Proc. Estonian Acad. Sci. Phys. Math., 2003, **52**, 4, 394–412

Fairness in automata theoretic model checking

Tuomo Malinen and Matti Luukkainen

Department of Computer Science, University of Helsinki, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland; {tamaline,mluukkai}@cs.helsinki.fi

Received 10 February 2003, in revised form 13 June 2003

Abstract. We describe and perform empirical evaluation of two different methods for taking fairness assumptions into account in automata theoretic model checking. Both of the methods are known from the literature, even though they have not been presented in this form before. One of these methods is based on incorporating the fairness assumption into the system model; in the other one the fairness assumption is taken care of algorithmically. The goal of this work is to put these methods together, present them in a unified formal framework, and to compare their relative usefulness in some problems.

Key words: automata theory, model checking, Büchi automata, fairness, verification.

1. INTRODUCTION

We study techniques for taking *fairness* into account in automatic verification of finite-state systems. Fairness of a computer system means informally that the different processes of the system each get a fair share of processor time if they wish and thus can advance in their computation. Fairness is needed to secure the *liveness* properties of the verified system. Obviously, fairness of the real system needs to be taken into account in formal verification of these properties.

Our context is, more specifically, automata theoretic model checking [1]. In automata theoretic model checking the verified system is modelled as a composition of several automata, typically describing the processes and shared memory of the system. Automata are also used to describe the correct behaviour of the system. The actual verification can be reduced to checking the emptiness of the language accepted by an automaton.

There exist different notions of fairness: *weak fairness* and *strong fairness*. We study methods for verification under both of these.

Two general approaches to dealing with fairness have been proposed. Lichtenstein and Pnueli [2] present algorithms for model checking of LTL formulae under (both kinds of) fairness assumptions. These are based on dividing the system model into its strongly connected components and studying their properties. In a certain kind of strongly connected component, a fair run of the system is guaranteed to exist. The model checking problem with fairness assumptions can thus be solved. In this approach, the supposed fairness of the verified system is taken into account *algorithmically*, i.e., in the model checking algorithm. The system model needs not be altered.

The other approach is to embed the fairness assumption into the system model. In our context, this is done by adding monitor automata to the composition of the system model. The idea is that these automata limit the system to only its fair computations [3−5]. The model checking is then usually performed with nested depth first search of the model [6]. The studies mentioned above are, however, limited to studying only weak (or some closely related form of) fairness. In automata theory based LTL model checking [1,7,8], essentially the same effect as the above is obtained by incorporating the fairness assumption into the property, i.e., instead of doing the model checking for a formula φ, it is done for implication $\varphi_{\text{fair}} \rightarrow \varphi$ where φ_{fair} is an LTL formula which evaluates true for fair paths only.

In this study, we wish to give rigorously defined methods for both approaches lined out above, and for both kinds of fairness. The main contribution is the empirical comparison of the computational complexity of these methods [9].

The article is structured as follows. In Section 2 we first discuss the basic setting of automata theoretic verification. In Section 3 we give definitions needed to give the model checking problem with fairness assumptions. In Section 4 we first discuss the algorithmic aspects of the two different approaches mentioned above. In Section 5 we discuss the relative theoretical and empirical performance of these methods. Finally, in Section 6, we draw some conclusions and propose future work.

2. AUTOMATA THEORETIC VERIFICATION

In this section we define the notation and basic concepts needed to formulate the automata theoretic verification in the context of fairness requirements.

2.1. Büchi automaton

The systems to be verified as well as the verified properties are described with Büchi automata [10,11].

Definition 1 (Büchi automaton).
A Büchi automaton is a tuple $B = (S, \Sigma, \Delta, s0, F)$*, where*

- S *is a finite set of states,*
- Σ *is a finite alphabet,*

- $\Delta \subseteq S \times \Sigma \times S$ *is the transition relation,*
- $s0 \in S$ *is the initial state, and*
- $F \subseteq S$ *is the set of accepting states.*

Definition 2. *Consider a Büchi automaton* $B = (S, \Sigma, \Delta, s0, F)$.

