Fundamenta Informaticae 34 (2000) 1–19
IOS Press

Coping With Strong Fairness *

Timo Latvala
Laboratory for Theoretical Computer Science
Helsinki University of Technology
P.O. Box 9700
FIN-02015 HUT
Finland
Timo.Latvala@hut.fi

Keijo Heljanko
Laboratory for Theoretical Computer Science
Helsinki University of Technology
P.O. Box 5400
FIN-02015 HUT
Finland
Keijo.Heljanko@hut.fi

Abstract. We consider the verification of linear temporal logic (LTL) properties of Petri nets, where the transitions can have both weak and strong fairness constraints. Allowing the transitions to have weak or strong fairness constraints simplifies the modeling of systems in many cases. We use the automata theoretic approach to model checking. To cope with the strong fairness constraints efficiently we employ Streett automata where appropriate. We present memory efficient algorithms for both the emptiness checking and counterexample generation problems for Streett automata.

Keywords: Verification, model checking, fairness, Streett automata, counterexamples.

*This research was funded by the National Technology Agency, Nokia Research Center, Helsinki Telephone Corporation and the Finnish Rail Administration. The second author gratefully acknowledges the financial support of Helsinki Graduate School on Computer Science and Engineering (HeCSE), the Academy of Finland (Project 47754), the Emil Aaltonen Foundation and the Nokia Foundation.

1. Introduction

A concurrent and distributed system can in many cases be an efficient and flexible solution for a system developer. Unfortunately concurrent and distributed systems are notoriously difficult to design and implement. They can contain errors which can be extremely hard to find.

The automata theoretic approach to model checking uses the connection between automata on infinite objects and temporal logic to verify that a system meets its specification. Two features of model checking which have made it popular are that it can relatively easy be automated, and that it is often able to produce a counterexample when the system does not meet its specification. The applicability of model checking is, however, seriously limited by the state space explosion problem, see e.g. [18]. One remedy to this problem is performing the model checking on-the-fly. This means that errors might be found without constructing the complete state space of the system being model checked.

When constructing a formal model of a system, different fairness assumptions are often employed [5]. Current model checkers for *linear temporal logic (LTL)* often employ Büchi automata. Büchi automata can express *weak fairness* assumptions efficiently. Since efficient modeling of systems in many cases requires *strong fairness* [5], coping with it in an efficient manner would be desirable. One could argue that since it is possible to express strong fairness in LTL, one can always verify properties in the form "$\textit{fairness} \Rightarrow \textit{property}$". The question is how practical this is from a modeling perspective, and how computationally efficient this approach is. We will discuss the problems associated with this approach in detail in Sect. 2.

A class of automata that can handle strong fairness constraints efficiently are the Streett automata, also known as the complemented pairs automata, see e.g. [17]. In this paper we present an on-the-fly verification method based on a combination of Büchi and Streett automata.

The main contributions of this work are the following. We present how the LTL model checking problem for Petri nets, with fairness constraints imposed transitions, can be solved using Streett automata emptiness checking in a straightforward manner. We present 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. We present simple and memory efficient algorithms for Streett automata emptiness checking and counterexample generation.

The rest of this paper is structured as follows. In Sect. 2 we introduce Petri nets and define a P/T net with fairness constraints. Section 3 covers the needed theory for automata on infinite words. The standard way of performing on-the-fly model checking, and our extension of it, is discussed in Sect. 4. In Sect. 5 we propose an algorithm for performing the emptiness checking of Streett automata and a new algorithm for finding counterexamples. The counterexample algorithm is experimentally evaluated in Sect. 6. In Sect. 7 we present the conclusions and discuss directions for further work.

2. Petri Nets

Petri nets are a widely used model for concurrent and distributed systems. Here we briefly define the basic notation.

Definition 2.1. A net is triple $N = \langle S, T, F \rangle$, where S is a set of places, T is a set of transitions such that $S \cap T = \emptyset$ and $F : (S \times T) \cup (T \times S) \mapsto \mathbb{N}$ is a flow relation.

A *marking* of a net is a mapping $M : S \mapsto \mathbb{N}$. We identify a marking M with a multi-set containing $M(p)$ copies of p for every $p \in S$. A tuple $\langle N, M_0 \rangle$ is a *Place/Transition (P/T) net system* if M_0 is a marking of the net N. We say that a transition $t \in T$ is *enabled* in a marking M if $\forall s \in S : F(s,t) \leq M(s)$. The function $en(M)$ is defined to be a function from the set of markings to the power set of transitions, and it returns all enabled transitions in the marking M. If a transition t is enabled in a marking M, i.e. $t \in en(M)$, the transition t can *occur* changing M into another marking M'. The new marking M' is given by $\forall s \in S : M'(s) = M(s) - F(s,t) + F(t,s)$. The occurrence of a transition is denoted by $M \xrightarrow{t} M'$.

The behavior of a P/T net system can be described by a Kripke structure.

Definition 2.2. The Kripke structure of a P/T net system $\Sigma = \langle S, T, F, M_0 \rangle$ is a triple $K = \langle R, \rho, M_0 \rangle$, where R, a set of reachable states, and ρ, the transition relation, are defined inductively as follows:

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

The executions of the system are infinite sequences $M_0 M_1 M_2 \ldots$ of states in R, where M_0 is the initial state and $(M_i, M_{i+1}) \in \rho$ for all $i \geq 0$. In this work we only consider net systems with a finite set of reachable states.

