceptance will be satisfied if a strongly fair execution is present in the component. Thus the component is non-empty.

Let A be non-empty. Any execution respecting the property sets will be a counterexample. We know from Theorem 1 that any execution respecting the FKS fairness sets must be both strongly and weakly fair. The simulation of the generalized Büchi sets (the property and the weak fairness sets) is done by setting $L_i = S$ and $U_i = F_i$. As each L_i is guaranteed to be present, each U_i must be satisfied which corresponds to the LGBA acceptance condition that each F_i must be satisfied. Since the component respects the fairness sets it is possible to construct an execution which respects the fairness sets (otherwise the component would be empty). Hence the component contains a weakly and strongly fair counterexample.

Corollary 1. *The procedure solves the fair model checking problem.*

The procedure tries to avoid the cost of the more expensive Streett emptiness check, whenever possible, by always first testing for weak fairness and only invoking the Streett check if there are strong fairness constraints enabled. This might result in faster running times compared to always performing the check. Also by performing the verification in an on-the-fly manner, checking one MSCC at a time, the cost of computing all MSCCs of the product automaton might be avoided.

There are several other algorithms for automata theoretic model checking LTL which have been presented in the literature. The nested-depth-first-search algorithm of [3] was designed for (non-generalized) Büchi automata. The algorithm of [14] is similar in the sense it uses both Büchi and Streett acceptance conditions, however their emptiness checking procedure is BDD based, making it unsuitable for the explicit state tool this procedure was designed for. The transitive closure and fixpoint computations the algorithm requires makes it an unwise choice in an explicit state setting. The same argument applies to the algorithm of [10]. An algorithm tailored to handle only generalized Büchi acceptance sets was presented in [4], but due to some optimizations it makes it could not be used here. It is, however, somewhat similar to the procedure presented in this work, as it also is a Tarjan based on-the-fly algorithm.

4 Implementation

The model checking procedure described in this work has been implemented in the MARIA analyzer [21]. The MARIA analyzer is a reachability analyzer for algebraic system nets [15, 16, 25] and it has been developed at the Laboratory for Theoretical Computer Science at Helsinki University of Technology.

4.1 The MARIA analyzer

The MARIA analyzer, is a reachability analyzer for Algebraic System Nets. The intention is to develop an analyzer with model checking capabilities for a formalism which is powerful enough to model in a straightforward manner high-level programming languages. By using language specific front ends, the idea is that MARIA can function as the analysis tool for several formalisms. Currently a front end for SDL [1] is under development.

The net class of MARIA is based on an algebra with powerful built-in data types and expressions. MARIA supports leaf types (bool, char, enum, int, unsigned) familiar from high-level programming languages and also complex structured types such as structs, arrays and bounded queues. There are built-in operations for multi-set operations, multi-set sums, etc.

4.2 Implementation

The implementation was programmed in C++, like the rest of the analyzer. The emptiness checking algorithm for Streett automata and counterexample generation follows the algorithms described in [19]. Some optimizations were however performed. Management of the arcs of the product automaton was carefully designed, so that only during the Streett emptiness checking phase were the arcs kept in main memory. The Streett emptiness check was also modified so that not all intermediate states were added to the FKS. Only transition instances related to a strong fairness constraint and some instances related to a weak fairness constraint caused an intermediate state to be added. Specifically the transition instances which belonged to a weak fairness set and could occur so that the resulting state was still in the current MSCC, caused an intermediate state to be generated. An external implementation [26] of the algorithm presented in [8] was used as the translator from an LTL formula to a LGBA.

5 Verifying a Sliding Window Protocol

The sliding window protocol provides reliable transmission over an unreliable communication medium. It is used in several data link control (DLC) protocols and it is serves as a basis for reliable transport in many protocols.

