TEKNILLINEN KORKEAKOULU
Sähkö- ja tietoliikennetekniikan osasto

Timo Latvala

Model checking linear temporal logic properties of Petri nets with fairness constraints

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, 9.10.2000

Supervisor: Professor Nisse Husberg

Instructor: Lic.Sc.(Tech.) Keijo Heljanko

TEKNISKA HÖGSKOLAN **SAMMANDRAG AV DIPLOMARBETET**

Utfört av: Timo Latvala

Arbetets namn: Model Checking Linear Temporal Logic Properties of Petri Nets with Fairness Constraints
Namn på svenska: Modellverifiering för Petrinät med rättviserestriktioner av egenskaper givna i linjär temporallogik

Datum: 9. October 2000	**Sid antal:** 45
Avdelning: Elektro- och telekommunikationsteknik	
Professur: Tik-79 Databehandlingsteori	
Övervakare: Prof. Nisse Husberg	**Handledare:** TkL. Keijo Heljanko

Sammandrag: Verifiering av livskraftsegenskaper kräver i många fall att man gör rättviseantagande om systemet som skall verifieras. Inom Petrinät teorin har rättviseantaganden utforskats men man har inte visat hur man skall kombinera modellverifiering och rättviseantaganden effektivt.

I det här arbetet utvidgas färgade Petrinät med rättviseantaganden anslutna till transitionerna. Semantiken för rättviseantaganden definieras med hjälp av en rättvis Kripkestruktur. En ny modellverfieringsprocedur för linjär temporallogik (LTL) presenteras som möjliggör verifiering av Petrinät med rättviseantaganden. Proceduren använder Streett-automater för att effektivt handskas med rättviseantaganden. Nya algoritmer för genering av motexempel och för Streett-automaternas tomhetsproblem har också utvecklats i arbetet.

Den nya modellverifieringsproceduren har implementerats i MARIA-analysatorn. Några experiment har utförts för att prova implementationen och jämföra den med andra sätt av att handskas med modellverfiering för Petrinät med rättviseantaganden. Resultaten visar att den nya proceduren klarar sig bra jämfört med andra metoder.

Nyckelord: Datorstödd verifiering, modellverifiering, Petrinät, linjär temporallogik, rättvisa, Streett-automater, motexempel.

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF THE MASTERS'S THESIS**

Author: Timo Latvala

Name of the Thesis: Model Checking Linear Temporal Logic Properties of Petri Nets with Fairness Constraints

Name in Swedish: Modellverifiering för Petrinät med rättvise restriktioner av egenskaper givna i linjär temporallogik.

Date: October 9, 2000	**Pages:** 45
Department: Electrical and Communications Engineering	
Professorship: Tik-79 Theoretical Computer Science	
Supervisor: Prof. Nisse Husberg	**Instructor:** Lic.Sc.(Tech.) Keijo Heljanko

Abstract: Verification of liveness properties of systems requires in many cases fairness constraints to be imposed on the system. In the context of modeling and analysis with Petri nets, fairness constraints have been defined but the results have not been extended to model checking.

In this work Coloured Petri nets are extended with fairness constraints on the transitions. The semantics of the fairness constraints are defined with a fair Kripke structure. Model checking linear temporal logic (LTL) properties of the Petri net is facilitated by introducing a new LTL model checking procedure. The procedure employs Streett automata to cope with the fairness constraints efficiently. Also, new algorithms for the emptiness checking problem of Streett automata and counterexample generation are presented.

The new procedure has been implemented in the MARIA analyzer. Some experiments are performed to test the implementation and compare it with other ways of coping with fairness constraints. The results show that the procedure scales well when compared to alternative approaches.

Keywords: Computer aided verification, Petri nets, model checking, fairness, Streett automata, counterexamples.

Foreword

This work could not have been done if it were not for the help and support of several people. I thank my supervisor Prof. Nisse Husberg for letting me work on topics which interest me. My instructor Keijo Heljanko has been very patient with me during the writing of this work. His comments have been very important. I am also grateful for the useful comments by Tommi Junttila. Marko Mäkelä has provided valuable input for the implementation of the model checker.

The financial support of TEKES, Nokia Research Center, Nokia Mobile Phones, Elisa Communications, and the Finnish rail administration is gratefully acknowledged.

Finally, I would like to thank my beloved Emma for making the moments when I am not working enjoyable.

Timo Latvala
Espoo, Finland, October 9, 2000

Contents

List of Symbols and Abbreviations

$\wedge$	logical and
$\vee$	logical or
$\neg$	logical negation
$\Leftrightarrow$	logical equivalence
$\Rightarrow$	logical implication
$\exists$	existential quantifier
$\exists^{\omega}$	exists infinitely many
$\forall$	universal quantifier
$\emptyset$	empty set
$\cap$	set intersection
$\cup$	set union
$\setminus$	set difference
$\subseteq$	subset
$\subset$	proper subset
$\|X\|$	set cardinality
2^X	power set
$\times$	Cartesian product
$O(f(x))$	is at most of the order of f
$\mathbb{N}$	the set of natural numbers (including zero)
$\mathbb{Z}$	the set of integers
CPN	Coloured Petri Net
FCPN	Fair Coloured Petri Net
FKS	Fair Kripke Structure
LGBA	Labeled Generalized Büchi Automaton
MSCC	Maximal Strongly Connected Component
NP	nondeterministic polynomial time
PSPACE	polynomial space
LTL	linear time temporal logic
CTL	computation tree logic
CTL*	full branching time logic CTL*
BDD	binary decision diagram
MB	megabyte
MHz	megahertz
SDL	System Description Language

1 Introduction

The field of computer aided verification is currently evolving rapidly. The complexity and the distributed nature of many modern systems which provide critical services have motivated this research, because parallel and distributed systems can contain subtle errors which can be very hard to find. Although the *state space explosion problem* (see e.g. [34]) severely restricts the applicability of computer aided verification there are currently several viable methods which have been successfully applied to large systems (see e.g. [3]). Most of the methods are based on *model checking*.

Model checking [4, 24] refers to a collection of techniques which all have in common that a system is verified by checking that a representation of the state space of the system is a model of a logical formula. This exhaustive enumeration (explicit or implicit) of the reachable states of the system covers all behaviors, in contrast to traditional validation techniques such as testing. However, the perhaps two most important features of model checking compared to other approaches of formal verification are that it is to a large extent an automated procedure and it is able to produce an error trace if the system does not conform to the given specification. In this work we will consider model checking within the automata theoretic framework [35, 36] using an explicit representation of the state space.

Model checking requires a formal model of the system from which the state space of the system can be computed. One commonly used class of formalisms are high-level Petri nets. As the greatest advantage of Petri nets is often mentioned that they combine a graphical notation with a well-defined semantics allowing formal analysis [13]. Hence, modeling the behavior of different types of systems is in many cases convenient using high-level Petri nets. Static properties can be checked directly from the model, while most dynamic properties such as safety and liveness properties require that the behavior of the net is analyzed. Although both safety properties and liveness properties of the system can be verified using model checking algorithms which analyze the reachable state space of the net, the proving of liveness properties is a bit more involved than the proving of safety properties. Mostly this is due to that safety properties do not require any additional assumptions to be made about the behavior of the Petri net model. The same does not, however, always apply to liveness properties. In many cases certain unwanted behaviors must be ignored in order to facilitate the model checking of liveness properties.

One common technique uses *fairness assumptions* [9] to restrict the behavior of the model, so that certain liveness properties will hold. Without these assumptions it is difficult to build a model which has the desired properties. However, no model checking procedure can currently perform the model checking directly on a Petri net model with fairness assumptions. Currently the only way is to model check properties of the form "*fairness* $\Rightarrow$ *property*" and potentially modify the model so that the fairness assumptions can be expressed. The question is how practical this is from a modeling perspective, and is it computationally efficient. These issues will be covered in detail in Section 2.

Motivated by the facts given above, the main contributions of this work are the following. We present how the LTL model checking problem for high-level Petri nets, with fairness constraints imposed on transitions, can be solved by employing Streett automata in a straightforward manner. We give an on-the-fly LTL model checking procedure which uses the emptiness checking for generalized Büchi automata to potentially avoid some of the more costly Streett automata emptiness checks. Also, simple and memory efficient algorithms for Streett automata emptiness checking and counterexample generation are developed. The algorithms have been implemented and tested in the MARIA tool [22]. Most of these results were first presented in the papers [19, 20].

The rest of this work is structured as follows. In Section 2 we introduce Petri nets, define Coloured Petri nets and discuss some of the modeling issues. Section 3 covers the necessary automata theory. Linear time temporal logic (LTL) and the automata theoretic approach to model checking LTL are described in Section 4. In Section 4 an extension of the normal model checking procedure, which respects the fairness constraints is also introduced. In Section 5 implementation issues are discussed and also the results of some experiments are presented. Conclusions and directions for further work are discussed in Section 6.

2 Petri Nets

Petri nets (see for e.g. [26]) are a widely used modeling formalism for concurrent and distributed systems. A Petri net has an explicit representation of both states and actions, which makes Petri nets a versatile modeling formalism appropriate for many different tasks. There are several kinds of Petri nets ranging from simple Place/Transition nets to different high-level nets. Here the basic notions of Petri nets are introduced and the well-known Coloured Petri nets (CPN) [13] are defined.

2.1 Preliminaries

Multi-sets are very similar to ordinary sets, but true to their name, the cardinality of an element in a multi-set can be any natural number.

Definition 1 *A* ***multi-set*** *m, over a non-empty set S, is a function $m : S \mapsto \mathbb{N}$, where $m(s)$, $s \in S$, is the number of appearances of s in the multi-set m. An element $s \in S$ is said to* **belong** *to the multi-set m iff $m(s) \neq 0$. The formal sum $\sum_{s\in S} m(s)\langle s\rangle$ is used to represent a multi-set m. By $\mathcal{MS}(S)$ we denote the set of all multi-sets over S.*

A simple multi-set m_1 over the set $\{a, b, c\}$ with four elements a, a, b, c is denoted like this: $m_1 = 2\langle a\rangle + \langle b\rangle + \langle c\rangle$.

One can define several operations over multi-sets. For this we first define the monus operator.

Definition 2 *Let $z_1, z_2 \in \mathbb{N}$. The* ***monus*** *of z_1 and z_2, denoted $z_1 \dot{-} z_2$ is defined in the following way.*

$$z_1 \dot{-} z_2 = \begin{cases} z_1 - z_2, & \text{if } z_1 \geq z_2 \\ 0, & \text{otherwise} \end{cases}$$

The normal arithmetic operations $+$ and $-$ can also be extended to multi-sets.