- *Input of a Büchi automaton is an infinite string* $\overline{a} = a_1 a_2 a_3 \cdots$ *of symbols in* Σ, so $\overline{a} \in \Sigma^\omega$. *In the sequel we call these infinite strings* words.
- *An* execution *of an automaton* B *over the word* $\overline{a}$ *is the infinite sequence of states* $\overline{s} = s_0 s_1 s_2 \cdots$ *provided that*
  - $s_0 = s0$, *and*
  - *for all* $i > 0 : (s_{i-1}, a_i, s_i) \in \Delta$.
- *For an execution* $\overline{s}$ *let* $\inf(\overline{s})$ *be the set of states* $s \in S$ *that occur in the sequence infinitely often, i.e.,* $s \in \inf(\overline{s})$ *iff* $s = s_i$ *for infinitely many* $i \geq 0$. *A Büchi automaton* B accepts *a word* $\overline{a}$ *iff it has an execution* $\overline{s}$ *over the word* $\overline{a}$ *such that* $\inf(\overline{s}) \cap F \neq \emptyset$.
- *The set of all words that an automaton* B *accepts is called the* language that it defines, *and we denote the language with* $L(B)$, *formally*:

$$L(B) = \{\, \overline{a} \in \Sigma^\omega \mid B \; accepts \; \overline{a} \,\}.$$

Thus Büchi automata are string acceptors that can be used to define languages consisting of infinite words of some alphabet. In the automata theoretic verification, a view is adopted where a Büchi automaton is seen as a way to define what are the infinite computations that a system can take part in. States of automata naturally describe the internal states of computation and transitions describe the execution of commands. Thus the alphabet defines the "commands" that the system executes during its lifetime.

An automaton, where all the states are accepting states, is called a Büchi automaton with a *trivial acceptance condition*.

In addition to the concept of normal Büchi automaton, we also need an extension of it, the generalized Büchi automaton, which is defined next.

Definition 3 (Generalized Büchi automaton).

- *A generalized Büchi automaton is a tuple* $B^g = (S, \Sigma, \Delta, s0, \mathcal{F})$, *where the rest of the components are the same as with normal Büchi automata, but* $\mathcal{F}$ *is now the* set of acceptance states, i.e., $\mathcal{F} \subseteq 2^S$.
- *An* execution *of generalized Büchi automaton over a word* $\overline{a}$ *is defined similarly as in the case of normal Büchi automata*.
- *A generalized Büchi automaton* B^g accepts *a word* $\overline{a}$ *iff it has an execution* $\overline{s}$ *over the word* $\overline{a}$ *such that for all* $F \in \mathcal{F}$*:* $\inf(\overline{s}) \cap F \neq \emptyset$.
- *The language of a generalized Büchi automaton* B *is defined as in the case of normal Büchi automata*.

Generalized and normal Büchi automata have the same expressive power [11], and a normal Büchi automaton can be obtained from a generalized one with the following construction [12].

Theorem 4. *Let $B^g = (S^g, \Sigma, \Delta^g, s0^g, \{F_1, \ldots, F_k\})$ be a generalized Büchi automaton. Now $B = (S, \Sigma, \Delta, s0, F)$, where*

- $S = S^g \times \{1, \ldots, k\}$,
- $((s_1, i), a, (s_2, j)) \in \Delta$ *if and only if*
  - $(s_1, a, s_2) \in \Delta^g$, $s_1 \notin F_i$ *and* $i = j$,
  - $(s_1, a, s_2) \in \Delta^g$, $s_1 \in F_i$ *and* $j = (i + 1) \mod k$,
- $s0 \in (s0^g, 1)$, *and*
- $F = F_1 \times \{1\}$,

is a Büchi automaton for which $L(B^g) = L(B)$.

2.2. Product automaton

Usually the systems which are to be verified are described so that each component is defined by means of a separate automaton, and the Büchi automaton defining the composite behaviour of the system is then obtained as a *product* of the components. Note that the components do not necessarily have the same alphabet. The symbols that are shared between several automata denote the communication events of the automata whereas such symbols that are used by only one automaton denote some internal activity of the components.

Definition 5. *Let $B_i = (S_i, \Sigma_i, \Delta_i, s0_i, F_i)$ be Büchi automata for $i \in \{1, \ldots, n\}$. The product $B_1 \times \cdots \times B_n$ is the (generalized) Büchi automaton $B = (S, \Sigma, \Delta, s0, \widehat{F})$, where*