Several papers have been written on the verification of the protocol. Recent works which focus on automatic verification of the protocol include [13, 9, 31]. In [13] Kaivola combines a compositional approach with property preserving preorders and equivalences to verify both safety and liveness properties for arbitrary channel lengths. Results up to a window size of seven is presented with bounded buffer sizes. Using Queue BDDs Godefroid and Long [9] verify a full duplex version of the protocol with queue sizes up to eleven. Smith and Klarlund [31] use a theorem proving tool to verify safety properties of the protocol for unbounded buffer sizes, window sizes and channel capacities. They also verify safety properties with a fixed window size as large as 256.

5.1 Protocol Description

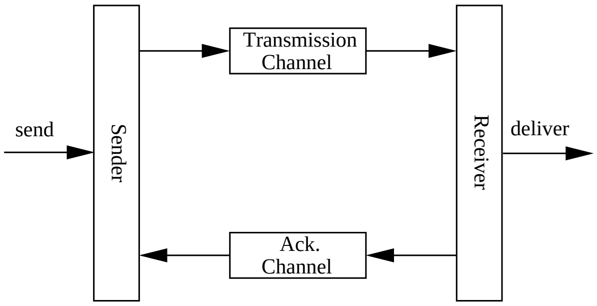

Figure 3. The main components of the protocol.

The specific version of the protocol on which we focus here is a unidirectional version due to [33]. The sliding window protocol consists of four main components. These are the sender, the receiver, the transmission channel and the acknowledgment channel. The structure of the protocol can be seen in Figure 3. Additionally we assume there is a data source which generates messages to be sent and a target which receives them. The protocol has the following important parameters:

- tw The size of the transmission window. This specifies the maximum amount of messages which the sender can send without receiving an acknowledgment.
- rw The size of the receive window. This specifies the maximum amount of messages which the receiver can receive without forwarding them to the target.
- w The maximum value of the sequence numbers.

Only when $tw + rw \leq w + 1$ holds will the protocol function properly [33].

The abstraction made in the following description follow Kaivola [13]. The sender side of the protocol functions in the following way. It receives data to be sent from the data source. It can then choose to send the message containing the data and a sequence number or receive more messages to be sent. The sender can send tw messages without receiving an acknowledgment. If no acknowledgment is received the sender can timeout and resend the messages. Following the timeout model of Kaivola, a timeout can occur only if no further messages can be sent to the transmission channel. When a valid acknowledgment is received the transmission buffer is emptied up till the last acknowledged message and the sender can receive new messages from the data source.

The receiver side of the protocol functions in the following way. When the receiver receives a message it is stored and marked as unacknowledged, if the sequence number of the message is in the reception window. Then the protocol proceeds to forward the messages to the data target and empties the receive buffer. The sequence number of the last message sent to the target is sent as an acknowledgment to the sender.

All channels in the protocol are modeled as queues with a fifo discipline, which can loose any message.

5.2 Properties

The aim of this small case study was to test the performance of the MARIA model checker. Hence, liveness properties are of special interest as they in many cases require fairness assumptions.

The liveness property required is that as many items should be delivered to the data target as are read from the data source. If the system is connected to a data source which generates the ω-regular language (see e.g. [34]) $0^\omega \cup 0^* \cdot 1$, using *data-independence* [38, 27] and properties of ω-regular languages Kaivola argues that it is then enough to verify the property:

LIVE Either infinitely many 0:s are delivered to the target or 1 is delivered.

The property can be expressed in LTL in the following way:

$$\Box\Diamond(\mathit{receive}(0)) \vee \Diamond(\mathit{receive}(1)),$$

where *receive*(i) denotes an atomic proposition indicating that the receiver has received a message with the data i. However, the property does not hold unconditionally. A fairness constraint is needed for the channels. The following fairness constraint is used:

FAIR For all sequence numbers $i \in \{0, \dots, w\}$, if a message with the sequence number i is sent to the receiver infinitely often, it receives a message with the sequence number i infinitely often. The same constraint applies for the acknowledgment channel.