Definition 3 ***Addition, comparison, subtraction*** *for multi-sets and the* ***size*** *of a multi-set are defined in the following way. Let m_1, m_2 and m be multi-sets over S.*

1. $m_1 + m_2 = \sum_{s\in S}(m_1(s) + m_2(s))\langle s\rangle$.
2. $m_1 \neq m_2$ *iff* $\exists s \in S : m_1(s) \neq m_2(s)$
   $m_1 \leq m_2$ *iff* $\forall s \in S : m_1(s) \leq m_2(s)$.

3. $|m| = \sum_{s \in S} m(s)$

4. $m_2 - m_1 = \sum_{s \in S} (m_2(s) \dot{-} m_1(s))\langle s \rangle$.

For instance, if we define $m_2 = \langle a \rangle + \langle b \rangle + 0\langle c \rangle$ we have that $m_1 + m_2 = 3\langle a \rangle + 2\langle b \rangle + \langle c \rangle$. Both associativity and commutativity hold for multi-sets with the operation "+".

For the definition of a Coloured Petri Net a semantics and a syntax for expressions is needed. A concrete syntax for the expressions will not be defined in this work. We will, however, assume that such a syntax exists and has a well defined semantics so that the following notations are defined:

- T - The elements of a *type* T.
- *Type*(v) - The type of the variable v.
- *expr* - A legal expression.
- *Type*(*expr*) - The type of an expression *expr*.
- *EXPR* - The set of all legal expressions.
- *Var*(*expr*) - The *set of variables* in an expression *expr*.
- A *binding* b which associates each variable $v \in V$ with a corresponding value $b(v) \in$ *Type*(v).
- *Var*(b) - The set of variables of a binding b.
- *expr*$\langle b \rangle$ - The *value* obtained by evaluating an expression *expr* with a binding b. We require that $Var(expr) \subseteq Var(b)$. The evaluation is performed by substituting each variable by the value given by b.

2.2 Petri Net Definitions

We are now going to define a Coloured Petri Net (CPN). The definition follows quite faithfully the definition of Coloured Petri Nets in [13]. 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.

Definition 4 *A tuple* $\Sigma = \langle \Pi, P, T, A, N, C, E, G, M_0 \rangle$ *is a* ***Coloured Petri Net*** *(CPN) [13] where,*

i.) Π *is a finite set of non-empty types called* ***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.) $N : A \mapsto (P \times T) \cup (T \times P)$ *is a* ***node function****. A node is either a place or a transition of the net.*

vi.) $C : P \mapsto \Pi$ *is a* ***colour function****.*

vii.) $E : A \mapsto EXPR$ *is an* ***arc expression function*** *such that* $\forall a \in A : E(a)\langle b\rangle \in \mathcal{MS}(C(p(a)))$ *holds for all legal bindings* b*, where* $p(a)$ *is the place component of* $N(a)$*.*

viii.) $G : T \mapsto EXPR$ *is a* ***guard function*** *such that* $G(t)\langle b\rangle \in \{\mathit{true}, \mathit{false}\}$ *for any legal binding* b *and* $t \in T$ *(see Definition 6 below).*

ix.) M_0 *is an* ***initialization function*** *(initial marking), which maps each place to a closed expression, i.e an expression without variables which can be evaluated immediately, of type* $\mathcal{MS}(C(p))$*, i.e. a multi-set over* $C(p)$*.*

The set of colour sets (i) determines the types and the expressions which can be used in the arc expressions, guard functions and initialization functions. The static structure of the net is described by the set of places, transitions, arcs and the node function (ii, iii, iv, and v). The node function maps each arc into a pair, where the first element is the source node and the second element is the destination node. Each place has a colour set attached to it by the colour function (vi), which determines its type. The arc expression function (vii) describes the tokens which move between the nodes of the net and must yield a multi-set which is of the same type as the place the expression is connected to. With the the guard expression (viii) it is possible to restrict the arc expression function further with a boolean expression. The initialization function (ix) maps each place to a closed expression which must be a multi-set over $C(p)$ (see also the definition of a marking below).

With the static structure of the net defined, the emphasis can now move to the behavior of the net. The following notation must, however, first be defined.

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

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

Each place in the net can be occupied by tokens. A distribution of tokens on the places of the net is called a marking. A marking of the net describes the current global state of the system being modeled.

Definition 5 *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.

Transitions are responsible for changing the marking in a CPN. The possible instances of a high-level transition are determined by the legal bindings of the transition. For a legal binding all variables must be bound with a value of the correct type and the guard of the transition must be true.

Definition 6 *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 legal binding of transition cannot always occur. We require the notion of enabledness, i.e. when a transition instance can occur and define the results of an occurrence.

Definition 7 *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 $M \xrightarrow{t\langle b\rangle} M'$.

The behavior of a CPN can be described by a Kripke structure.

Definition 8 *A triple* $K = \langle S, \rho, s_0\rangle$*, where* S *is a set of markings,* ρ *is a transition relation, and* s_0 *an initial state, is the* ***Kripke structure*** *of a CPN* $\Sigma = \langle \Pi, P, T, A, N, C, E, G, M_0\rangle$ *where* S *and* ρ *are defined inductively as follows:*

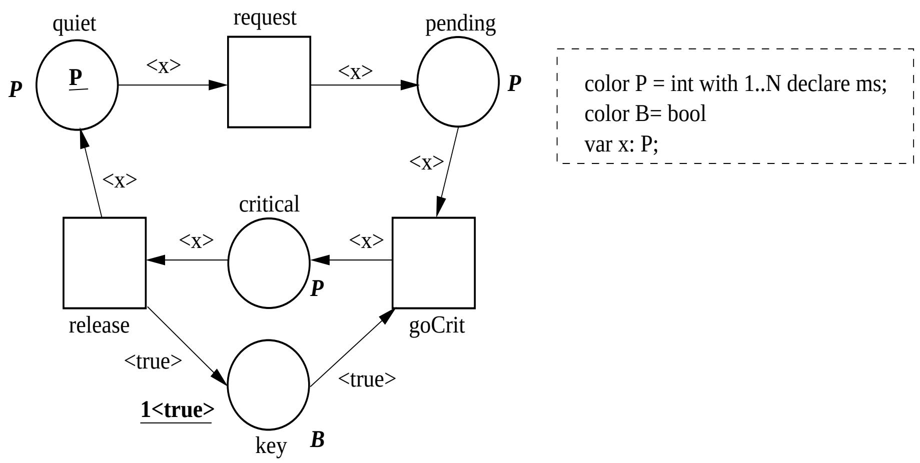

Figure 1: A simple mutex algorithm for N processes.

1. $s_0 = M_0 \in S$
2. *If* $M \in S$ *and* $M \xrightarrow{t\langle b\rangle} M'$*, then* $M' \in S$ *and* $\langle M, M'\rangle \in \rho$.
3. S *and* ρ *have no other elements.*

The executions of the net are infinite sequences of states $M_0 M_1 M_2 \ldots \in S^\omega$*, where* M_0 *is the initial state and* $(M_i, M_{i+1}) \in \rho$ *for all* $i \geq 0$.

A well-known class of distributed algorithms are the mutual exclusion algorithms. The basic problem they solve is how a critical resource should be shared between N processes in such a way that only one process can enter the critical section at a time. There are several solutions for this problem, and one of the simplest is known as the contentious mutex algorithm (see e.g. [28]). The algorithm has been modeled as an CPN for the case of N processes in Figure 1 (see Appendix A for the MARIA net description). In the figure, the letter written in bold italic beside the places describe the colour set of the place. Underline is used for denoting the initial marking. If a color declaration has "declare ms" in its definition, the type name can be used to initialize a place with a multi-set containing each element of the multi-set once. This has been used in the place "quiet". The model behaves in the following way. Any process may spontaneously request access to the critical section. Access is granted by giving the process the *key* to the critical section. As there is only one key, mutual exclusion is assured. The process returns the key when it exits the critical section. In Figure 2 the Kripke structure for $N = 2$ has been generated. Using a model checker for Petri nets, it could easily be verified that the mutual exclusion property truly holds.

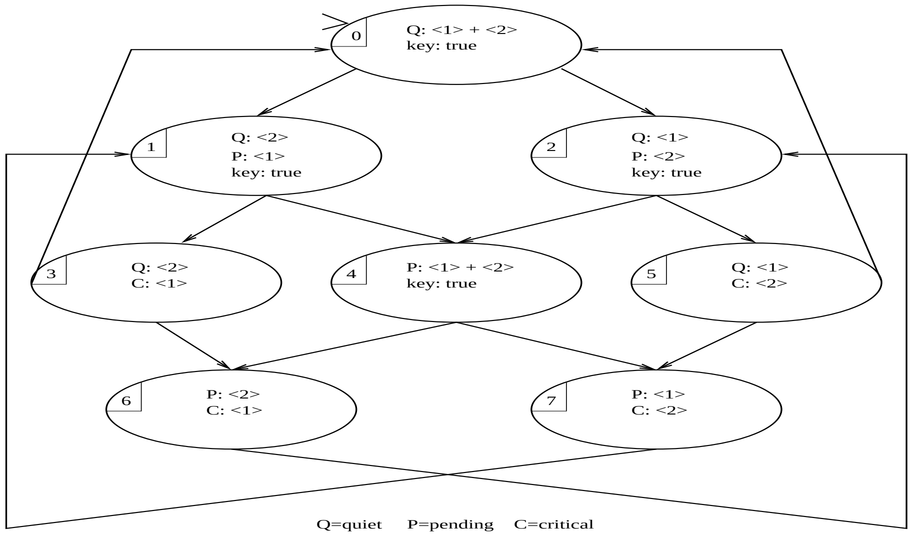

Figure 2: The Kripke structure of the net in Figure 1 for $N = 2$.

2.3 Petri Nets and Fairness

Applying a model checker to the Kripke structure in Figure 2 and checking that for all processes that if they try to get access to the critical section they eventually will, the model checker will report that the property does not hold. A sequence where always one process gains access to the critical section and thus denying the others access is possible. If the counterexamples are studied, one will notice that they all represent executions where one or several transitions are never fired, even though they are infinitely often enabled. This sort of *unfair* behavior, although theoretically possible, is not what one would expect from a physical system. Usually they contain a scheduler which disallows unfair behavior or it is considered impossible for some other reason. Clearly, some kind of mechanism is required for disqualifying these unfair executions and not accepting them as legal counterexamples.