- $S = S_1 \times \cdots \times S_n$,
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$,
- $s0 = (s0_1, \ldots, s0_n)$, *and*
- $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ *if and only if for all* $i \in \{1, \ldots, n\}$:
  - *if* $a \in \Sigma_i$ *then* $(s_i, a, s'_i) \in \Delta_i$,
  - *if* $a \notin \Sigma_i$ *then* $s_i = s'_i$.
- *For the acceptance state information $\widehat{F}$ we distinguish two different cases*:
  - *if $F_i = S_i$ for all $i \in \{1, \ldots, n-1\}$ (only one of the components, the component numbered n, possibly has a nontrivial acceptance condition), then B is a normal Büchi automaton, and $\widehat{F} = S_1 \times \cdots \times S_{n-1} \times F_n$,*
  - *if $F_i \neq S_i$ at least for two $i \in \{1, \ldots, n\}$ (at least two of the components have nontrivial acceptance conditions), then B is a generalized Büchi automaton, and $\widehat{F} = \{S_1 \times \cdots \times S_{n-1} \times F_n, \ldots, F_1 \times S_2 \times \cdots \times S_n\}$.*

2.3. Verification by language inclusion

Assume that we model the system with a Büchi automaton B which is obtained as a product of automata $B_1, \ldots, B_n$ which define the individual system

components. Now $L(B)$, the language defined by B corresponds to all possible computations of the system which is to be verified. If we use yet another automaton B_φ to describe all the correct computations with respect to the property φ, the system meets the requirement if and only if all of its computations belong to the ones B_φ is accepting, i.e., $L(B) \subseteq L(B_\varphi)$.

The language inclusion problem of the Büchi automaton can be solved by reducing it to the emptiness problem of the Büchi automaton in the following way [1]. Let $B_{\overline{\varphi}}$ be the complement of automaton B_φ, i.e., $B_{\overline{\varphi}}$ is an automaton accepting a word if and only if B_φ does not accept it. Now the following result can be proven:

Theorem 6. *$L(B) \subseteq L(B_\varphi)$ iff $L(B) \cap L(B_{\overline{\varphi}}) = \emptyset$ iff $L(B \times B_{\overline{\varphi}}) = \emptyset$.*

Thus the verification is reduced to the emptiness problem of a single Büchi automaton, and as shown in Section 3, the emptiness of an automaton can be solved algorithmically (i.e., the emptiness of a Büchi automaton is a decidable problem). The intuition behind the result is that since $L(B)$ defines all the possible computations of the system and $L(B_{\overline{\varphi}})$ defines all the computations that do not conform to the property φ, those sets should be disjoint. If there is a common element $\overline{a}$ in the sets, it is a computation of the system that fails to satisfy the property.

Note that the set of languages definable with Büchi automata (the ω-regular languages) is closed under set theoretical operations, thus it is known that B_φ can be complemented [13]. The automata theoretic verification method extends easily to LTL model checking since every LTL formula can be translated to a corresponding Büchi automaton [1,8].

3. FAIRNESS

The product B defined in the previous section thus contains all the possible computations of the system. However, some of the computations may not occur in real life because of the scheduling mechanisms of the system. $L(B)$ contains also computations where, for example, component B_1 which is always ready to execute is never given a turn to execute. If we are verifying *liveness* properties, such as "if *request* is sent, *response* has to arrive within a finite amount of time", we should be able to filter out these *unfair* computations from our system model B. Let us now formally define the concept of *fairness* [14,15].

Definition 7 (Fairness).

Let us consider automata $B_1, \ldots, B_n$ and B, where $B_i = (S_i, \Sigma_i, \Delta_i, s0_i, F_i)$ and $B = B_1 \times \cdots \times B_n = (S, \Sigma, \Delta, s0, F)$, and an infinite computation $c_0 c_1 c_2 \cdots$ of B. Note that c_i is now a vector belonging to the set $S_1 \times \cdots \times S_n$.

- *B_i is* enabled *in $(s_1, \ldots, s_i, \ldots, s_n) \in S$ iff there exists a transition $((s_1, \ldots, s_i, \ldots, s_n), a, (s'_1, \ldots, s'_i, \ldots, s'_n)) \in \Delta$ such that $a \in \Sigma_i$ and $(s_i, a, s'_i) \in \Delta_i$, i.e., B_i takes part in the transition.*