5.3 The MARIA Model

Modeling the protocol in the MARIA Petri Net class is quite straightforward. Using the queue and array types of MARIA, it is easy to model the channels as queues and the internal data buffers of the sender and the receiver as arrays. The complete model consists of 12 places and 9 high-level transitions. Two places are used to make it possible to express the liveness property **LIVE** in LTL. The fairness constraints on the channels are easy to express by putting strong fairness constraints, tied to the sequence number of the message, on the channels which handle the receiving of data and acknowledgments respectively. Hence there is a fairness set for each channel for each sequence number. A weak fairness constraint is needed on the receiver side to guarantee progress in the sequential parts. To obtain the complete model see http://www.tcs.hut.fi/~timo/pn2001.

5.4 Results

The new procedure was tested by model checking the liveness property **LIVE** for different window sizes with the fairness constraint **FAIR**. The parameters were chosen such that $rw = tw$ and the capacity of the queues modeling the channels were w. To yet decrease the possible values for the parameters the value of w was fixed such that $w = tw + rw - 1$.

All tests were performed on a PC with a 1 GHz AMD Athlon processor and 512 MB of RAM. The system was running the Debian GNU/Linux distribution, version 2.2. Times were measured using the UNIX command "time" and the user plus system time was recorded. In the table of Figure 4 the results can be seen. The "states" and the "arcs" column indicate the number of nodes and the number of arcs respectively in the reachability graph. The |product| column gives the number of states in the product. The same protocol was also modeled using the Spin [11] model checker. However, because the automata translator of Spin could not translate the fairness constraints to an automaton for $w > 1$, due to memory exhaustion, a comparison was not deemed fair and hence omitted. Actually, no algorithm available to the author could translate the fairness constraints, for the window sizes presented here, on the hardware used. As can be seen from the results the MARIA model checker can due to the efficient handling of the fairness constraints cope with quite large window sizes. It should be noted that no reduction methods were used on the state space. Using reduction methods could probably enable model checking of larger window sizes.

Figure 4. Statistics for the model checking.

w	$tw = rw$	states	arcs	|product|	time (s)
1	1	118	250	297	0.1
3	2	2048	6104	5337	2.3
5	3	14952	50982	39659	36.5
7	4	79720	296088	212997	508.2
9	5	356570	1404910	955745	5978.4
11	6	1427184	5871588	3830669	59748.2

6 Conclusions

In this work LTL model checking for high-level Petri nets has been extended with nets, which have fairness constraints on the transitions. For P/T Nets a similar construction was presented, but not implemented, in [19]. The semantics for the fairness constraints is given by an execution based semantics and a fair Kripke structure. Using Streett automata the model checking procedure is extended to handle the fairness constraints in an efficient manner.

The procedure has been implemented in the MARIA analyzer. Using a model of a unidirectional sliding window protocol, the implementation was tested. Results indicate that the procedure seems to scale well, even when there are many strong fairness constraints present. The protocol could not have been verified for so large window sizes, had the fairness constraints been part of the property to be verified, instead of being part of the model.

There are still some open questions concerning the new LTL model checking procedure. It is clear that not all intermediate states have to be added when model checking. It should be possible to formulate a better sufficient condition, which could be statically checked, for transitions in the model which need the intermediate states to be generated. This could reduce the number of intermediate states needed in the procedure. It also could be interesting to generalize this method to encompass the full branching time logic CTL*. As CTL* model checking can be reduced to several calls to a LTL model checker [6] this should be possible. Another interesting question is what kind of effect would the procedure have on partial order methods, such as the stubborn set method [36].

Acknowledgments

Thanks to Keijo Heljanko many fruitful discussions and comments, to Prof. Ilkka Niemelä for suggesting the sliding window protocol example and to the anonymous referee whose comments helped me find a bug in the original Maria model of the protocol.