An example. We consider an example system which we model with a P/T net system. A simple mutual exclusion algorithm, known as the contentious mutex algorithm [12], functions in the following way. We have two parallel processes l and r which at some point must use a shared critical resource, with the restriction that the resource may not be accessed simultaneously. The mutual exclusion is obtained by using a shared boolean variable *key*, which each process checks before using the shared critical resource. The algorithm can be modeled with a P/T net system of Fig. 1. This P/T net model satisfies the mutual exclusion property, but if we carefully examine the behavior of the model we notice that it allows executions of the system where one of the processes never is able to access the shared critical resource even if it is trying. It is possible for a process to infinitely often use the shared critical resource without ever allowing the other process access. In many cases we expect this kind of *unfair* behavior to be impossible which is why we would like to ignore it when applying model checking to the system. One possibility is to add an explicit scheduler to the model, formulate a fairness constraint in LTL for the scheduler and check properties of the form *fairness* $\Rightarrow$ *property*. This solution has several drawbacks. Adding

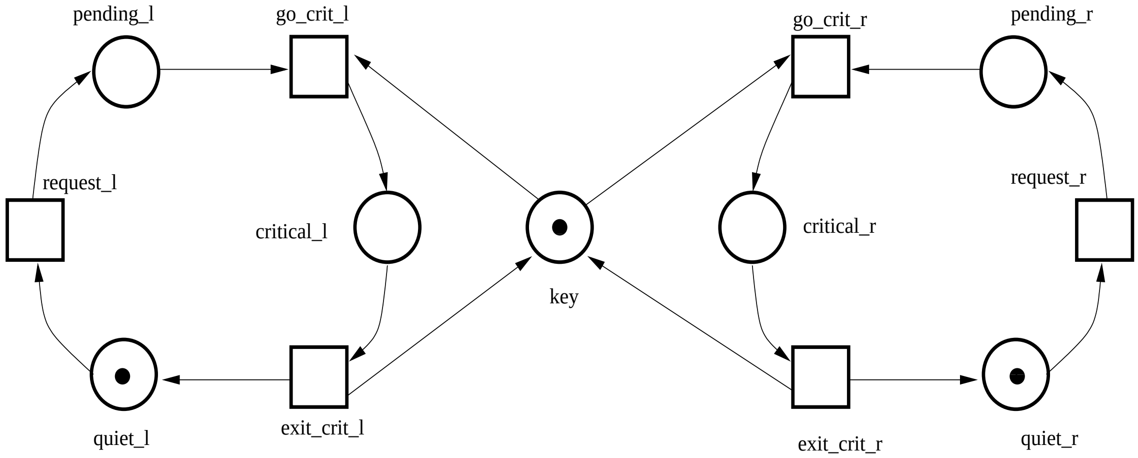

Figure 1. The contentious mutex algorithm

schedulers to the model is cumbersome at best, and when the model is large and there are several independent entities which must be scheduled, it can be a challenging task. Another drawback is that both the state space of the model and the automaton representing the property grow, which makes verification harder. Each fairness constraint requires an additional LTL expression and consequently the growth of the Büchi automaton expressing the property is potentially exponential in the number of fairness constraints, see e.g. [6]. Also, if the fairness constraint used refers to the firing of a transition, say the transition t, then we have to add some state to the model which corresponds to the property "the transition t has just fired" (because LTL is state based and cannot thus in general express this property). Adding such things to the model makes the model with fairness constraints to have more reachable states, and/or less concurrency between transitions than the original P/T net model.

An alternative solution we suggest here is to extend P/T nets with *fairness constraints* on the transitions. For our purpose it is sufficient to distinguish between two notions of fairness. We can adapt the definitions used in [5] for P/T nets. *Weak fairness* requires that the occurrence of a transition will not be indefinitely postponed if it is continuously enabled. A weakly fair scheduler with a waiting queue is guaranteed to eventually schedule an event once the event has entered the queue. Other examples where weak fairness is appropriate are systems with busy waiting or resource-allocation processes. Weak fairness does not, however, cover all situations encountered in modeling. Quite a usual assumption made for communication protocols is that if a message is sent infinitely often it will eventually be successfully received. This assumption could be violated although the communication process would be weakly fair. *Strong fairness* requires that if a transition is infinitely often enabled it must occur infinitely often. Using strong fairness assumptions we can easily model the communciation process so that infinitely often sent messages are received infinitely often.

Adopting these assumptions to transitions in P/T nets we can define fair P/T nets. (Note that our notion of weak fairness differs slightly from Reisig's assumption of progress in [12].)

Definition 2.3. A fair P/T net system is a tuple $N_F = \langle N, M_0, f\rangle$, where N is a P/T net, M_0 an initial marking of the net and $f : T \mapsto \{nf, wf, sf\}$ is a function which maps each transition to a fairness requirement. Here we use "*nf*" for no fairness, "*wf*" for weak fairness and "*sf*" for strong fairness.

A Kripke structure does not properly describe the behavior of a fair P/T net system, because it does not take the fairness constraints into account. We extend the definition of a Kripke structure to be able to describe the behavior of a fair P/T net system. A *fair Kripke structure (FKS)* [9] is a tuple $K_F = \langle R, \rho, r_0, \mathcal{J}, \mathcal{C}\rangle$, where R is a set of states, $\rho \subseteq R \times R$ is a transition relation and $r_0 \in R$ is an initial state. Computations, i.e. fair executions of the system, are infinite sequences $\sigma = r_0 r_1 r_2 \ldots$ of states in R that obey the fairness requirements (to be defined below), where r_0 is the initial state, and for all $i \geq 0$, $(r_i, r_{i+1}) \in \rho$. The fairness requirements are defined by a set of justice requirements[1], or *weak fairness* requirements, $\mathcal{J} = \{J_1, J_2, \ldots, J_k\}$ where $J_i \subseteq R$, and a set of compassion requirements, or *strong fairness* requirements, $\mathcal{C} = \{\langle L_1, U_1\rangle, \ldots, \langle L_m, U_m\rangle\}$ where $L_i, U_i \subseteq R$. By denoting the quantifier "there exist infinitely many" by $\exists^\omega$ we define for notational convenience the set

$$Inf(\sigma) = \{q \in Q \mid \exists^\omega i : \ \sigma(i) = q\}.$$

$Inf(\sigma)$ is the set of states occurring infinitely often in the execution ρ. The justice requirement demands that $\bigwedge_{i=1}^{k} Inf(\sigma) \cap J_i \neq \emptyset$, for every computation σ of K_F. The compassion requirement demands that $\bigwedge_{i=1}^{m} Inf(\sigma) \cap L_i = \emptyset \vee Inf(\sigma) \cap U_i \neq \emptyset$, for every computation σ of K_F.