Using *fairness assumptions* for the transitions is the perhaps the most convenient way one can restrict the set of legal executions to the desired ones. The most common fairness assumptions are known as *weak fairness* and *strong fairness*. In [9] 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. Weak fairness is usually appropriate for systems with busy waiting to ensure progress. In a weakly fair scheduler, once a process has been scheduled for execution, it will eventually be executed. For some situations, however, weak fairness is not enough. A situation where a process infinitely often requests access to a critical section, but never gains it, is weakly fair since the process is not

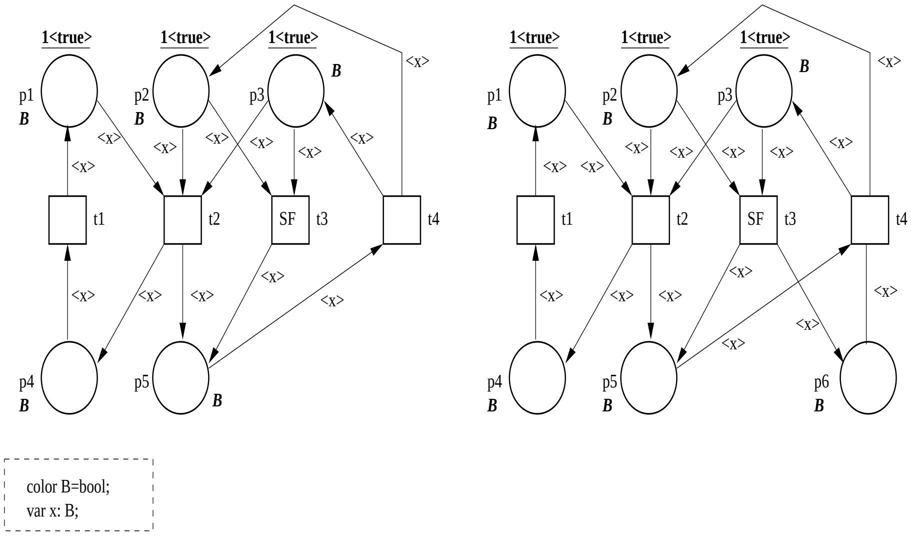

Figure 3: An LTL formula cannot capture fairness in the left net.

continuously enabled. In this situation *strong fairness* is appropriate. Strong fairness assumes that if an event is infinitely often enabled then it will occur infinitely often. An important thing to notice is that both weak fairness and strong fairness are expressible in LTL (see Section 4 for details).

In [8] 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 [11] while in [14] a procedure for model checking LTL properties of BDDs with strong fairness constraints was presented. However, no procedure existed prior to this work which directly could perform LTL model checking of a Petri net model with fairness constraints. 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 model. These modifications in a sense model a scheduler and have to be done because 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 increases the size of the state space, and the size of the Büchi automaton representing the property can grow exponentially in the number of fairness constraints (see e.g. [10]). A more subtle drawback is that adding the scheduler reduces the concurrency in the model, which may affect the performance of some partial order methods (see e.g. [34]).

Consider the net on the left side of Figure 3. It is an example of a net which must be modified so that the transition t_3 can be given strong fairness constraints with an LTL formula. The reason for this is that it is impossible to distinguish if t_2 or t_3 has occurred just by observing the markings. By adding an additional place p_6 to the net as shown on the right net in Figure 3, the occurrence of t_3 can be detected by checking if the place p_6 is marked. With this modification it is now possible to express a strong fairness constraint in LTL for t_3

Modifying the net is not the only way to give transitions fairness constraints. The solution suggested in this work is to extend Petri nets with fairness constraints on the transitions. For this to be useful we must also change the semantics of a legal execution and modify the LTL model checking procedure accordingly. We begin by extending the definition of a CPN.

Definition 9 *A **fair CPN** (FCPN) is tuple $\Sigma_F = \langle \Sigma, WF, SF \rangle$ where Σ is a CPN and $WF = \{wf_1, \dots wf_k\}$ a set of weak fairness functions, where $wf_i : T \mapsto EXPR$ 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. $SF = \{sf_1, \dots, sf_m\}$ is the corresponding set of strong fairness functions with similar restrictions. An execution is an infinite sequence of transitions and markings $\xi = M_0 \stackrel{t_0\langle b_0 \rangle}{\rightarrow} M_1 \stackrel{t_1\langle b_1 \rangle}{\rightarrow} \dots$, for which $M_i \stackrel{t_i\langle b \rangle}{\rightarrow} M_{i+1}$ for some $t_i\langle b \rangle$ and ξ obeys the fairness constraints defined below.*

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

We need to define some notation before the semantics of the fairness constraints can be defined.

Let $\xi = M_0 \stackrel{t_0\langle b_0 \rangle}{\rightarrow} M_1 \stackrel{t_1\langle b_1 \rangle}{\rightarrow} \dots$ be an execution and F a fairness function. We define $EN_{F,i}(\xi) = true$ if $\exists t\langle b \rangle \in en(M_i) : F(t)\langle b \rangle = true$; otherwise $EN_{F,i}(\xi) = false$. Also let $OC_{F,i}(\xi) = true$ if $F(t_i)\langle b_i \rangle = true$; otherwise $OC_{F,i}(\xi) = false$. Denote the quantifier "there exist infinitely many" by $\exists^\omega$ and let $InfEN_F(\xi)$ and $InfOC_F(\xi)$ be defined in the following way:

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

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

The strong and weak fairness constraints for transitions can now be defined conveniently using the previously defined notation. An execution ξ of a FCPN is legal when it respects the strong fairness constraint, i.e. if a set of transitions instances, defined by a fairness function F, are infinitely often enabled implies that they occur infinitely often [13].

$$\forall F \in SF : InfEN_F(\xi) \Rightarrow InfOC_F(\xi).$$

A legal execution ξ must also obey the weak fairness constraints. The definition for weak fairness is that persistent enabling implies an occurrence [13].

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

The semantics of our fairness constraints are equivalent to those presented by Jensen in [13], but the notation is a little different. However, for analysis purposes these execution based semantics do not suffice. Something similar to a Kripke structure is needed. In the following discussion we only consider finite state and finitely branching systems.

The behavior of a fair CPN 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 by extending the definition of a Kripke structure to fair Kripke structure.

Definition 10 *A* ***fair Kripke structure (FKS)*** *[14] is a tuple $K_F = \langle S, \rho, s_0, \mathcal{W}, \mathcal{S} \rangle$, 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. 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 (to be defined below). 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 $L_i, U_i \subseteq S$.*

We define for notational convenience the set

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

$Inf(\sigma)$ is the set of states occurring infinitely often in the execution σ. An execution σ is a computation if both the weak and strong fairness requirements are satisfied. $\bigwedge_{i=1}^{k} Inf(\sigma) \cap J_i \neq \emptyset$ is demanded by the weak fairness requirement. The strong fairness requirement demands that $\bigwedge_{i=1}^{m}(Inf(\sigma) \cap L_i = \emptyset \vee Inf(\sigma) \cap U_i \neq \emptyset)$.

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

Using the acceptance conditions, it is possible to only accept the computations which adhere to the fairness constraints on the transitions. 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[2]. 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_i the sequence will be $M_j M_i M_{j+1}$. With the intermediate states added it is now possible to use the justice and compassion 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 fair CPN system $\Sigma_F = \langle \Sigma, \mathit{WF}, \mathit{SF}\rangle$, we define S and ρ inductively as follows:

1. $s_0 = \langle M_0, \perp\rangle \in S$.
2. If $\langle M, \perp\rangle \in S$ and $M \overset{t\langle b\rangle}{\to} M'$ then, $\langle M', t\langle b\rangle\rangle \in S, \langle M', \perp\rangle \in S$ and $(\langle M, \perp\rangle, \langle M', t\langle b\rangle\rangle) \in \rho$, $(\langle M', t\langle b\rangle\rangle, \langle M', \perp\rangle) \in \rho$.
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
   - $J_i = \{\langle M, \perp\rangle \in S \mid \forall t\langle b\rangle \in en(M) : wf_i(t)\langle b\rangle = \mathit{false}\} \cup \{\langle M', t\langle b\rangle\rangle \in S \mid wf_i(t)\langle b\rangle = \mathit{true}\}$.
2. For each $sf_i \in SF$ the strong fairness sets are
   - $L_i = \{\langle M, \perp\rangle \in S \mid \exists t\langle b\rangle : t\langle b\rangle \in en(M) \wedge sf_i(t)\langle b\rangle = \mathit{true}\}$ and
   - $U_i = \{\langle M', t\langle b\rangle\rangle \in S \mid sf_i(t)\langle b\rangle = \mathit{true}\}$.

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.

[2]In an actual model checker implementation some of the intermediate states would not have to be added. See Section 5 for details.

Theorem 11 *Given a FCPN Σ_F, Σ_F has a fair execution $\xi = M_0 \stackrel{t_0\langle b_0\rangle}{\rightarrow} M_1 \stackrel{t_1\langle b_1\rangle}{\rightarrow} \ldots$, 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:
Let ξ' be the execution of the FKS, where each marking of ξ is mapped to the corresponding marking in the FKS and the transition instances to the corresponding intermediate state. This simple mapping is bijective between ξ and ξ'. It remains to be proven that ξ' is a computation. Let $L_i(\xi') = L_i \cap Inf(\xi')$, and let $U_i(\xi')$ and $J_i(\xi')$ be defined in the same way for U_i and J_i respectively.

Let us first consider the strong fairness case. If $\neg InfEN_{sf_i}(\xi)$ then $L_i(\xi') = \emptyset$, because according to the definition of a FKS, L_i includes all the states where a transition instance of the strong fairness set i is enabled. When $L_i(\xi) = \emptyset$ all executions is are accepted, as should be the case. If $InfEN_{sf_i}(\xi)$ then $L_i(\xi) \neq \emptyset$ as it will include all states where a transition of the set is enabled, and because Σ_F is finite state, some state in L_i must occur infinitely often. According to the proposition, now $InfOC_{sf_i}(\xi')$ must also hold. $U_i(\xi')$ contains all the states which appear infinitely often in ξ', where a fair transition instance has just occurred. As $InfOC_{sf_i}(\xi)$ holds it follows that $U_i(\xi') \neq \emptyset$ holds, due to the finiteness of Σ_F. Hence ξ' is a computation of the FKS with respect to the L_i and U_i sets and respects strong fairness.

Let us now consider the weak fairness case. In ξ each weakly fair transition instance class either is infinitely often not enabled or occurs infinitely often. $J_i(\xi')$ is the set of states, which appear infinitely often in ξ', where either no fair transition instance of wf_i is enabled or some instance has just occurred. If a weak fairness set i is infinitely often not enabled in ξ, the set $J_i(\xi')$ will be non-empty and the execution is accepted as a computation, as it should be. If a weak fairness set i is infinitely often enabled in ξ, the set J_i will consist of all the markings where a transition of the set i is not enabled or it has just occurred. From the definition of weak fairness for a FCPN, projecting ξ on ξ' we get for each state s_j in the execution ξ' there is a state s_k, where $k \geq i$ such that $\neg EN_{wf_i,k}(\xi)$ or $OC_{wf_i,k}(\xi')$. Thus states of $J_i(\xi')$ must occur infinitely often in the execution due to the finiteness of Σ_F. Hence ξ' is computation of the FKS also with respect to the weak fairness sets.