References

1. CCITT. Specification and description language (SDL). Technical Report Z.100, ITU-T, 1996.
2. E.M. Clarke and E.A. Emerson. Design and synthesis of syncronization of skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, pages 52–71. Springer-Verlag, 1981. LNCS 131.
3. 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.
4. J-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceeding of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1, pages 253–271, Berlin, 1999. Springer-Verlag. LNCS 1708.
5. M. Daniele, F. Giunchiglia, and M.Y Vardi. Improved automata generation for linear temporal logic. In *Proceedings of the International Conference on Computer Aided Verification (CAV'99)*, pages 249–260, Berlin, 1999. Springer-Verlag. LNCS 1633.
6. E.A. Emerson and C-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.
7. N. Francez. *Fairness*. Springer-Verlag, New York, 1986.
8. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.
9. P. Godefroid and D.E Long. Symbolic protocol verification with Queue BDDs. *Formal Methods in System Design*, 14(13):257–271, May 1999.
10. R. Hojati, V. Singhal, and R.K. Brayton. Edge-Streett / edge-Rabin automata environment for formal verification using language containment. Memorandum UCB/ERL M94/12, Electronics Research Laboratory, University of California, Cory Hall, Berkley, 1994.
11. G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
12. K. Jensen. *Coloured Petri Nets*, volume 1. Springer-Verlag, Berlin, 1997.
13. R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 48–59. Springer-Verlag, 1997. LNCS 1254.
14. Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal properties. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP 1998)*, pages 1–16. Springer-Verlag, 1998. LNCS 1443.
15. E. Kindler and W. Reisig. Algebraic system nets for modeling distributed algorithms. *Petri Net Newsletter*, 51:16–31, 1996.
16. E. Kindler and H. Völzer. Flexibility in algebraic nets. In *Proceedings of the International Coneference on Application and Theory of Petri Nets 1998 (ICAPTN'98)*, pages 345–364. Springer-Verlag, 1998. LNCS 1420.
17. R.P. Kurshan. *Computer-Aided Verfication of Coordinating Processes: The Automata-Theoretic Approach*. Princeon University Press, Princeton, New Jersey, 1994.
18. T. Latvala. Model checking linear temporal logic properties of Petri nets with fairness constraints. Research Report A67, Helsinki University of Technology, Jan 2001.

19. T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.
20. O. Lichtenstein and A. Pnueli. Checking that finite state programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.
21. M. Mäkelä. Maria: Modular reachability analyzer for algebraic system nets. On-line documentation, 1999. <http://www.tcs.hut.fi/maria>.
22. A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.
23. J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1981.
24. M. Rauch Henzinger and J.A. Telle. Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory(SWAT'96)*, 1997.
25. W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, March 1991.
26. M. Rönkkö. A distributed object oriented implementation of an algorithm converting a LTL formula to a generalised Büchi automaton. On-line documentation, 1998. <http://www.abo.fi/%7Emronkko/PROJECT/LTL2BUCHI/abstract.html>.
27. K. Sabnani. An algorithmic technique for protocol verification. *IEEE Transactions on Communications*, 36(8):235–248, 1988.
28. S. Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science*, pages 319–327. IEEE, 1988.
29. K. Schmidt. LoLA: A low level analyser. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000)*, pages 465–474. Springer-Verlag, 2000. LNCS 1825.
30. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.
31. M.A. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In *Proceeding of the 20th Joint International Conference on Formal Methods for Distributed Systems and Communication Protocols (FORTE/PSTV 2000)*, pages 19–34. Kluwer Academic Publishers, 2000.
32. F. Somenzio and R. Bloem. Efficient büchi automata from LTL formulae. In *Proceedings of the International Conference on Computer Aided Verification (CAV2000)*, pages 248–263. Springer-Verlag, 2000. LNCS 1855.
33. N.V Stenning. A data transfer protocol. In *Computer Networks*, volume 11, pages 99–110, 1976.
34. W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier, 1990.
35. W. Thomas. Languages, automata and logic. In G Rozenberg and A Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 385–455. Springer-Verlag, New York, 1997.
36. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, pages 429–528. Springer-Verlag, 1998. LNCS 1491.
37. K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD reference manual. Technical Report 13, Helsinki University of Technology, 1995.
38. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–194, 1986.