Generating a FKS from a fair P/T net requires some care. We have to take into account the fairness constraints on the transitions and ensure that the legal computations uphold these constraints. The fairness constraints talk about the occurrence of certain transitions, which is information that is not explicitly available from the set of reachable states or the transition relation of the Kripke structure. One way to remedy this is to add an intermediate state for each transition, so that each occurred transition has an own state in the FKS. Thus the computations of the FKS will be infinite sequences, where the initial state and other states at even indexes will correspond to "normal" states, while the states at odd indexes will correspond to the transitions which occur between these "normal" states. Using this construction the justice and the compassion sets can be used to enforce that only executions which obey the fairness constraints are accepted.

We define the states of the FKS as pairs $\langle M, t\rangle$ in order to separate the intermediate states from the "normal" states. If the state is not an intermediate state t is replaced by a special symbol $\perp$. Hence, to obtain a FKS $K_F = \langle R, \rho, M_0, \mathcal{J}, \mathcal{C}\rangle$ from a fair P/T net system $\Sigma = \langle S, T, F, M_0, f\rangle$, we define R and ρ inductively as follows:

1. $\langle M_0, \perp\rangle \in R$
2. If $\langle M, \perp\rangle \in R$ and $M \xrightarrow{t} M'$ then, $\langle M', t\rangle \in R, \langle M', \perp\rangle \in R$ and $(\langle M, \perp\rangle, \langle M', t\rangle) \in \rho$, $(\langle M', t\rangle, \langle M', \perp\rangle) \in \rho$.

[1]Here we follow the terminology of [9] and speak about justice and compassion.

3. R and ρ have no other elements.

The justice sets and the compassion sets are defined as:

1. For all $t_i \in T : f(t_i) = \mathit{wf}$ the justice sets are $J_i = \{\langle M, \perp \rangle \in R : t_i \notin \mathit{en}(M)\} \cup \{\langle M', t_i \rangle \in R\}$.
2. For all $t_i \in T : f(t_i) = \mathit{sf}$ the compassion sets are $L_i = \{\langle M, \perp \rangle \in R : t_i \in \mathit{en}(M)\}$ and $U_i = \{\langle M', t_i \rangle \in R\}$.

With this definition of the FKS, the justice sets consist of states where the weakly fair transitions are not enabled or the transition has just occurred. As the executions must include states from all the justice sets, we accept only executions where the weakly fair transitions are not continuously enabled, or they occur continuously, and hence we have managed to capture the notion of weak fairness as we intended. The compassion sets L_i consist of states where the strongly fair transition t_i is enabled and the U_i sets of intermediate states where the transition t_i has just occurred. This results in that we only accept executions in which, if a transition t_i is enabled infinitely often, it must occur infinitely often. Consequently, we have also successfully captured the notion of strong fairness with our definition[2].

Running example. If we again consider the contentious mutex algorithm and how we should modify the model we notice that there are two pairs of transition which we must modify. Strong fairness constraints on the *go_crit* transitions ensure that only executions where both processes have access to the critical section, when they are both continuously trying, are accepted. A weak fairness constraint would not be enough because these transitions are disabled if the other process enters the critical section. For the *exit_crit* transitions a weak fairness constraint is enough to ensure that we do not accept executions where one of the processes never leaves the critical section. We not need to place any fairness constraints on the *request* transitions because we want to accept executions where the other process never requests entry to the critical section allowing the other process to use it exclusively. The modified model is shown in Fig. 2.

3. Automata on Infinite Words

The theory of automata on infinite words provides the theoretical foundation we use for model checking LTL. We use state labeled automata in order to be consistent with [6, 13].

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

[2] In an implementation not all of the intermediate states would have to be added. However, these implementation details are beyond the scope of this work.