Consider now a computation ξ' of the FKS. The corresponding execution ξ of the FCPN can be obtained by using the same mapping as above. We must now prove that ξ is a legal computation of Σ_F. From the definition of a computation we know that in a computation ξ', if state which is member of a set L_i appears infinitely often, then also a state which is member of U_i appears infinitely often in ξ'. This implies for ξ that if a transition instance class is enabled infinitely often, it also occurs infinitely often. In

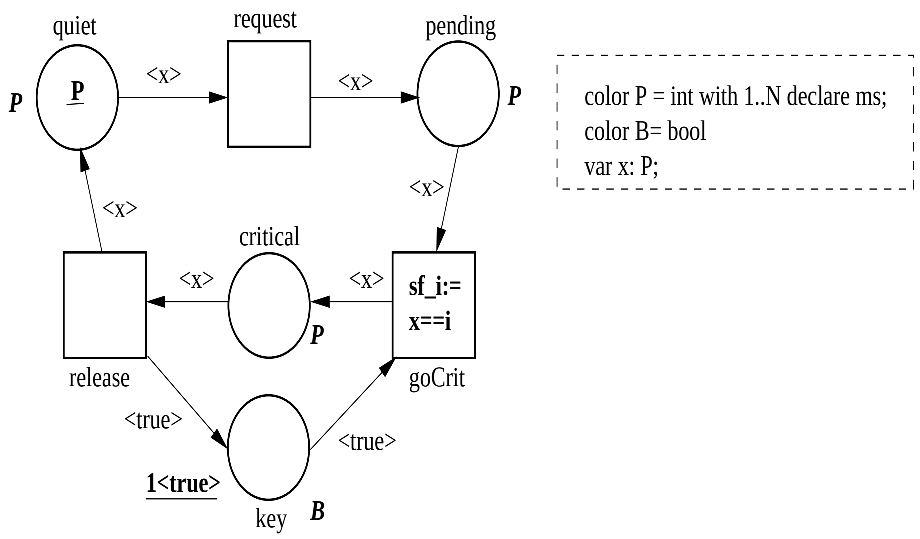

Figure 4: The fair mutex algorithm.

the previously defined notation this is expressed

$$\forall F \in SF : InfEN_F(\xi) \Rightarrow InfOC_F(\xi).$$

Thus ξ also respects the strong fairness constraint.

In each computation states of $J_i(\xi')$ must occur infinitely often, because of the definition of a computation of a FKS. This implies that in the future of each state in the execution ξ there is either a state where the corresponding transition to fairness set has occurred or it is not enabled. Using the notation defined previously it holds that

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

Thus ξ respects the weak fairness constraint. $\square$

As the execution based semantics defined previously are equivalent to Jensen's semantics for fairness [13], the theorem also shows that the FKS construction adheres to them.

If we again consider the contentious mutex example, the fairness constraints can now be easily added. A fair model can be found in Figure 4. The *goCrit* transitions must be equipped with strong fairness constraints, so that access to the shared variable can be guaranteed for all processes. Weak fairness is not enough, because if one process gains access to the variable, the transition is disabled for other since the *key* place is empty. The *request* transition should not be given any fairness constraints, since it should be possible for a process not to request access to the shared variable indefinitely. Now a process will always gain access if it tries to.

3 Automata on Infinite Words

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 12 *A* ***labeled generalized Büchi automaton*** *(LGBA) [5] 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 will be possible to extend the LTL model checking procedure to also cope with strong fairness in an efficient manner.

Definition 13 *A* ***Streett automaton*** *(see [33] 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. [30]). 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.

The synchronous product, denoted $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$, between two LGBAs $\mathcal{A}_1 = \langle Q_1, \Delta_1, I_1, \mathcal{F}_1, \mathcal{D}, \mathcal{P}_1 \rangle$ and $\mathcal{A}_2 = \langle Q_2, \Delta_2, I_2, \mathcal{F}_2, \mathcal{D}, \mathcal{P}_2 \rangle$ is defined as:

- $Q = \{(q_1, q_2) \in Q_1 \times Q_2 \mid \mathcal{P}(q_1) \cap \mathcal{P}(q_2) \neq \emptyset\}$,
- $\Delta = \{\langle (r_i, s_i), (r_{i+1}, s_{i+1}) \rangle \in Q \times Q \mid (r_i, r_{i+1}) \in \Delta_1$ and $(s_i, s_{i+1}) \in \Delta_2\}$,
- $I = \{(r, s) \in Q \mid r \in I_1$ and $s \in I_2\}$.
- $\mathcal{P}(q) = \mathcal{P}_1(r) \cap \mathcal{L}(s)$, where $r \in Q_1$ and $s \in Q_2$.
- $\mathcal{F} = \{F_1, F_2, \ldots, F_{n+m}\}$ with $F_i = \{(r, s) \in Q \mid r \in F_i^1\}$, for $1 \leq i \leq n$ and $F_i = \{(r, s) \in Q \mid s \in F_{i-n}^2\}$, for $n + 1 \leq i \leq n + m$. Here F_i^1 denotes an acceptance set of $\mathcal{A}_1$ and F_i^2 that of $\mathcal{A}_2$ respectively.

The product is defined analogously for Streett Automata, except that the acceptance sets are defined in the following way.

- $\Omega = \{(L_1, U_1), (L_2, U_2), \ldots (L_{n+m}, U_{n+m})\}$ such that $L_i = \{(r, s) \in Q \mid r \in L_i^1\}$ for $1 \leq i \leq n$ and $L_i = \{(r, s) \in Q \mid s \in L_{i-n}^2\}$, for $n + 1 \leq i \leq n + m$. Replace L_i with U_i to get the definition of the U sets.

Theorem 14 *For two LGBAs or Streett Automata* $\mathcal{L}(\mathcal{A} \times \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$

Proof:
Let ξ be an infinite word. Assume first that $\xi \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. Let $\sigma_{\mathcal{A}}$ be the corresponding run for $\mathcal{A}$ and $\sigma_{\mathcal{B}}$ for $\mathcal{B}$ respectively. The product automaton has the initial states common to the two automata, with the common labels. Therefore a run must start with a state which is common for the automata. Because the states of $\mathcal{A} \times \mathcal{B}$ have the common labels of the two automata and the transition relations takes one step, when both of the two automata can take a step, ξ defines a run σ of $\mathcal{A} \times \mathcal{B}$ which is $\sigma_{\mathcal{A}}$ and $\sigma_{\mathcal{B}}$ synchronized according to the definition of the transition relation. Thus σ is a run of the product automaton if and only if it is defines a run of both automata. The new acceptance condition spelled out dictates that if a there was a condition for acceptance in one of the automata, then it is also included as a condition in the product. The run must thus be accepted by all sets in both automata. Hence, only words which deal with the sets of both automata are accepted, i.e. only words common for both automata pass. Thus we can deduce that $\xi \in \mathcal{L}(\mathcal{A} \times \mathcal{B})$.

Consider the case $\xi \in \mathcal{L}(\mathcal{A} \times \mathcal{B})$. The corresponding run σ of ξ must be a run of both automata because the product automatons transition relation only takes a step when both automata take a step with common labels. Accepting runs for $\mathcal{A}$ and $\mathcal{B}$ can easily be constructed by projecting σ onto its components. As ξ is a word which must be accepted by both automata acceptance condition, it is also a word in the language of both automata (see above). Thus $\xi \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. $\square$

4 Model Checking LTL

Linear temporal logic has established itself as a popular way of specifying properties of reactive systems. It has been deemed expressive enough for most purposes, while retaining a relatively simple syntax and semantics. Although there are several ways of model checking LTL, the most used among tool developers is the automata theoretic approach. Model checking a LTL formula is PSPACE-complete in the length of the formula (see e.g. [10]). This means that long formulas tend to be intractable and that only short formulas can be verified efficiently.

4.1 Linear Temporal Logic

Linear temporal logic (LTL) [23] 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. In LTL each point of time only has one possible future as opposed to *branching time logics* [18]. Hence, for a system to satisfy a LTL formula *all* its executions must satisfy the formula.

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 $\xi = x_0x_1x_2\ldots$ over the alphabet 2^{AP}, i.e. a mapping $\xi : \mathbb{N} \mapsto 2^{AP}$. 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 $x_ix_{i+1}x_{i+2}\ldots$. 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 for $\wedge$, $\Rightarrow$ and $\Leftrightarrow$.

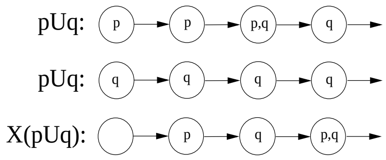

Figure 5: Sequences which satisfy different LTL formulas.

LTL formulas can express a variety of properties. The fairness properties defined in Section 2 can be expressed in the following way:

$$\begin{array}{rl} \text{Weak fairness :} & \Diamond\Box(enabled) \Rightarrow \Box\Diamond(occur),\ \text{or equivalently} \\ & \Box\Diamond(\neg enabled \vee occur) \\ \text{Strong fairness :} & \Box\Diamond(enabled) \Rightarrow \Box\Diamond(occur) \end{array}$$

Expressing that p and q may not hold at the same time is simple.

$$\Box\neg(p \wedge q)$$

More complicated properties do not either require much effort. The following states that a process will always be eventually able to enter the critical section if it tries to.

$$\Box(try \Rightarrow \Diamond crit)$$

Also properties typical to protocols are expressible in LTL. This states that an answer is only possible if a request has been sent before.

$$\Diamond ans \Rightarrow (\neg ans\ U\ req)$$

There are also properties which cannot be expressed in LTL, such as Petri net liveness. Petri net liveness is an example of a proper *branching time* property, and is expressible in CTL. For more discussion see e.g. [34].

There are 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 $\xi \models \varphi$ hold for every execution ξ of Σ.
Fair Model Checking Problem: Given a FCPN Σ_F and an LTL formula ψ, does $\xi \models \psi$ hold for every fair execution ξ of Σ_F.

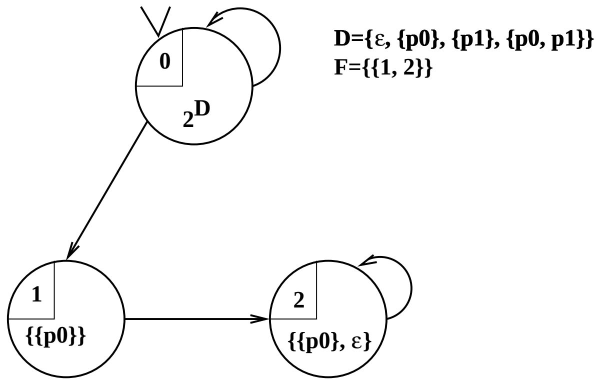

Figure 6: A Büchi automaton corresponding to $\Diamond(p_0 \wedge \Box\neg p_1)$.