- $c_0c_1c_2\cdots$ *is* weakly fair *with respect to* B_i *iff the following condition holds*: *if* B_i *is enabled for all* c_j*, starting from a state* c_k*,* $k \geq 0$*, then infinitely many of the transitions* $(c_j, a, c_{j+1}) \in \Delta$ *are such that* B_i *has taken part in the transition.*
- $c_0c_1c_2\cdots$ *is* strongly fair *with respect to* B_i *iff the following condition holds*: *if* B_i *is enabled for infinitely many* c_j*, then infinitely many of the transitions* $(c_j, a, c_{j+1}) \in \Delta$ *are such that* B_i *has taken part in the transition.*
- $c_0c_1c_2\cdots$ *is a* weakly fair *computation of* B *if it is weakly fair with respect to all of the automata* $B_1, \ldots, B_n$*.*
- $c_0c_1c_2\cdots$ *is a* strongly fair *computation of* B *if it is strongly fair with respect to all of the automata* $B_1, \ldots, B_n$*.*

Some kind of a fairness assumption is usually needed if liveness properties are verified. Taking fairness into account, the verification question can be formulated in the following way:

$$L(B) \cap L(B_{\text{fair}}) \subseteq L(B_\varphi),$$

which is equivalent to

$$L(B \times B_{\text{fair}} \times B_{\overline{\varphi}}) = \emptyset,$$

where B_{fair} is an automaton that accepts a computation $c_0c_1c_2\cdots$ of the system if and only if the computation is fair. Thus it is enough that just the fair computations of the system satisfy the property. Fairness depends on the enabledness of individual system components, thus the automaton B_{fair} cannot be defined within the framework we have so far. Thus, some additional notation has to be defined first.

Definition 8 (Enabledness labelled Büchi automaton).

Let $B_i = (S_i, \Sigma_i, \Delta_i, s0_i, F_i)$ *be Büchi automata for* $i \in \{1, \ldots, n\}$*, and* $B = (S, \Sigma, \Delta, s0, F)$ *the product* $B_1 \times \cdots \times B_n$*. The* enabledness labelled Büchi automaton *of* B *is* $B^l = (S, \Sigma, \Delta, s0, F, E)$*, where* $E \subseteq S \to 2^{\{1,\ldots,n\}}$ *is defined as follows*:

$$\forall s \in S : E(s) = \{\ i \in \{1, \ldots, n\}\ \mid B_i\ is\ enabled\ in\ s\ \}.$$

If the Büchi automaton B describes a system, the enabledness labelled version of it, B^l, is the same automaton and in addition, each state is labelled with numbers of those subcomponents that are enabled in that state.

In addition to enabledness labelled automata, we need another extension, the monitor Büchi automata, which can use the enabledness information in the transitions.

Definition 9 (Monitor Büchi automata).

Consider a system consisting of the components $B_1, \ldots, B_n$*, denote with* Enc *the set* $\{en_1, dis_1, \ldots en_n, dis_n\}$*. The monitor Büchi automaton corresponding to the components* $B_1, \ldots, B_n$ *is a tuple* $M = (S, \Sigma, \Delta, s0, F)$*,*

- S *is a finite set of states,*
- Σ *is a finite alphabet,*

- $\Delta \subseteq S \times \Sigma \times 2^{Enc} \times S$ *is a transition relation,*
- $s0 \in S$ *is the initial state, and*
- $F \subseteq S$ *is the set of acceptance states.*

Let $M_i = (S_i, \Sigma_i, \Delta_i, s0_i, F_i)$ *be monitor automata for* $i \in \{1, \ldots, n\}$. *The product* $M_1 \times \cdots \times M_n$ *is the generalized monitor Büchi automaton* $M = (S, \Sigma, \Delta, s0, \mathcal{F})$, *where*

- $S = S_1 \times \cdots \times S_n$,
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$,
- $((s_1, \ldots, s_n), a, l, (s'_1, \ldots, s'_n)) \in \Delta$ *if and only if for all* $i \in \{1, \ldots, n\}$:
  - *if* $a \in \Sigma_i$ *then* $(s_i, a, l_i, s'_i) \in \Delta_i$,
  - *if* $a \notin \Sigma_i$ *then* $s_i = s'_i$, *and*
  - $l = l'_1 \cup \cdots \cup l'_n$, *where* $l'_i = l_i$ *if* $a \in \Sigma_i$ *and in other case* $l'_i = \emptyset$.
- $s0 = (s0_1, \ldots, s0_n)$, *and* $\mathcal{F} = \{S_1 \times \cdots \times S_{n-1} \times F_n, \ldots, F_1 \times S_2 \times \cdots \times S_n\}$.

A generalized monitor Büchi automaton can be transferred to a normal monitor automaton with the method described in Theorem 4.

Finally, let us define the product of monitor and enabledness labelled automata, yielding a generalized Büchi automaton.