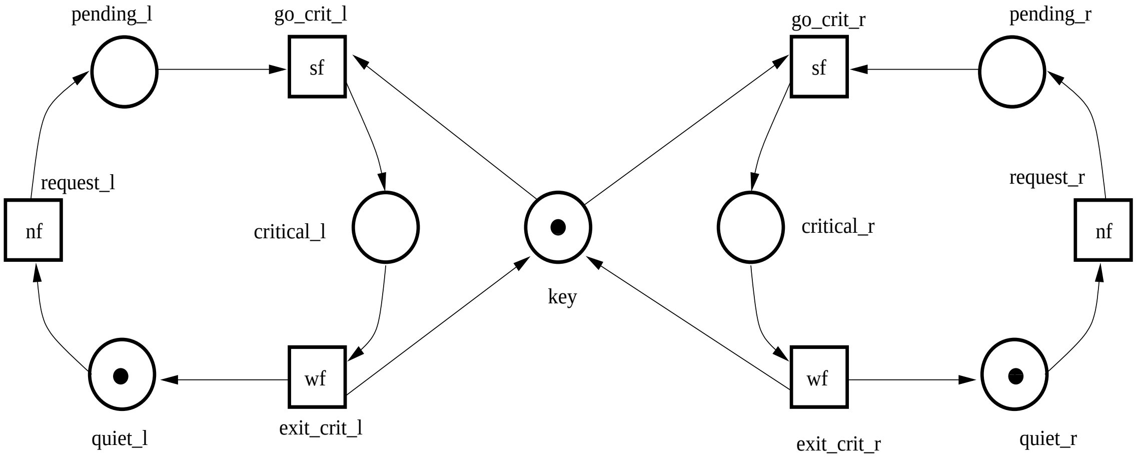

Figure 2. A fair model of the contentious mutex algorithm

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. $Inf(\rho) \cap F_i \neq$ for each $F_i \in \mathcal{F}$. An infinite word $\xi = x_0x_1x_2\ldots \in \mathcal{D}^\omega$ is accepted iff there exists an accepting run $\rho = q_0q_1q_2\ldots$ of $\mathcal{A}$ such that for each $i \geq 0$, $x_i \in \mathcal{L}(q_i)$.

We define a Streett automaton analogously by replacing $\mathcal{F}$ with a set of pairs of acceptance sets and redefine when a run of the automaton is accepting.

A Streett automaton (see [17] for an arc labeled version) is a tuple $\mathcal{A} = \langle Q, \Delta, I, \Omega, \mathcal{D}, \mathcal{L}\rangle$, where Q, Δ, I, $\mathcal{D}$ and $\mathcal{L}$ 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 $\rho = q_0q_1q_2\ldots$ if $\bigwedge_{i=1}^{k}(Inf(\rho) \cap L_i = \emptyset \vee Inf(\rho) \cap U_i \neq \emptyset)$. We can read this 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”. Intuitively the acceptance condition corresponds to strong fairness conditions as defined in [5]. We define the set of infinite words accepted by $\mathcal{A}$ analogously to the LGBA case, using the new acceptance condition Ω.

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. Testing whether $\mathcal{L}(\mathcal{A}) = \emptyset$ is referred to as performing an emptiness check. Note that generalized Büchi acceptance can be easily simulated by Streett acceptance by letting $L_i = Q$ and $U_i = F_i$ for all $1 \leq i \leq k$. However there is no polynomial translation from Streett to Büchi automata, see e.g. [14]. Also note that the FKS discussed in the previous section can be seen as a Streett automaton, as the justice requirements correspond to generalized Büchi acceptance sets (which can be simulated with Streett sets as described above), and the compassion requirements are Streett acceptance sets. All we need to do is to add a labeling, which labels each state of the FKS with the (unique) set of atomic propositions which hold in that state.

4. LTL Model Checking

In order to reason about the reactive behavior of the system, in our case the behavior of the Petri net model, one must be able to reason about its infinite computations. One suitable candidate is *linear temporal logic* (LTL). LTL is interpreted over infinite computations and can therefore be used for specifying properties of reactive, non-terminating systems.

Using the idea that infinite computations can also be viewed as infinite words over 2^{AP}, where AP is a set of atomic propositions, it was shown in [15] that there is an automaton on infinite words that accepts exactly the computations satisfying a given LTL formula. Later in [19] an explicit construction was given to convert an LTL formula φ into a Büchi automaton $\mathcal{A}_\varphi$ which accepts exactly the computations satisfying φ. A refined algorithm was presented in [6] which performed the translation to a generalized Büchi automaton on-the-fly. This gives us a method to check whether a system conforms to its specifications using automata-theoretic constructions. Fix a labeling of the reachable states of the net system, which labels each state with the set of atomic propositions that hold in that state.

The steps performed to verify that a system has a property given by a LTL formula φ are the following [2, 10]:

1. Construct a 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 Büchi automaton $\mathcal{K}$.
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 system satisfies the specification. Combining several steps of this approach in a single algorithm is referred to as "on-the-fly" LTL model checking [2, 10].

A limitation of the on-the-fly method described in [2] is that it can only deal with normal (non-generalized) Büchi automata. To handle generalized Büchi automata, the algorithm needs to translate these into Büchi automata, see e.g. [6]. This can also be done on-the-fly, but the number of states of the resulting product automaton can be the number of states of the original product automaton times the number of generalized Büchi acceptance sets.

As previously mentioned Büchi automata cannot handle strong fairness efficiently, as there is no polynomial translation from Streett to Büchi automata, see e.g. [14]. Weak fairness, however, is manageable with generalized Büchi automata. By combining the best of both worlds it is possible to deal with both strong and weak fairness and verify claims given in LTL in an efficient manner.

We propose the following procedure:

1. Construct a generalized Büchi automaton $\mathcal{A}_{\neg\varphi}$.
2. The fair Kripke structure K_F of the system is generated from the Petri net model of the system and interpreted both as a generalized Büchi automaton and as a Streett automaton. The interpretation is easy as the justice requirements correspond to generalized Büchi acceptance conditions and the compassion requirements correspond to Streett acceptance conditions. We call this automaton the FKS automaton.

3. The product automaton of the above two is created on-the-fly using a procedure similar to that of [2][3]. The main difference is the handling of the acceptance sets. The states of the product automaton obtain acceptance conditions from both the formula- and the FKS automaton, and thus have to fulfil both the fairness conditions imposed by the system and Büchi acceptance conditions imposed by the formula. Simultaneously Tarjan's algorithm [16] is used to calculate the next *maximal strongly connected component (MSCC)* of the product automaton.
4. When a MSCC of the product automaton has been calculated, we check for generalized Büchi acceptance (ignoring Streett acceptance sets for a moment). If the component does not contain a state from each Büchi acceptance set, i.e. it does not contain a weakly fair counterexample and hence is not accepted, we return to step 3.
5. If a component is accepted as weakly fair and it does not contain strong fairness constraints, we can directly generate a counterexample at step 7 using only generalized Büchi sets interpreted as Streett acceptance sets U_i and with each L_i set initialized to the universal set.
6. We know now that the MSCC contains a weakly fair counterexample. To ensure that there is also some strongly fair counterexample we have to use a Streett emptiness checking algorithm on this MSCC. (Using the Streett emptiness checking to handle strong fairness constraints goes back to at least [4, 11].) However, we cannot yet ignore the generalized Büchi acceptance sets. Therefore each generalized Büchi acceptance sets is simulated with a Streett acceptance set using the same technique as in the step 5. If no weakly and strongly fair counterexample is found, we continue from step 3 with a next MSCC of the product automaton.
7. A counterexample is generated using the subset of vertices of the MSCC, which the emptiness checking algorithm gives to the counterexample algorithm.