4.2 Automata Theoretic Model Checking

The automata theoretic approach to model checking utilizes the intimate relationship between LTL and automata on infinite words. In [31] it was first proven that the set infinite words defined by an LTL formula can be accepted by some automaton on infinite words. Several procedures [10, 6, 7, 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 this is not a problem as they are a special case of the LGBA. An ordinary Büchi automaton can accept the same language as a LGBA, but for a LGBA with n acceptance sets the ordinary Büchi automaton can be n-times larger. Figure 6 shows an automaton corresponding to the formula $\Diamond(p_0 \wedge \Box\neg p_1)$.

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 [5, 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 $\mathcal{F} = \emptyset$.

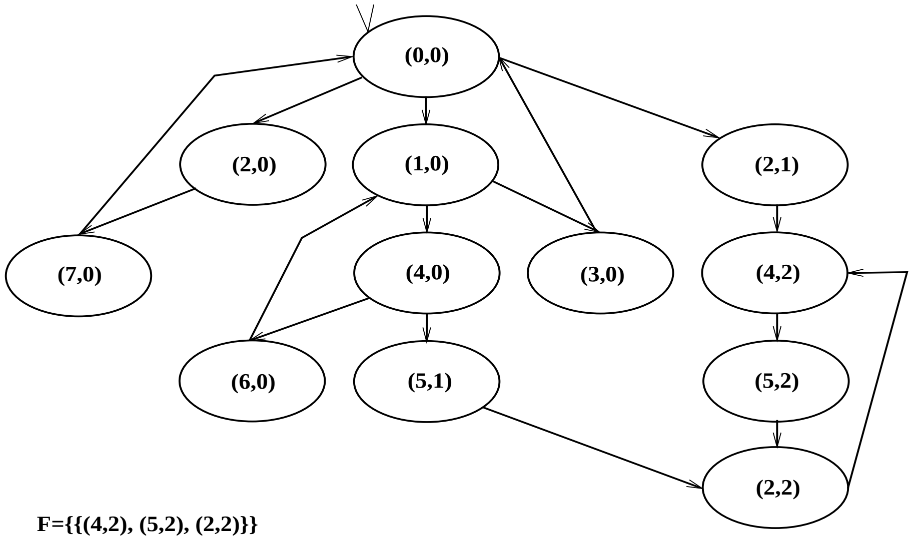

Figure 7: A partial product between Figure 6 and Figure 2

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 [5, 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.

Consider the net in Figure 1. If we want to verify that always process one eventually will gain access if it tries, we first translate the negation of the formula into a Büchi automaton. The result can be seen in Figure 6. The atomic proposition p_0 is interpreted as "the process tries to enter the critical section" and p_1 is interpreted as "the process has successfully gained access to the critical section". Following this the product between the Kripke structure of the net (see Figure 2) and the automaton is generated. Figure 7 shows a partially generated product. By examining the partial product we find that the sequence $\{(0,0),(2,1)(4,2),(5,2),(2,2),(4,2),(5,2),(2,2),\dots\}$ is accepting, because it is an infinite sequence where states belonging to the acceptance set occur infinitely often. Hence the property does not hold. The sequence corresponds to an execution of the net where the second process is always given access to the critical section.

The afore mentioned procedure is not appropriate for model checking a FKS and solving the fair model checking problem. What is needed is a procedure which can handle both generalized Büchi acceptance sets and Streett accep-

```
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 8: The new model checking procedure

tance sets. Of course, the procedure should also avoid using the more time consuming (see e.g [25]) Streett emptiness checking procedure if possible.

To solve the fair model checking problem the new procedure, shown in Figure 8, 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 by assigning the memberships of the fairness sets 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
   - $\exists t\langle b \rangle \in en(M), s' \in Q : \ wf_i(t)\langle b \rangle = \mathit{true}$ 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 15 why this works. If the component is accepted, i.e. it contains all weak fairness set, and has no strong fairness constraints, we can directly generate a counterexample at step 7 with generalized Büchi sets interpreted as Streett acceptance sets U_i and with each L_i set initialized to the universal set and other sets computed according to the definition of a FKS.

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 [8, 21].) 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 simulated with Streett acceptance sets, using the technique given in step 5. Before the component is given to the Streett emptiness algorithm, also the fairness sets L and U must be computed and the necessary intermediate states added. Therefore the MSCC is converted to a FKS according to the definition of a FKS. The correctness of this step is proven in Theorem 16. 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 15 *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, 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 automaton. If a weakly fair counterexample is present in the component, then for all weak fairness sets there are transition instances in such a way that the set is infinitely often disabled or can occur infinitely often. It is easy to see that all sets will be present if this is possible. The construction of the sets guarantees that a state in C belongs to weak fairness set 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 meaning it is possible to construct an execution where it occurs infinitely often. Thus all weak fairness sets are present if there is a weakly fair counterexample, and thus C is non-empty.

If C is non-empty, we know from step 3 of the procedure that the component contains an counterexample. Any execution respecting the acceptance sets of the property is a counterexample. From the previous part of the proof we know a weakly fair execution is present in the component if all the sets are present. 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. $\square$

Theorem 16 *Let C be a MSCC of the product automaton. The component contains a strongly fair counterexample if and only if the Streett automaton C' 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 C', as they are simulated by some Streett sets. From Theorem 11 we know that the sets simulating the weak fairness sets will be present as the counterexample

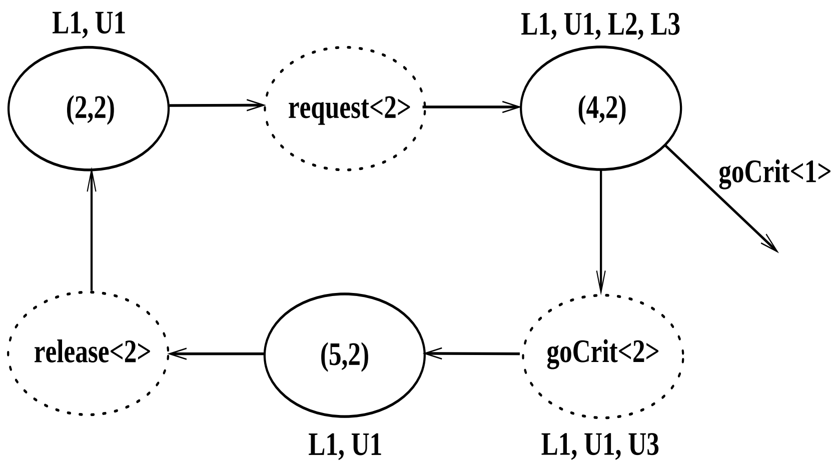

Figure 9: A component of the FKS.

is weakly fair. As the counterexample is also strongly fair for each strong fairness set there are transition instances, such that if the set is infinitely often enabled or the set occurs infinitely often. In C' each state which has a transition instance enabled, belonging to a fairness set belongs to the corresponding L set, thus marking all possible states where some set is enabled. An intermediate state is generated which will belong to the corresponding U set, which will be inside the component 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 the set 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 C' be non-empty. Any execution respecting the property sets will be a counterexample. We know from Theorem 11 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 17 *The new model checking procedure described in Figure 8 solves the fair model checking problem.*

Consider the fair version of the contentious mutex algorithm in Figure 4. If we run the new model checking procedure, trying to verify the accessibility property again for the first process, the component $\{(4,2),(5,2),(2,2)\}$ will be reported as violating the property during the normal model checking phase (see Figure 7). We must now check if the counterexample is weakly and strongly fair. As there are no weakly fair transitions we can skip the weak fairness phase. There are, however, strongly fair transitions and we must hence convert the Kripke structure into a Streett automaton with sets from both the property automaton and the FKS of the FCPN. The set of the property automaton is converted to L_1 and U_1 respectively. The *goCrit* transition has two strong fairness sets; one for each possible instance. The state $(4,2)$ will be member of both L_2 and L_3 because both $goCrit\langle 1 \rangle$ and $goCrit\langle 2 \rangle$ are enabled in $(4,2)$. The intermediate state of the strongly fair transition $goCrit\langle 2 \rangle$ will be member of U_3. When we examine the possible executions, we notice that there is no execution which can satisfy the condition for L_3, as there is no state which is member of U_3 in the component. The component is hence rejected and not considered strongly fair. This is as it should since the transition $goCrit\langle 1 \rangle$ is infinitely often enabled but does not occur infinitely often. If the procedure would be allowed to complete, we would find that no fair counterexample exists.

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 [5] was designed for (non-generalized) Büchi automata, and hence would pay a linear penalty in space in the number of acceptance sets if it were used here. 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, as is that of [11]. An algorithm tailored to handle only generalized Büchi acceptance sets was presented in [6], 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.

4.3 Emptiness Checking of Streett Automata

The emptiness checking problem for Streett Automata is not as easily solved as for Büchi Automata. This is due to the more involved acceptance condition. Formally, we must ascertain if there exists a run ρ of the automaton

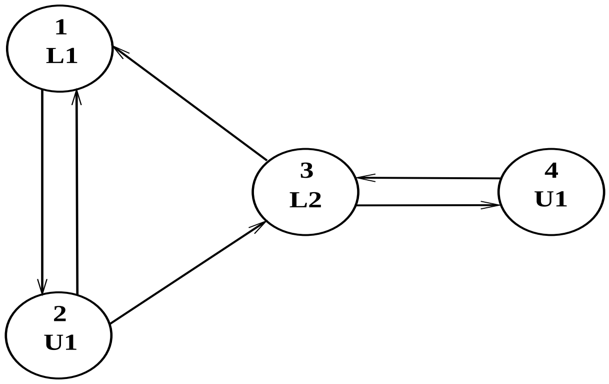

Figure 10: A Streett automaton.

$\mathcal{A} = \langle Q, \Delta, I, \langle (L_1, U_1), \ldots, (L_k, U_k) \rangle, \mathcal{D}, \mathcal{L} \rangle$ such that if a state in a L_i set appears infinitely often in the run, a state from the corresponding U_i set must be in the run for all $1 \leq i \leq k$.

From an algorithmic point of view this means that we must determine if there is cycle (not necessarily simple) in A such that: if the cycle contains state $q \in L_i$ it also contains a state $v \in U_i$ for all $1 \leq i \leq k$.

In the following sections $S \subseteq Q$ denotes a MSCC of the automaton. Also let $bits(S) = \sum_{i=1}^{k} |S \cap L_i| + |S \cap U_i|$, $|S| = n$ and $|\Delta| = m$.

The main idea of the Streett emptiness checking algorithm goes back at least to Emerson and Lei [8]. The same idea was also independently developed in [21]. An improvement on the algorithm was presented in [25]. Several BDD based algorithms also exist [11, 14].

We do the following for all MSCCs of the automaton. The algorithm in Figure 11 receives a MSCC S of the automaton. All the *bad* states of the component are computed. A state $s \in S$ is said to be *bad* if $s \in L_i$ but there is no state $t \in S$ such that $t \in U_i$. The bad states are deleted and the bad states of the remaining states are computed. These two steps are repeated until no more bad states are found. The procedure is repeated because the deletion of a state can cause another state to become a bad state. Tarjan's algorithm is used to compute the MSCCs of the remaining states. Then the above procedure is repeated for all components. If no states are removed from a component it is accepted, if it is non-trivial. A MSCC is non-trivial if it has more than one state or it has a self-loop. Hence, the algorithm in broad terms proceeds by partitioning the states of the component into deleted states, trivial states, and possibly an accepting component also called a good component.

Consider the component in Figure 10. During the first round of the algorithm state three will be deleted, because no state in the component belongs to U_2.