Definition 10. *Let* $B^l = (S^l, \Sigma, \Delta^l, s0^l, F^l, E^l)$ *be the enabledness labelled Büchi automaton of the product* $B_1 \times \cdots \times B_n$ *and* $M = (S^m, \Sigma, \Delta^m, s0^m, F^m)$ *monitor automaton corresponding to automata* $B_1, \ldots, B_n$. *The product* $B^l \times M$ *is now defined as the generalized Büchi automaton* $(S, \Sigma, \Delta, s0, \mathcal{F})$, *where*

- $S = S^l \times S^m$,
- $s0 = (s0^l, s0^m)$,
- $((s_1, s'_1), a, (s_2, s'_2)) \in \Delta$ *if and only if*
  - $(s_1, a, s_2) \in \Delta^l$,
  - $(s'_1, a, l, s'_2) \in \Delta^m$, *and*
  - *for all* $x \in l$: *if* $x = en_i$ *then* $i \in E(s_1)$, *if* $x = dis_i$ *then* $i \notin E(s_1)$.
- $\mathcal{F} = \{S^l \times F^m, F^l \times S^m\}$.

With an enabledness labelled automaton we can represent a system which is to be verified in such a way that monitor automata can be used to define the fairness constraints. Thus, we have defined all the machinery needed to represent fairness constraints in a uniform manner within automata theoretic verification. Next, let us define monitor automata for weak and strong fairness.

Definition 11. *Let* $B_1, \ldots, B_n$ *be the automata defining the system under verification, and denote the alphabet of automaton* B_i *with* Σ_i. *Monitor for weak fairness is* $B_{\text{wf}} = (S, \Sigma, \Delta, s0, F)$, *where*

- $S = \{0, 1, \ldots, n\}$,
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$,
- $s0 = 0$,
- $F = \{0\}$, *and*

- $\Delta = \{(0, a, \emptyset, 1) \mid a \in \Sigma\} \cup$
  $\bigcup_{i \in \{1,\ldots,n\}} (\{(i, a, en_i, i) \mid a \in \Sigma \setminus \Sigma_i\}$
  $\cup \{(i, a, \emptyset, (i+1) \bmod n) \mid a \in \Sigma_i\}$
  $\cup \{(i, a, dis_i, (i+1) \bmod n) \mid a \in \Sigma \setminus \Sigma_i\})$

Monitor for strong fairness B_{sf} *is the product* $B_{\mathrm{sf}_1} \times \cdots \times B_{\mathrm{sf}_n}$. *For all* $i \in \{1, \ldots, n\}$, *the components are defined as follows* $B_{\mathrm{sf}i} = (S_i, \Sigma, \Delta_i, s0_i, F_i)$, *where*

- $S_i = \{0, 1, 2\}$,
- $s0 = 0$,
- $F = \{1, 2\}$,
- $\Delta_i = \{(0, a, \emptyset, 0) \mid a \in \Sigma\} \cup \{(0, a, \emptyset, 2), (2, a, \emptyset, 2) \mid a \in \Sigma_i\}$
  $\cup \{(0, a, dis_i, 1), (1, a, dis_i, 1), (2, a, \emptyset, 0) \mid a \in \Sigma \setminus \Sigma_i\}$.

The intuition behind the monitor automata B_{wf} and B_{sf} becomes clear from the proof of the following theorem which formulates the automata theoretic verification under a fairness assumption [9].

Theorem 12. *Let* B *denote the product* $B_1 \times \cdots \times B_n$, *which describes the system to be verified, and let* $B_{\overline{\varphi}}$ *be the automaton describing the negation of the property* φ *which is to be verified, and let* B_{wf} *and* B_{sf} *be the fairness automata of the previous definition.*

1. B *satisfies* φ *assuming weak fairness iff* $L((B^l \times B_{\mathrm{wf}}) \times B_{\overline{\varphi}}) = \emptyset$.
2. B *satisfies* φ *assuming strong fairness iff* $L((B^l \times B_{\mathrm{sf}}) \times B_{\overline{\varphi}}) = \emptyset$.

Proof. In both of the cases the theorem follows directly from the fact that the fairness automaton accepts a computation if and only if the computation is fair.

1. Consider a weakly fair computation $\overline{s}$. Now for each automaton B_i, either the automaton is disabled infinitely often, or infinitely many of the transitions in the computation are such where B_i is taking part. This implies that the automaton B_{wf} can only wait a finite number of steps in its state i. Thus, the weakly fair computation $\overline{s}$ causes B_{wf} to enter its acceptance state infinitely often.