The steps 1-3 can be done by a single on-the-fly algorithm, as the product creation and Tarjan's algorithm can be implemented on-the-fly (see e.g. [7] for an on-the-fly CTL model checker based on Tarjan's algorithm). If the property automaton has generalized Büchi acceptance sets $\{F_1, F_2, \ldots, F_n\}$, and the FKS automaton has the justice sets $\{J_1, J_2, \ldots, J_m\}$, then the product automaton will have the generalized Büchi sets $\{F_1, F_2, \ldots, F_n, F_{n+1}, \ldots, F_{n+m}\}$. The Streett sets are directly inherited from the compassion sets of the FKS automaton. Note that the steps 4 and 5 are not needed for correctness, they are only an optimization to avoid the more costly Streett emptiness check whenever possible. By performing the verification in this on-the-fly manner it is possible to find errors without computing all MSCCs of the product automaton, which might result in faster running times. Also the fact that only components which contain weakly fair counterexamples, and have enabled transitions with strong fairness requirements, are checked for the existence of a counterexample satisfying also strong fairness, potentially results in less work compared to a naive implementation.

[3]The synchronization actually synchronizes the property automaton only with "normal" states of the FKS. Thus the FKS automaton takes two steps for each step of the property automaton. The implementation of this small change is quite straightforward.

To our knowledge the procedure in exactly this form has not been presented in the literature. The on-the-fly model checker of [2] uses only (non-generalized) Büchi automata in the nested-depth-first-search algorithm. The algorithm of [9] 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 [8]. The use of Tarjan's algorithm in emptiness checking is well known, see e.g. [10]. The only Tarjan based emptiness checking algorithm we are aware of which explicitly claims to be on-the-fly is that of [3]. However, it won't work for our purposes, as it is tailored to handle only generalized Büchi acceptance sets.

5. Emptiness Checking of Streett Automata

The emptiness checking algorithm is given a MSCC of the product Streett automaton. The product Streett automaton can be seen as directed graph $G = (V, E)$, where the number of vertices $|V| = n$ and the number of edges $|E| = m$. The Streett pairs (L_i, U_i), $L_i, U_i \subseteq V$ with $1 \leq i \leq k$ are given and $bits(S)$ is defined as $\Sigma_{i=1}^{k}|S \cap L_i| + |S \cap U_i|$ for $S \subseteq V$. Performing an emptiness check on a Streett automaton is then to check whether G contains a cycle such that: if the cycle contains a vertex from L_i then it also contains a vertex from U_i, for all $1 \leq i \leq k$.

5.1. Emptiness Checking Algorithm

The main idea of the Streett automata emptiness checking algorithm goes back to at least Emerson and Lei [4], and it was also independently developed in [11]. The algorithm is given a maximal strongly connected component (MSCC) of the product automaton calculated by Tarjan's algorithm [16]. The algorithm begins dynamically modifying the graph by deletion of so called *bad* vertices from the MSCC. A vertex is bad if it belongs to some L_i set, but the MSCC it belongs to does not contain a vertex from the corresponding U_i set. All other vertices are good. A MSCC containing only good vertices is said to be a good component. After the deletion of bad vertices the MSCCs of the modified graph are recalculated and checked again for bad vertices. The algorithm terminates when it has either found a non-trivial good component or it can conclude that no such component exists. (A MSCC is non-trivial if it has more than one vertex, or it has a single vertex with a self-loop. Otherwise it is trivial.)

The algorithm presented here is similar to that of [13] to the extent that we could call our emptiness checking algorithm a simplified version of it. The data structures designed in this work are simpler, and thus also easier to implement. As an example, the data structures of the algorithm of [13] have double linked lists for each set L_i, which enables them to easily access all of the states in $L_i \cap S$. This implies that each set membership "bit" must have at least two pointers associated with it. To obtain this set of states, we will do extra work, however we save the storage for these two pointers. The algorithm of [13] has also a so-called lock-step-search case, which is also missing here due to memory consumption reasons, as it requires the reverse transition relation of the product automaton to be available.

The simpler data structures were a conscious design choice, motivated by the fact that in many cases the amount of memory available is the limitation in model checking. The memory overhead of [13] is larger by only a constant factor, though. Therefore, implementing it might be a better choice in some cases.

5.2. Data Structures

The algorithm needs to keep track of bad vertices and into which MSCC each vertex belongs to as the original MSCC may split into several MSCCs during the emptiness checking. It is also necessary to keep track of which fairness sets are present in the currently processed component. The notation in this section has been chosen in order to be consistent with [13].

We use three global sets L, U, and *Badsets* of size k which are implemented as a combination of a stack and a bitmap. They require one-time initialization which takes $O(k)$ time. This is done the first time the emptiness checking algorithm is called. With this implementation set membership can be tested in $O(1)$ time. Set union $A := A \cup B$, set difference $A := A \setminus B$, and set clear $B := \emptyset$ can be done in $O(|B|)$ time.