```
proc Empty(S, k) ≡
    Queue Q_1, Q_2;
    List B;
    boolean change;
    InitSets(k);
    C(S) := Construct(S);
    put(Q_1, C(S));
    while (Q_1 ≠ ∅) do
        C(S) := get(Q_1);
        change := false;
        while (B := Bad(C(S)) ≠ ∅) do
            C(S) := Remove(C(S), B);
            change := true;
        od
        if (change AND C(S) ≠ ∅) then
            Tarjan(C(S), Q_2);
            RemoveLargestMSCC(Q_2);
            while (Q_2 ≠ ∅) do
                B := get(Q_2);
                C(S) := Remove(C(S), B);
                put(Q_1, Construct(B));
            od
            put(Q_1, C(S));
        else                                        Good component found!
            if (NotTrivial(C(S))) then
                Counterexample(C(S));
                return true;
            fi
        fi
    od
    return false;
.
```

Figure 11: The emptiness checking algorithm

Tarjan's algorithm will produce two MSCCs: $\{1, 2\}$ and $\{4\}$. The smaller component consisting of only one state will pass the acceptance condition check. No L_i set is without a corresponding U_i set. It is, however a trivial component and hence it will be rejected. The remaining component, consisting of $\{1, 2\}$, is both non-trivial and satisfies the acceptance requirements. The algorithm will accept this as a good component.

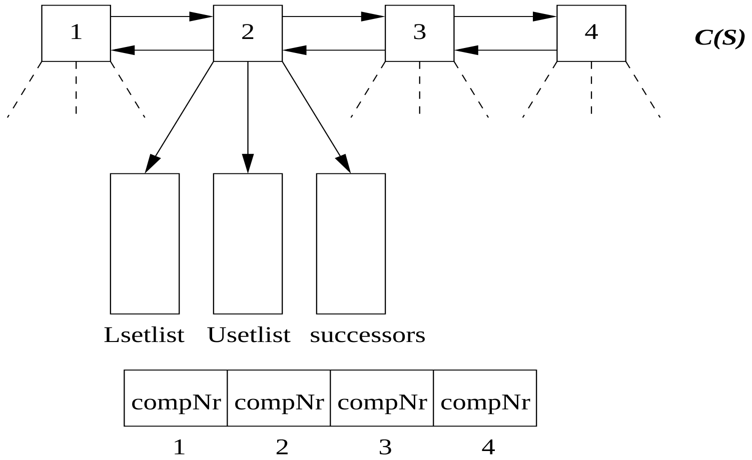

Figure 12: Data structure representing a Streett automaton

4.3.1 Data Structures

The data structures must facilitate fast recomputation of bad states, fast deletion of states and simple implementation of Tarjan's algorithm, while consuming little memory. As memory usually is the critical resource in model checking the data structures have been designed with this in mind.

The transition relation and the L and U sets are represented by similar structures. Each state has three lists associated with it, which contain the successors of the state, the L sets and the U sets of the state respectively. The algorithm uses three global sets $LSet$, $USet$ and $BadSet$, which are implemented as a combination of a bit vector of size k and a stack. After a one time initialization, which takes time $O(k)$, with this implementation set membership can be tested in constant time, union $A := A \cup B$, set intersection $A := A \cap B$, set difference $A := A \setminus B$, and set clear $B := \emptyset$ can be done in $O(|B|)$ time.

The component S is represented by doubly linked list $C(S)$. The list is doubly linked in order to facilitate fast deletions of states. For each state in $C(S)$ we store the component number. Figure 12 shows how $C(S)$, the transition relation, and the fairness sets are related to each other. The algorithm uses the following operations:

Construct(S) initializes and returns the data structure $C(S)$.
Remove$(C(S), B)$ removes B from S and returns $C(S \backslash B)$ for $B \subseteq S \subseteq V$.
Bad$(C(S))$ returns $\bigcup_{1 \leq i \leq k} S \cap L_i | S \cap U_i = \emptyset$.

Lemma 18 *The operation Construct(S) can be implemented with a running time of $O(|S|)$.*

Proof:
The given vertex list S is traversed. For each vertex an entry in the doubly linked list is created. The component number of each state is initialized to the next available component number. □

Lemma 19 *The operation* $Remove(C(S), B)$ *can be implemented with a running time of* $O(|B|)$.

Proof:
Traversing the given list of bad vertices, B, and removing each entry in $C(S)$, and resetting the component number of each state to zero takes time $O(|B|)$. □

The operation $Remove(C(S), B)$ is the main reason why a doubly linked list is used. A simple linked list could in the worst case require a quadratic overhead in time.

The most time consuming operation, together with the recomputation of the MSCCs, is the computation of the bad states.

Lemma 20 *The operation* $Bad(C(S))$ *can be implemented with a running time of* $O(|S| + bits(S))$.

Proof:
Traverse the set lists (see Figure 12) of each vertex in $C(S)$. Whenever a vertex is member of an L_i set or a U_i set add the set number to $LSet$ or $USet$ respectively. This takes time $O(|S| + bits(S))$. Form the set $BadSet = L \backslash U$ and reset $LSet$ and $USet$. This can be done in time $O(min(k, bits(S)) = O(bits(S))$. Add all vertices to a list of bad vertices for which $L.setlist \cap BadSets \neq \emptyset$, reset $Badsets$ and then return the list. This takes time $O(|S| + bits(S))$ giving a total running time of $O(|S| + bits(S))$. □

When compared to [25], this algorithm spends much more time on computing bad states. The algorithm in [25] can access the bad states in constant time, due to its more involved data structures.

4.3.2 Emptiness Checking Algorithm

The algorithm described above can be seen in Figure 11. Correctness of the algorithm can be proved by focusing the attention on how the algorithm partitions states.

Theorem 21 *The emptiness algorithm will find a good component if it exists.*

```
proc Bad(C(S)) ≡
    Set LSet;
    Set USet;
    Set BadSet;
    State s;
    node set;
    List badList;
    forall s ∈ C(S) do
        forall set ∈ s.L.setlist do
            LSet := LSet ∪ {set};
        od
        forall set ∈ s.U.setlist do
            USet := USet ∪ {set};
        od
    od
    BadSet := LSet \ USet;
    clear(LSet);
    clear(USet);
    forall s ∈ C(S) do
        forall set ∈ s.L.setlist do
            if (set ∈ BadSet) then
                append(set, badList);
                break;
            fi
        od
    od
    clear(BadSet);
    return badList;
.
```

Figure 13: The bad algorithm

Proof:
The main loop of the algorithm maintains the invariant that all vertices are either bad, trivial or still in the queue. Initially the algorithm puts all states in the queue. In the second while loop all currently bad states are removed. If the component has changed, the MSCCs of the remaining states are computed. The MSCCs partitions the remaining states into sets and no states are lost. If the component has not changed, it either accepted or deemed trivial. Nowhere are states lost. Hence the invariant holds and the algorithm will find a good component if it exists. □

Analysis of the running time of the algorithm shows that there are two major factor contributing to the running time. The recomputation of the MSCCs and the computation of the bad states. The algorithm in [25] has an improved running time on the computation of the bad states and the computation of the MSCCs.

Theorem 22 *The running time of the algorithm without the Counterexample algorithm is* $O((m + bits(S))\ min(n, k))$

Proof:
The number of calls to Tarjan's algorithm is bounded by $min(n, k)$, because before each call at least one vertex and one fairness set has been taken care of. Thus the total cost contributed by the calls to Tarjan's algorithm is $O(m\ \min(n, k))$. The same factor $\min(n, k)$ bounds the number of calls to *Bad*, *Construct*, and *Remove*. Hence they contribute $O((n + bits(S))$ $\min(n, k)) = O((m + bits(S))\ \min(n, k))$ to the running time giving a total of $O((m + bits(S))\ \min(n, k))$. □

The memory usage of the algorithm is linear in the number of states, edges, sets and members in the fairness sets, as expected. The main drawback of the algorithm of [25] is the memory overhead used in storing the reverse transition relation, and storing $s.L.setlist$ and $s.U.setlist$.

Theorem 23 *The memory usage of the emptiness algorithm is bounded by* $O(n + m + k + bits(S))$

Proof:
The memory for representing the vertices and the edge information accounts for the term $n + m$. The memory required for the $C(S)$ data structure with the Streett set information amounts to $O(n + bits(S))$. Finally the sets *Badsets*, L and U use $O(k)$ memory giving a total of $O(n+m+k+bits(S))$. □

The emptiness algorithm presented above is similar to the algorithm in [25] to the point that this algorithm could be called a simplified version it. The choices made in this algorithm, however, have favored simplicity and memory efficiency over speed. The data structures used in this algorithm are simpler, and thus easier to implement. The algorithm in [25] uses in addition to to L and U set lists for each state $s \in S$, two doubly linked set lists $L_i \cap S$ and $U_i \cap S$ for each $1 \leq i \leq k$, a doubly linked list of at most length k, and a doubly linked list of bad set lists $L_i \cap S$. The algorithm in [25] also requires the predecessor relation of the automaton for the so called lock-step-search case, which we have not included because of memory considerations.

In spite of the more involved data structures the memory usage of [25] is still linear and only larger by a constant factor. Hence, in some cases where memory can be exchanged for a better running time choosing [25] would be wise.

4.4 Counterexample Generation

If the emptiness algorithm finds a good component, we still only know that the property does not hold. For this reason, it is also very important to be able to generate a counterexample to the given property to ease the the location of design errors. The ability of model checkers to generate counterexamples is one reason why model checking is so popular.

The counterexample algorithm is given good component S, from which it should extract a counterexample. A valid counterexample is a cycle, which respects the acceptance condition that if a state $s \in L_i$ is on the path there also is a state $t \in U_i$ on the path. Short counterexamples are preferred because they are considered more informative. Finding a short counterexample is non-trivial, because the path can be a cycle which contains several loops. Actually, determining whether there exists a counterexample of length n, where n is the number of states is NP-complete [2] (proof with a reduction from a Hamiltonian cycle problem). Clearly it is not feasible to expect to get the shortest possible counterexample. A reasonable compromise would for most of the time give short counterexamples, never use excessive amounts of memory, and also have a low upper limit on the running time.

The new algorithm is an extension of the simple breadth-first search. It tries to find a cycle back to the initial state, while respecting the acceptance conditions. The breadth-first search spawns a path tree, where each path is uniquely determined by the corresponding state on bottom level of the tree. Under certain conditions the algorithm chooses a path, i.e. a state. The algorithm chooses a path if it encounters a node s such that

- $s \in L_i$ and we have not encountered L_i previously, or
- $s \in U_i$ and we have encountered L_i previously.

After the path has been chosen, it is printed from memory, and the breadth-first search state is reset. Due to this reset at most a linear amount of memory, in the number of states in the component, is used for book keeping. Then algorithm proceeds with the breadth-first search towards the initial state. Because the intermediate states added in the model checking phase should not affect the length of the path, the algorithm moves immediately to the next state without logging the move when intermediate states are encountered. This is possible because intermediate states always have a unique successor. One counter, a variable called *unseenL*, is maintained by the algorithm to keep track of how many L_i sets have been encountered, for which the corresponding U_i set has not been encountered. The algorithm terminates when it reaches the initial state and *unseenL* equals zero.

The function *checkstate*, see Figure 14, is the function which determines if a path is to be chosen by by investigating the given state according to the