Consider then a computation $\overline{s}$ that causes B_{wf} to enter its acceptance state infinitely often. This means that computation passes infinitely often through state i, which in turn means that the automaton B_i has either taken part in infinitely many transitions or has been disabled infinitely many times. Thus, the computation $\overline{s}$ is weakly fair.

2. Consider a strongly fair computation $\overline{s}$. Now for each automaton B_i, either the automaton is disabled from some state on, or infinitely many of the transitions in the computation are such where B_i is taking part. This implies that the automaton $B_{\mathrm{sf}i}$ passes infinitely many times through an accepting state. Obviously the automaton B_{sf} then passes infinitely many times through each of its acceptance sets, and thus, B_{sf} too accepts the strongly fair computation.

Consider then a computation $\overline{s}$ that B_{sf} accepts. Equivalent is that B_{sf} visits some component from each of its acceptance sets infinitely many times. This implies that the automaton $B_{\mathrm{sf}i}$ passes infinitely many times through an accepting

state. So, the automaton B_i has either taken part in infinitely many transitions, or has been disabled from some state on. Thus, the computation $\overline{s}$ is strongly fair. □

Note that we can easily modify the monitors B_{wf} and B_{sf} such that fairness is assumed from just a subset of the processes $B_1, \ldots, B_n$. In B_{wf} waiting states are needed only for those processes for which fairness is required, and in B_{sf} a component $B_{\mathrm{sf}\,i}$ is needed only if fairness is assumed for the process B_i.

4. ALGORITHMS FOR CHECKING THE EMPTINESS

In the previous section we showed how Büchi automata can be used in verification under fairness assumptions. Theorem 12 showed that verification under fairness can be reduced to the emptiness problem of a single automaton. Let us now consider the algorithmic aspects of the emptiness checking. The following result (see, e.g., [1,3]) is the key to the algorithmic solvability of the emptiness problem.

Theorem 13. *Let $B = (S, \Sigma, \Delta, s0, F)$ be a Büchi automaton. The following are equivalent*:

1. $L(B) \neq \emptyset$.
2. *There exists $s_f \in F$ such that*:
   - $\exists s_1, \ldots, s_n \in S, a_1, \ldots, a_{n-1} \in \Sigma$ *such that* $\forall i \in \{1, \ldots, n-1\}$: $(s_i, a_i, s_{i+1}) \in \Delta$, $s_1 = s0$ *and* $s_n = s_f$, *and*
   - $\exists s_1, \ldots, s_n \in S, a_1, \ldots, a_{n-1} \in \Sigma$ *such that* $\forall i \in \{1, \ldots, n-1\}$: $(s_i, a_i, s_{i+1}) \in \Delta$, $s_1 = s_f$ *and* $s_n = s_f$, *where* $n > 1$.
3. *There exists a strongly connected component C in the state space of B such that*:
   - $\exists s_1, \ldots, s_n \in S, a_1, \ldots, a_{n-1} \in \Sigma$ *such that* $\forall i \in \{1, \ldots, n-1\}$: $(s_i, a_i, s_{i+1}) \in \Delta$, *and* $s_1 = s0$ *and* $s_n \in C$.
   - $C \cap F \neq \emptyset$.
   - $\exists s, s' \in C$ *such that for some* $a \in \Sigma : (s, a, s') \in \Delta$.

Intuitively, the language defined by an automaton is nonempty if there exists an accepting state which is reachable from the initial state and from the state itself (using at least one transition). Since the number of states in an automaton is finite, the emptiness problem is clearly decidable. In the following we will describe two different types of algorithms for the emptiness problem. The first one uses nested depth first search in order to find an above mentioned type of acceptance state. The second algorithm first constructs the strongly connected components of the automaton. Then the existence of the above mentioned type of state can be checked by observing the properties of the strongly connected components. In the context of the latter algorithm, the verification under fairness assumptions can be treated directly, *without* the need to employ the monitor automaton. This is a clear advantage since the usage of the monitor causes the memory (and time) usage of the emptiness check to be multiplied.