The data structure $C(S)$ stores the component information, and for each vertex the information into which sets L and U sets the vertex belongs to. It is implemented using a doubly linked list containing all the vertices of the MSCC. From each node in this list there are pointers to set lists which specify to which L and U sets the vertex belongs to. These sets are referred to as *L.setlist* and *U.setlist* respectively. Each vertex also records to which MSCC it belongs by using a component number.

As the original component may split into several MSCCs during the run of the algorithm, a queue Q is kept where the different components are stored. We define the following operations for the data structure $C(S)$:
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 $\bigcup_{1 \leq i \leq k} \{S \cap L_i \mid S \cap U_i = \emptyset\}$ for $S \subseteq V$.

Lemma 5.1. *The operation Construct(S) can be implemented with a running time of $O(|S|)$.*

Proof:
The given vertex list S is traversed. Each vertex is added to the doubly linked list of C(S). □

Lemma 5.2. *The operation Remove$(C(S), B)$ can be implemented with a running time of $O(|B|)$.*

Proof:
Traversing the given list of vertices B, and removing each entry from the doubly linked list of $C(S)$ takes time $O(|B|)$. □

Lemma 5.3. *The operation Bad$(C(S))$ can be implemented with a running time of $O(|S| + bits(S))$.*

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

Theorem 5.1. *The emptiness checking algorithm will find a good component if it exists.*

Proof:
The main loop of the algorithm maintains the invariant that all vertices are either bad, belong to a trivial MSCC, or are still in the queue. The algorithm initially puts all vertices in the queue. In the second while loop all bad vertices are deleted, causing the MSCC to be recomputed and the remaining vertices put back into the queue and again tested for bad vertices. If a component has no bad vertices the component is accepted, unless it is trivial. Hence the invariant holds and the algorithm will find a good component if it exists. □

Theorem 5.2. *The running time of the algorithm without the Counterexample subroutine is* $O((m + bits(V))min(n, k))$

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

Theorem 5.3. *The memory usage of the emptiness checking algorithm is bounded by* $O(n+m+k+bits(V))$

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 $O(n + bits(V))$. Finally the sets *Badsets*, L, and U use $O(k)$ memory giving a total of $O(n + m + k + bits(V))$. □

5.3. The MSCC search

The MSCC are computed using the Tarjan's algorithm [16]. It is modified to perform the search for maximal strongly connected components in the subgraph containing only the nodes and edges of the component $C(S)$. From each found MSCC a list is created, which contains all the states in that MSCC. These lists are then stored in the queue Q_2.

```
proc Empty (S, k) ≡
    Queue Q1, Q2;
    List B;
    boolean change;
    InitSets (k);                                  Initialize sets L, U, Badsets
    C(S) := Construct (S);
    put (Q1, C(S));
    while (Q1 ≠ ∅) do
        C(S) := get (Q1);
        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), Q2);                     Recalculate the MSCCs
            RemoveLargestMSCC (Q2);                Any MSCC will do
            while (Q2 ≠ ∅) do
                B := get (Q2);
                C(S) := Remove (C(S), B);
                put (Q1, Construct (B));
            od
            put (Q1, C(S));
        else                                       Good component found!
            if (NotTrivial (C(S))) then
                Counterexample (C(S));             Generate counterexample
                return true;
            fi
        fi
    od
    return false;                                  No good component exists
.
```

Figure 3. The emptiness checking algorithm

5.4. The Counterexample Algorithm

Generating a counterexample to the given property is very important to help in the location of design errors. The counterexample algorithm given here produces a counterexample after the emptiness checking algorithm has passed it a good component. Finding a counterexample is non-trivial because the counterexample can be a cycle which contains several loops. Short counterexamples are preferred as they are considered more informative and do not contain much unnecessary information.

The algorithm we propose searches in a breadth-first manner from the MSCC entry vertex, which we will henceforth refer to as the root, for a path back to the root. The path must of course satisfy the requirement that if there is a vertex $v_i \in L_l$ in the path, the path must also include a vertex $v_j \in U_l$, for all $1 \leq l \leq k$. As the breadth-first search spawns a path tree, one must choose which path to use. The algorithm freezes the path traversed to the current vertex when

- the vertex belongs to an unseen L_i set, or
- the vertex belongs to an unseen U_i set corresponding to a previously encountered L_i set.

The traversed path is then printed from memory, the breadth-first search state is reset using logs and then search for a path back to the root can proceed. The resetting allows one to minimize memory requirements because the algorithm only keeps at most one simple cycle of the counterexample path stored in memory. The algorithm terminates if it reaches the root and the traversed path satisfies the requirements stated above. To know when to terminate, the algorithm keeps track of the number of encountered L_i sets for which the corresponding U_i set has not been found using the variable *unseen_L*.

The function that determines whether to freeze the breadth-first search is called *checkstate*. It returns true if we are in a vertex $v \in L_i$, and no state belonging to L_i has been seen before. It also returns true if the vertex v belongs to an unseen U_i set for which a corresponding vertex $u \in L_i$ has already been seen. The printing of the path from memory is done by *lockpath*. This function also marks all U_i sets in the locked path which have not been encountered before as seen U_i sets, and resets the breadth-first search state.