```
proc checkstate (s, seenL, seenU, unseenL) ≡
    boolean lockpath := false;
    forall i ∈ s.L.setlist do
        if (i ∉ seenL) then
            seen_L := seenL ∪ {i};
            lockpath := true;
            if (i ∉ seenU) then
                unseenL++;
            fi
        fi
    od
    forall i ∈ s.U.setlist  do
        if (i ∉ seenU AND i ∈ seenL) then
            unseenL−−;
            lockpath := true;
        fi
    od
    return lockpath;
.
```

Figure 14: The checkstate algorithm.

conditions given above. If a path is chosen the path is printed and all U_i sets on the path are marked as seen by the function *lockpath*.

To analyze the performance of the counterexample algorithm we first consider the auxiliary functions. The running time of the *checkstate* function is clearly dependent on the how many sets the state is member of.

Lemma 24 *The running time of checkstate(s) (see Figure 14) is $O(bits(\{s\}))$.*

Proof:
The function traverses the set list of the states and can in $O(1)$ time check if a specific set has been taken care of. The time required for the traversal is hence $O(bits(\{s\}))$. □

For the function *lockpath* the running time depends on the length of the path and the number of sets the states are member of.

Lemma 25 *The running time of lockpath(s) (see Figure 15) is $O(|S| + bits(S))$*

Proof:
The function must reset the log storing the path, and go through the set

```
proc lockpath (s, seenL, seenU, unseenL) ≡
    Stack stack;
    state t;
    node set;
     do
        push (s, stack);
        forall set ∈ s.U.setlist do
            if (set ∉ seenU) then
                seenU := seenU ∪ {set};
                if (set ∈ seenL) then
                unseenL − −;
                fi
            fi
        od
        t := s;
        s := father (t);                       Go to father
        log_father (t, ∅);                     Reset log
    while (s ≠ 0);
    print_stack (stack);
.
```

Figure 15: The lockpath algorithm.

lists of the vertices in the path, and mark all unseen L_i sets encountered as seen. This gives a running time of $O(|S| + bits(S))$. □

The correctness of the algorithm depends on its ability to produce a path which is adheres to the acceptance conditions. First we consider an interesting special that occurs if the component contains no vertex for which $v \in \bigcup_{1,\ldots,k} L_i$. In this case the search reduces to a simple breadth-first search for a path back to the root. This can be done in linear time and space. The path found is also optimal in the sense that it involves the minimum number of vertices.

Theorem 26 *The Counterexample algorithm finds a counterexample, when given a good component with no vertex belonging to a L_i set, and its running time is $O(n + m + bits(S))$.*

Proof:
Because no state belongs to an L_i set, any path back to the root is a valid counterexample. The algorithm only resets the path if state belongs to an unseen L_i set or a corresponding U_i set, and consequently will not perform a reset. Hence the algorithm does a breadth-first search for a path back to

```
proc Counterexample (C(S)) ≡
    Queue Q;
    state s, root, t;
    Set seenL;
    Set seenU;
    int unseenL := 0;
    root := root (C(S));
    PrintPathTo (root);                              Print prefix.
    visit (root);
    put (Q, root);
    log_father (root, 0);
    while (Q ≠ ∅) do
        s := get (Q);
        if (checkstate (s, seenL, seenU, unseenL)) then
            lockpath (s, seenL, seenU, unseenL);     Print path, reset BFS
        fi
        if (unseenL = 0) then
            forall  t ∈ succ_in_comp(s)  do
                if (t = root) then                   Are we done?
                    return ;
                fi
            od
        fi
        forall  t ∈ succ_in_comp (s) do
            if (¬(visited (t)) then
                visit (t);
                put (Q, t);
                log_father (t, s);                   Store the path.
            fi
        od
    od
.
```

Figure 16: The counterexample algorithm

the root, potentially doing a *checkstate*(s) call at each state. It will find a path to the root (the component is a MSCC), which is the counterexample, achieving the running time of $O(n + m + bits(S))$. □

The proof for the general case is also quite straightforward. Realizing that the algorithm after each reset can find a new set is the key to successfully carry out the proof.

Theorem 27 *The Counterexample algorithm always finds a counterexam-*

ple when given a good component, and its running time is $O((m + bits(S))$ $min(n, k))$.

Proof:
By keeping track for how many L_i sets the corresponding U_i set has not been seen, the algorithm can be guaranteed to terminate only if the path is a valid counterexample. The algorithm stores the traversed path up to a reset. After the reset any state can be visited (the states are always reachable as we are traversing a MSCC). The algorithm will always find a new L_i, or a corresponding U_i after a reset because all states are reachable and visitable and a new reset will not be performed unless any of the above are found or it enters the root state and can terminate. Hence the algorithm will always find an accepting path given a good component. The algorithm performs $\min(n, 2k)$ resets in the worst case. Consequently the algorithm may have to traverse the graph and perform a checkstate at most $\min(n, 2k)$ times. This gives a total running time of $O((n + m + bits(S)) \min(n, k))$. □

As we only keep at most a path of length $|S| = n$ the memory consumption of the algorithm is kept reasonable.

Theorem 28 *Memory usage of the counterexample algorithm is bounded by* $O(n + m + k + bits(S))$.

Proof:
The functions *lockpath* and *checkstate* can use the same sets Bad, $LSet$ and $USet$ for their bookkeeping as the emptiness algorithm. Consequently the algorithm does not need additional data structures to those already created by the emptiness algorithm, except for a breadth-first search log and father log, which only incurs a linear penalty in the number of states n. □

In the worst case this algorithm can produce a counterexample of length $n\ min(n, 2k)$. It is possible to construct an algorithm which in the worst case gives a maximum length of $n\ min(n, k)$, as presented in [14]. In this approach, the algorithm does a breadth-first search for all U_i sets for which the corresponding L_i set is non-empty in increasing i order. In [19] the two approaches were experimentally compared using randomly generated state spaces. In these experiments the new algorithm had a better average performance than [14].

5 Implementation

The model checking procedure described in this work has been implemented in the MARIA analyzer [22]. The MARIA analyzer is a reachability analyzer for algebraic system nets [15, 16, 27], developed at the Laboratory for Theoretical Computer Science at Helsinki University of Technology.

5.1 The MARIA analyzer

The MARIA analyzer, or the ModulAr ReachabIlity 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 FIFOs. There are built-in operations for FIFO operations, multi-set operations, multi-set sums, etc.

The analysis methods of MARIA are still under development. Apart from the model checking feature developed in this work, MARIA supports exhaustive reachability analysis and reachability graph exploration. It is also possible to evaluate expressions in the states of the reachability graph. Simple on-the-fly verification of safety properties is available through detection of constraint violations and other dynamic errors.

5.2 Implementation

The new model checking algorithm was implemented as a module of the analyzer. As an implementation platform MARIA was very suitable because it had almost all of the necessary infrastructure ready. The powerful expression evaluator of the analyzer made the implementation of the computation of the fairness sets straightforward. Also, the decision to use an external implementation of the LTL formula to Büchi automaton translator saved time. Hence, most of the effort was put into implementing the model checking routines.

The implementation was programmed in C++, like the rest of the analyzer. The procedure described in this work was followed quite faithfully in the implementation. Some optimizations were however performed. Management of

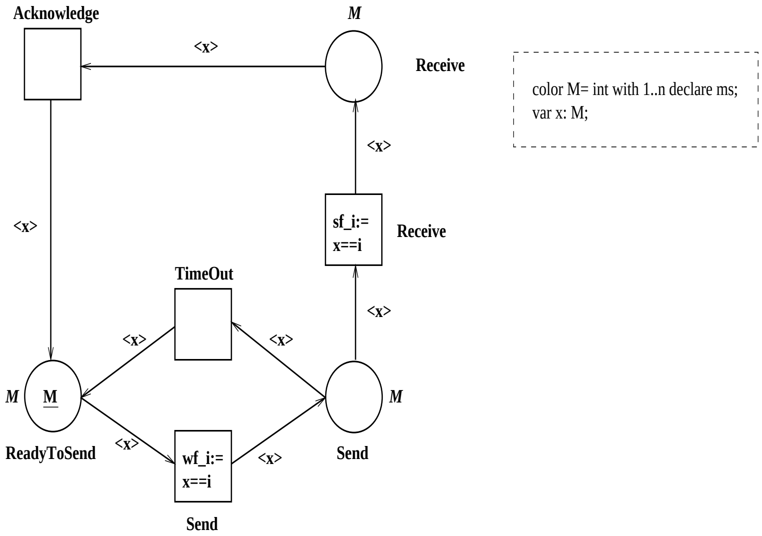

Figure 17: The second test model.

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.

5.3 Experimental Results

In order to get some picture on how well the new model checking procedure performs, some limited experiments were performed with the already familiar contentious mutex model in Figure 4 (see Appendix A for the Maria description) and a second simple model in Figure 17 (see Appendix B for the Maria description). The second model describes a simple communications channel, which can loose arbitrary messages. The mutex model was chosen because it is very simple and yet it includes transitions with strong fairness constraints. There already are tools which automatically support some notion of fairness close to weak fairness [12], which is why testing strong fairness constraints is the most interesting. Initially the plan was to use a more intricate model, but the LTL to Büchi automata translator [29], based on the algorithm presented in [10], could not handle formulas with more than five fairness constraints on the hardware used. The other model was tested because it included both weak and strong fairness constraints and therefore it exercised all aspects of the new algorithm.

To have some kind of reference, all results are compared with the normal approach of specifying the fairness constraints as LTL formulas. All experiments were performed on a PC with a 266 MHz Intel Pentium II processor having 128 MB RAM running the Debian/GNU Linux 2.1 operating system.

Two different properties were verified on the mutex model to highlight two different aspect of the performance of the procedure. The first property was a property which holds in the system, namely accessibility ($\Box(\mathit{pending} \Rightarrow \Diamond \mathit{critical})$). This was tested in order see how fast and how memory efficient the algorithm is by measuring the time needed to verify the formula and the size of the product automaton. The property that the transition *request* adheres to the strong fairness constraint, which does not hold for the system, was used as the second property. With this, it was possible to test how long it took to generate a counterexample and how long they were. Both formulas can be found in Appendix C.1. In all measurements the time to translate the formula to an automaton is also included in the time.

The second model, which included both weak and strong fairness constraints, was only tested for a property which held in the model. Using the old way of including the fairness constraints into the model yielded a product state space of size 577 and took 58 seconds to analyze. The figures for the new procedure were 79 states and two seconds respectively (see the table in Figure 20). Using the new procedure it was possible to increase the size of the parameter, while using the "normal" way the model checking did not complete with $N > 3$. The formulas used can be found in Appendix C.2. Note that for neither of the models, no change was necessary to express the fairness requirement in LTL. The transition occurrences were always visible for both models from the markings, because the output place for each transition was unique. Therefore no model changes were needed.

The results for the first property are summarized in the table Figure 18. Results for the second property are given in the table of Figure 19. The size figures in parenthesis for the new procedure give the size of the good component with intermediate states while the other number is the number of generated product states, without intermediate states. By comparing these two tables it is evident that the new procedure scales much better than giving the fairness constraints as LTL formulas in these experiments. Most of the time consumed by the normal approach is spent in the formula translator, which needs a lot of time to translate the long formulas. Both the time and the size grow exponentially for both the new algorithm and the normal way, which is due to the state space explosion. The lengths of the counterexamples are very similar with only a few steps difference in favor for the normal way.

	New procedure.		"Normal" way	
N	Size	Time	Size	Time (s)
2	21	0	204	1
3	48	0	1919	10
4	109	0	17170	107
5	246	1	145757	1945
6	551	3	-	-
7	1224	6	-	-
8	2697	16	-	-
9	5898	35	-	-
10	12881	82	-	-

Figure 18: Results for the first property.

	New procedure.			"Normal" way.		
N	Size	Time (s)	Witness	Size	Time (s)	Witness
2	14(10)	0	7	94	2	4
3	32(34)	0	10	1786	20	7
4	82(88)	1	13	15233	530	10
5	194(224)	2	16	125555	21997	13
6	450(544)	3	19	-	-	-
7	1026(1280)	5	22	-	-	-
8	2306(2944)	13	25	-	-	-
9	5122(6656)	30	28	-	-	-
10	11266(14848)	80	31	-	-	-

Figure 19: Results for the second property.

	New procedure		"Normal" way	
N	Size	Time	Size	Time
3	79	2	577	58
4	225	3	-	-
5	659	5	-	-
6	1957	14	-	-
7	5847	39	-	-

Figure 20: Results for the second test model.

6 Conclusions

In this work LTL model checking for high-level Petri nets has been extended to cope with Petri nets, which have fairness constraints on the transitions. Semantics for the fairness constraints are given through a fair Kripke structure. The fairness semantics is proved to be equivalent to that given in [13]. Using Streett automata the model checking procedure is extended to handle the fairness constraints in an efficient manner. Also a new algorithm for solving the emptiness problem of Streett automata and for generating a counterexample from an accepting component are presented. Most of the ideas were first presented in the papers [19, 20].

The limited experiments performed indicate that the new procedure for dealing with fairness constraint is clearly superior to the normal way of coping with strong fairness constraints. The experiments also indicated that the method scales fairly well.

Other similar methods have been presented for coping with strong fairness. For CTL model checking the idea was first presented in [8, 21]. The method was extended to LTL and BDDs in [14]. This work, however, is the first to extend the method to high-level Petri nets. Semantics for Petri nets with fairness is not new and is e.g. discussed in [13, 28], but these works do not give any procedure to model check Petri nets with fairness constraints.

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 [8] 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 [34].

References

[1] CCITT. Specification and description language (SDL). Technical report, ITU-T, 1996.

[2] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical Report TR CMU-CS-94-204, School of Computer Science, Carnegie Mellon University, Pittsburg, 1994.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] N. Francez. *Fairness.* Springer-Verlag, New York, 1986.

[10] 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.

[11] 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.

[12] G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[13] K. Jensen. *Coloured Petri Nets*, volume 1. Springer-Verlag, Berlin, 1997.

[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] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[19] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.

[20] Timo Latvala and K Heljanko. Coping with strong fairness – on-the-fly emptiness checking for streett automata. In H.-D. Burkhard, L. Czaja, H-S. Nguyen, and P. Starke, editors, *Proceedings of the Workshop on Concurrency, Specification & Programming (CS& P'99)*, pages 107–118, Warsaw, Poland, September 1999.

[21] O. Lichtenstein and A. Pnueli. Checking that finite state programs satisfy their linear specifination. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[22] M. Mäkelä. Maria: Modular reachability analyzer for algebraic system nets. On-line documentation, 1999. <http://www.tcs.hut.fi/maria>.

[23] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[24] 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.

[25] 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.

[26] W. Reisig. *Petri Nets. An Introduction*, volume 4 of *EATCS Monographs on Computer Science*. Springer-Verlag, 1985.

[27] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, March 1991.

[28] W. Reisig. *Elements of Distributed Algorithms*. Springer-Verlag, Berlin, 1998.

[29] 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/mronkko/LTL2BUCHI/abstract.html>.

[30] S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, 1989.

[31] R. Sherman, A. Pnueli, and D. Harel. Is the interesting part of process logic uninteresting: a translation from PL to PDL. *SIAM Journal on Computing*, 13(4):825–839, 1984.

[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] 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.

[34] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, pages 429–528. Springer-Verlag, 1998. LNCS 1491.

[35] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, pages 238–266. Springer-Verlag, 1996. LNCS 1043.

[36] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, 1986.

A Test Net 1 - Maria Description