4.1. Nested depth first search approach

Theorem 12 showed that verification under fairness assumptions can be reduced to checking the emptiness of a language accepted by a Büchi automaton. The Büchi automaton in this case is, using the notation from Theorem 12, the product $(B^l \times B_{\mathrm{wf}}) \times B_{\overline{\varphi}}$ or $(B^l \times B_{\mathrm{sf}}) \times B_{\overline{\varphi}}$, i.e., the automaton with embedded monitor automata for either weak or strong fairness. Theorem 13 on the other hand shows that this problem of language emptiness can be decided by finding a cycle containing an accepting state. In [6] an efficient algorithm was presented for this purpose. We review it briefly here.

The algorithm is based on two nested depth first searches. The idea can be described as follows. The first search finds all the reachable accepting states and sorts them in post-order $f_1, \ldots, f_n$. In this order, the "nested search" is started from each of the states f_i and looks for cycles including f_i. The existence of such a cycle can be determined easily; a cycle exists if f_i is reachable from itself. It can be shown that both searches need to visit each state of the product automaton at most once, and thus the algorithm works in linear time in the size of the automaton mentioned above. The searches can also be interleaved, i.e., when the first accepting state f_1 is found, the first nested search is started. Thus the emptiness can be checked on-the-fly. A pseudo-code is given in Algorithm 14.

Algorithm 14. Find accepting cycles. Suppose $B = (S, \Sigma, \Delta, s0, F)$ is a Büchi automaton. The following algorithm reports the existence of an accepting cycle in B.

```
program 2dfs(B)
begin
    S0 := ∅;
    S1 := ∅;
    dfs(s0);
end.

procedure dfs(s)
begin
    S0 := S0 ∪ s;
    for each s' for which (s, a, s') ∈ Δ for some a ∈ Σ do
        if s' ∉ S0 then
            dfs(s');
    if s ∈ F then
        begin
            seed := s;
            ndfs(s);
        end;
end;
```

```
procedure ndfs(s)
begin
    S1 := S1 ∪ s;
    for each s' for which (s, a, s') ∈ Δ for some a ∈ Σ do
        begin
            if s' ∉ S1 then
                ndfs(s');
            else if s' = seed then
                report cycle;
        end;
end;
```

For proof of correctness, see [6].

□

4.2. Strongly connected component approach

Theorem 13 states that the emptiness of the language accepted by a Büchi automaton can be decided by checking if it contains a strongly connected component with at least one transition and one accepting state. Thus this is a solution to the model checking problem of Theorem 6. It turns out that the *fairness assumption*, as well, can be included in the *algorithm* that looks for the strongly connected components.

Lichtenstein and Pnueli [2] present an algorithm for deciding whether a strongly connected component in the reachability graph of a concurrent system includes a fair cycle. In the following, we give a brief description of the algorithm, which was originally described in a somewhat different setting. We do not discuss actually finding the strongly connected components, as Tarjan's algorithm [16] is the well-known standard solution for this.

Note that in Section 4.1, the Büchi automaton given as input to the nested search algorithm was the one with embedded monitor automaton. Here, it is important to note that the Büchi automaton given as input to the algorithms described in this section is, using the notation from Theorem 12, $B^l \times B_{\overline{\varphi}}$, i.e., without the monitor automaton.

In the following, by a strongly connected component of a Büchi automaton $B^l = (S, \Sigma, \Delta, s_0, F, E)$ we mean any subset of S that is strongly connected by transitions in Δ. Also, we call a strongly connected component *nontrivial* if it contains at least one transition and *maximal* if no state can be added to it so that it remains strongly connected. When meaning is clear from the context, we address different parts of the tuple that makes up a Büchi automaton without explicit reference to the automaton.

We review the algorithms separately for the cases of weak and strong fairness. In both cases, it turns out we can give simple conditions under which there exists a fair cycle in a given strongly connected component. In the case of weak fairness, this condition is exclusive. The use of this condition is more complex in the case of strong fairness.

4.2.1. Weak fairness

In the case of weak fairness, the condition can be employed in a straightforward manner. Assume we want to find out whether a Büchi automaton $B^l \times B_{\overline{\varphi}}$ contains a weakly fair cycle with an acceptance state. Let C be a nontrivial strongly connected component of $B^l \times B_{\overline{\varphi}}$ such that $F \cap C \neq \emptyset$. C contains an accepting weakly fair cycle if and only if the following holds for all B_i: if $i \in E(s)$ holds for all $s \in C$, then $\exists s, s' \in C : (s, a, s') \in \Delta$ for some $a \in \Sigma$ such that B_i takes part in the transition $(s, a, s') \in \Delta$.