An interesting special case occurs if the component contains no vertex for which $v \in \bigcup_{1 \leq i \leq k} L_i$. 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 5.4. *The Counterexample algorithm finds the counterexample, when given a good component with no vertex belonging to a L_i set, and its running time is $O(n + m + bits(V))$.*

Proof:
Because no vertex belongs to an L_i set the algorithm will not reset. Hence the algorithm does a breadth-first search for a path back to the root, potentially doing a *checkstate*(s) call once for each state. It will find a path to the root, which is the counterexample, achieving the running time of $O(n + m + bits(V))$. □

Lemma 5.4. *The running time of checkstate(s) is $O(bits(\{s\}))$ for $s \in S$.*

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

```
proc checkstate(s) ≡
    Set seen_L;
    Set seen_U;
    boolean lockpath := false;
    integer unseen_L;
    forall v ∈ s.L.setlist do
        if (v ∉ seen_L) then
            seen_L := seen_L ∪ {v};
            lockpath := true;
            if (v ∉ seen_U) then
                unseen_L++;
            fi
        fi
    od
    forall v ∈ s.U.setlist do
        if (v ∉ seen_U AND v ∈ seen_L) then
            unseen_L−−;
            lockpath := true;
        fi
    od
    return lockpath;
.
```

Figure 4. The checkstate algorithm which determines when the BFS path is frozen

Lemma 5.5. *The running time of lockpath(s) is $O(|S| + bits(S))$*

Proof:
The function must reset the log storing the path, and go through the set lists of the vertices in the path and mark all unseen L_i sets encountered as seen. This gives a running time of $O(|S| + bits(S))$. □

Theorem 5.5. *The Counterexample algorithm always finds a counterexample when given a good component, and its running time is $O((m + bits(S))min(n, k))$.*

Proof:
The algorithm stores the traversed path up to the reset using the BFS search logs created by the subroutine *log_father*. After the reset any state can be visited (the states are always reachable as we are traversing a MSCC). Thus the algorithm will always find a new $s_i \in L_l$, or a corresponding $s_j \in U_l$ 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 and can terminate. Hence the algorithm will always find an accepting path given a good component. The algorithm

```
proc Counterexample (C(S)) ≡
    Queue Q;
    state s, root, t;
    root := root (C(S));
    PrintPathTo (root);                          Print prefix to the loop using search logs
    visit (root);
    put (Q, root);
    log_father (root, 0);
    while (Q ≠ ∅) do
        s := get (Q);
        if (checkstate (s)) then                 Do we freeze the BFS?
            lockpath (s);                        Print path and reset the BFS state
        fi
        forall t ∈ succ_in_comp(s) do            Check if we are done
            if (t = root AND unseen_L_i = 0) then
                return ;
            fi
        od
        forall t ∈ succ_in_comp (s) do           Put the successors in the queue
            if (¬(visited (t)) then
                visit (t);
                put (Q, t);
                log_father (t, s);               Store the path
            fi
        od
    od
.
```

Figure 5. The counterexample 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 $O((n + m + bits(S))min(n, k))$. □

Theorem 5.6. *The 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 *Badsets*, L and U for their book-keeping as the emptiness checking algorithm. Consequently the algorithm does not need additional data structures to those already created by the emptiness checking algorithm, except for a breadth-first search log and a father log created by the subroutine *log_father*, which only incurs

a linear penalty in the number of states n. □

The maximum length of the counterexample is $n\ min(n, 2k)$. There is a slightly different approach presented in the literature. In this approach the algorithm always goes trough all U_i sets (in increasing i order) for which the corresponding L_i set is non-empty. Using this idea it generates a counterexample which has a maximum length of $n\ min(n, k)$ [9] (see also [8]). (It would in fact be quite easy to first run both of the algorithms "silently" on the input, only calculating the length of the created counterexample. When these lengths would be known, one could then choose the better algorithm for the counterexample output to the user.) Deciding whether there exists a counterexample of length n, where n is the number of nodes is in fact NP-complete [1] (proof with a reduction from a Hamiltonian cycle problem).

6. Experimental Results

In order to evaluate the performance of the counterexample algorithm, it was compared against the non-lazy algorithm suggested in [9]. Both algorithms were implemented in Java and tested with randomly generated graphs[4]. As the algorithms perform roughly the same amount of work, their speed is not an interesting thing to compare. The critical measure of their performance is the ability to produce short counterexamples.

The algorithms were evaluated in the following way. Random graphs $G = (V, E)$ of size N were generated with k fairness sets, and $(v_i, v_j) \in E$ with a probability ρ. Each vertex $v \in V$ belongs to a L_i set or a U_i set with the probability p. The emptiness algorithm was used to calculate a good component of the graph, which was given to the two counterexample algorithms. The length of the counterexample was then recorded.

There are several possibilities to investigate how the algorithms scale when varying some parameter. The one we used is to see how the algorithms differ when the number of fairness sets is varied. This shows how the algorithms cope when the model has more fairness constraints, which make the counterexamples more complex. This was tested by letting $k \in \{5, 15, 25, 35, 45, 55\}$ and generating twenty graphs for each value of k. The values of the other parameters were $N = 600, \rho = 0.05$ and $p = 0.1$. In Fig. 6 we can see the results averaged over the twenty times. The error bars represent the standard deviation of the runs. From the figure it can be seen that the new algorithm scales better than the non-lazy algorithm of [9]. Especially when the number of fairness sets grows, the lazy nature of the new algorithm gives it an edge.

7. Conclusions

Including both weak and strong fairness constraints in the modeling language one can make the modeling of many systems more straightforward than when the fairness constraints are supplied by an LTL formula (and potential changes in the model to accommodate this). However especially the handling of strong fairness efficiently requires changes to the LTL model checking procedure.

[4]The sources can be found from http://www.tcs.hut.fi/~timo/FI-Experiment.