```
//Comment out the fairness constraints for the transitions when
//specifying fairness constraints with a LTL formula

//number of processes, should be >=2
unsigned N=2;

//process type
typedef int(1..N) process_t;

//process places
place quiet (0..N) process_t: process_t p: p;
place pending (0..N) process_t;
place critical (0..1) process_t;

//control places
place key (0..1) bool: true;

//transitions
trans Request
in {place quiet:x;} out {place pending:x;};
trans GoCrit
in {place pending:x; place key:z;} out {place critical:x}
strongly_fair x==1 strongly_fair x==2;
trans Release
in {place critical: x;} out {place quiet:x;place key: true};
```

B Test Net 2 - Maria Description

```
//Comment out the fairness constraints for the transitions when
//specifying fairness constraints with a LTL formula
//nr procs
unsigned N=3;

//typedefs
typedef unsigned (1..N) procs_t;

//places
place ReadyToSend procs_t: procs_t p: p;
place Send procs_t;
place Receive procs_t;

//transitions
trans send
in {place ReadyToSend: x;} out {place Send: x;}
weakly_fair x==1 weakly_fair x==2
weakly_fair x==3;
trans timeOut
in {place Send: x;} out {place ReadyToSend: x;};
trans receive
in {place Send: x;} out {place Receive: x;}
strongly_fair x==1 strongly_fair x==2
strongly_fair x==3;
trans acknowledge
in {place Receive: x;} out {place ReadyToSend: x;};
```

C LTL Formulae

C.1 Mutex model

New Procedure

Property 1:

```
verify [](atom(is process_t N subset place pending) =>
<>atom(is process_t N subset place critical));
```

Property 2:

```
verify []<>(atom(is process_t N subset place quiet)) =>
[]<>atom(is process_t N subset place pending);
```

Fairness in Property (N=5)

Property 1:

```
verify ( (([]<>(is process_t 1 subset place pending)) =>
         (([]<>(is process_t 1 equals place critical)))) &&
        (([]<>(is process_t 2 subset place pending)) =>
         (([]<>(is process_t 2 equals place critical))))   &&
        (([]<>(is process_t 3 subset place pending)) =>
         (([]<>(is process_t 3 equals place critical)))) &&
        (([]<>(is process_t 4 subset place pending)) =>
         (([]<>(is process_t 4 equals place critical))))  &&
        (([]<>(is process_t 5 subset place pending)) =>
         (([]<>(is process_t 5 equals place critical)))))
  =>
([]atom(is process_t N subset place pending) =>
<>atom(is process_t N subset place critical));
```

Property 2:

```
verify ( (([]<>(is process_t 1 subset place pending)) =>
         (([]<>(is process_t 1 equals place critical)))) &&
        (([]<>(is process_t 2 subset place pending)) =>
         (([]<>(is process_t 2 equals place critical))))   &&
        (([]<>(is process_t 3 subset place pending)) =>
         (([]<>(is process_t 3 equals place critical)))) &&
        (([]<>(is process_t 4 subset place pending)) =>
```

```
         (([]<>(is process_t 4 equals place critical))))  &&
        (([]<>(is process_t 5 subset place pending)) =>
         (([]<>(is process_t 5 equals place critical)))))
  =>
(([]<>atom(is process_t N subset place quiet)) =>
([]<>atom(is process_t N subset place pending)));
```

C.2 Second Test Model

New Procedure

```
verify [](atom(is procs_t N subset place ReadyToSend) =>
<>atom(is procs_t N subset place Receive));
```

Fairness in Property (N=3)

```
verify  (
        (([]<>(is procs_t 1 subset place Send)) =>
         ([]<>(is procs_t 1 subset place Receive))) &&
        (([]<>(is procs_t 2 subset place Send)) =>
         ([]<>(is procs_t 2 subset place Receive))) &&
        (([]<>(is procs_t 3 subset place Send)) =>
         ([]<>(is procs_t 3 subset place Receive))) &&
         ([]<>atom(!(is procs_t 3 subset place ReadyToSend) ||
        (is procs_t 3 subset place Send))) &&
         ([]<>atom(!(is procs_t 2 subset place ReadyToSend) ||
        (is procs_t 2 subset place Send))) &&
         ([]<>atom(!(is procs_t 1 subset place ReadyToSend) ||
        (is procs_t 3 subset place Send)))
        )
         =>
([](atom(is procs_t N subset place ReadyToSend) =>
<>atom(is procs_t N subset place Receive)));
```