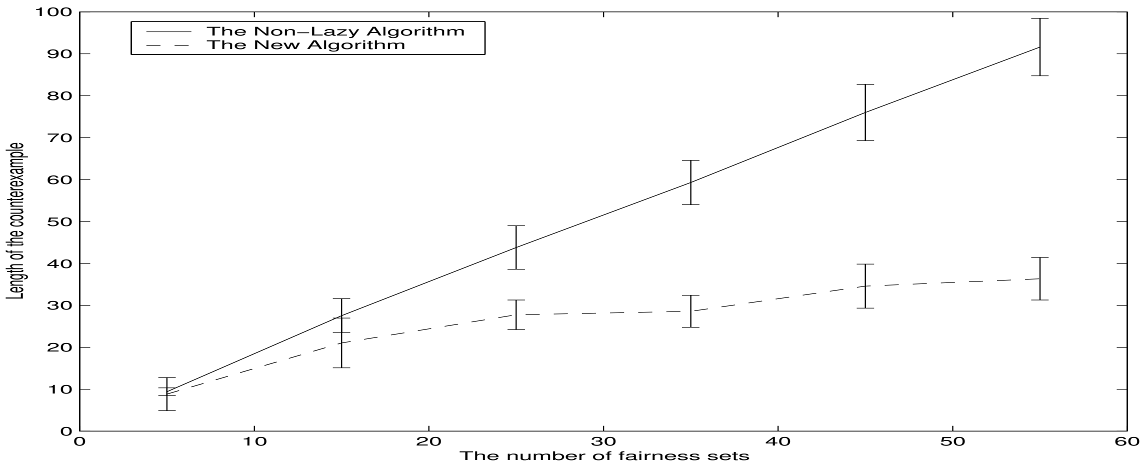

Figure 6. Performance of the two algorithms

We have presented an approach which makes a set of design choices on how to handle strong fairness efficiently. We give semantics to fair P/T nets through fair Kripke structures borrowing notation from [9]. Our proposed on-the-fly LTL model checking procedure presented in Sect. 4 is new, and it tries to avoid the costly Streett emptiness checking whenever possible. The emptiness checking algorithm with data structure design choices were motivated by [13], however we choose to use more time in order to save some memory. The counterexample generation algorithm is new, and seems to sometimes work better than the algorithm of [9]. We have also implemented the emptiness checking algorithm and the counterexample algorithm in Java.

As future work we would like to extend this approach to model checking CTL* under fairness assumptions. Also the effect of the fairness constraints on the partial-order reduction algorithms such as stubborn sets and persistent sets (see e.g. [18]) needs to be investigated. Also lifting the fairness notions to high-level Petri nets should be investigated from the perspective of modeling convenience.

Acknowledgements

The authors also wish to thank Tommi Junttila for helpful criticisms and comments on earlier versions of this work, and Heikki Tauriainen for sharing his random Kripke structure generator with us.

References

[1] Clarke, E., Grumberg O., McMilllan K. and Zhao, X.: Efficient Generation Counterexamples and Witnesses in Symbolic Model Checking. *Technical Report TR CMU-CS-94-204*, Carnegie Mellon University, School of Computer Science, Pittsburg, 1994.

[2] Courcoubetis, C., Vardi, M., Wolper, P. and Yannakakis, M.: Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, vol 1. pp. 275-288, 1992.

[3] Couvreur, J.-M.: On-the-fly Verification of Linear Temporal Logic. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), Volume I*, pp. 253–271, Springer, 1999. LNCS 1708.

[4] Emerson, E.A. and Lei, C-L.: Modalities for Model Checking: Branching Time Logic Strikes Back. *Science of Computer Programming*, vol. 8, no. 3, pp 275-306, 1987.

[5] Francez, N.: *Fairness*. Springer Verlag, New York, 1986.

[6] Gerth, R., Peled, D., Vardi, M.Y. and Wolper, P.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. *Proceedings of the 15th Workshop on Protocol Specification, Testing and Verification*, pp. 3-18. Chapman and Hall, Warsaw Poland, 1995.

[7] Heljanko, K.: Model Checking the Branching Time Temporal Logic CTL. *Research Report A45*, Digital Systems Laboratory, Helsinki University of Technology, 1997.

[8] Hojati, R., Singhal, V., and Brayton, R.K.: Edge-Strett/ Edge-Rabin Automata Environment for Formal Verification Using Language Containment. *Memorandum No. UCB/ERL M94/12*, Electronics Res. Lab., Cory Hall, University of California, Berkeley, 1994.

[9] Kesten Y., Pnueli, A. and Raviv, L.: Algorithmic Verification of Linear Temporal Properties. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP 1998)*, Lecture Notes in Computer Science, vol. 1443, pp. 1-16. Springer-Verlag, 1998.

[10] Kurshan, R.P.: *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.

[11] Lichtenstein, O. and Pnueli, O.: Checking that finite state concurrent programs satisfy their linear specifications. *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pp 97-107, 1985.

[12] Reisig, W.: *Elements of Distributed Algorithms*, Springer Verlag, Berlin Heidelberg, 1998.

[13] Rauch Henzinger, M., Telle, J.: Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT'96)*, pp. 10-20. 1997.

[14] Safra, S.: *Complexity of Automata on Infinite Objects*, PhD Thesis, The Weizmann Institute of Science, 1989.

[15] Sherman, R., Pnueli, A. and Harel, D.: Is the Interesting Part of Process Logic Uninteresting: a Translation From PL to PDL. *SIAM Journal on Computing*, vol. 13, no. 4, pp. 825-839, 1984.

[16] Tarjan, R.: Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, vol. 1, no. 2, pp 146-160, 1972.

[17] Thomas, W.: Languages, Automata and Logic, in: *Handbook of Formal Languages* (G. Rozenberg, A. Salomaa, Eds.). Vol III, pp. 385-455, Springer-Verlag, New York, 1997.

[18] Valmari, A.: The State Explosion Problem, in: *Lectures on Petri Nets I: Basic Models*, pp. 429–528, Springer, 1998. LNCS 1491.

[19] Vardi, M.Y., Wolper, P.: Reasoning About Infinite Computations. *Information and Computation*, vol. 115, no. 1, pp. 1-37, 1994.