Let us consider the correctness of the above condition. The backward implication is clear. Now suppose there is an automaton B_i such that $i \in E(s)$ for all $s \in C$ and none of whose transitions are included in C. Then in any infinite execution $\overline{s} = s_0 s_1 s_2 \ldots$, where $s_i \in C$ holds for all i, the automaton B_i always has an enabled transition. But as there is no transition $(s, a, s') \in \Delta$ of the automaton B_i such that $s, s' \in C$, B_i cannot take part in any transition of $\overline{s}$. Thus no weakly fair cycle can exist in C.

We have seen that C contains an accepting weakly fair cycle if and only if all the automata that are enabled in every state of C can execute some transition within C. And thus, $B^l \times B_{\overline{\varphi}}$ contains a weakly fair cycle if and only if the above condition holds for any one of its nontrivial strongly connected components that contain at least one accepting state. Of course, it is enough to go through all the maximal strongly connected components C, as these include all the cycles of their subsets.

The condition described above can clearly be checked by going through a strongly connected component once. This is described in algorithm 15.

Algorithm 15. Find accepting cycles assuming weak fairness. Suppose $B^l = (S, \Sigma, \Delta, s_0, F, E)$ is an enabledness labelled Büchi automaton and $C \subseteq S$ forms a strongly connected component. If C includes an accepting weakly fair cycle, the following algorithm reports it.

procedure *WF*(C)
begin
  if $\nexists s, s' \in C : (s, a, s') \in \Delta$ for some $a \in \Sigma$ or $C \cap F = \emptyset$ **then**
    report no cycle;
  else
    begin
      Ex := $\{i \mid \exists s, s' \in C : (s, a, s') \in \Delta$ for some $a \in \Sigma$
          such that B_i takes part in the transition $(s, a, s') \in \Delta\}$;
      En := $\{i \mid i \in E(s)$ for all $s \in C\}$;
      if $En \subseteq Ex$ **then**

report cycle;
end;
end;

The correctness of the algorithm follows from the above discussion.

□

4.2.2. Strong fairness

Let us now proceed to consider the case of strong fairness. Let $B^l \times B_{\overline{\varphi}}$ and C be as above. The condition now takes the following form: C contains a strongly fair accepting cycle if the following holds for all B_i: if $\exists s \in C$ such that $i \in E(s)$, then $\exists s, s' \in C : (s, a, s') \in \Delta$ for some $a \in \Sigma$ such that B_i takes part in the transition $(s, a, s') \in \Delta$.

This is quite clear, as assuming the condition holds, one can construct an infinite execution $\overline{s} = s_0 s_1 s_2 \ldots$, where $s_i \in C$ holds for all i, and where all the processes that become enabled at some point can execute their transition guaranteed to exist within C. Now consider the case when the condition does not hold. Then there must exist some set of states $R \subseteq C$, where some automaton B_i is enabled whose transitions are not included in C. It is clear that a strongly fair cycle in C cannot pass through any of these states. But we still cannot deduce that no strongly fair cycle exists, as there may be cycles avoiding these so called "bad" states. In algorithm 16, the idea is simply to remove the set R of bad states from C and process $C \setminus R$ recursively.

Algorithm 16. Find accepting cycles assuming strong fairness. Suppose $B^l = (S, \Sigma, \Delta, s_0, F, E)$ is an enabledness labelled Büchi automaton and $C \subseteq S$ forms a strongly connected component. If C includes an accepting strongly fair cycle, the following algorithm reports it.

procedure *SF*(*C*)
begin
if $\nexists s, s' \in C : (s, a, s') \in \Delta$ for some $a \in \Sigma$ or $C \cap F = \emptyset$ **then**
report no cycle;
else
begin
Ex := $\{i \mid \exists s, s' \in C : (s, a, s') \in \Delta$ for some $a \in \Sigma$
such that B_i takes part in the transition $(s, a, s') \in \Delta\}$;
En := $\{i \mid i \in E(s)$ for some $s \in C\}$;
if *En* $\subseteq$ *Ex* **then**
report cycle;
else
begin
R := $\{s \mid s \in C$ where $\exists i \in En \setminus Ex : i \in E(s)\}$;

```
                $C := C \setminus R$;
                for maximal strongly connected components $C'$ in $C$ do
                    if $C' \cap F \neq \emptyset$ then
                        $SF(C')$;
            end;
        end;
end;
```

First note that if C contains no transitions or no accepting state, obviously no accepting cycle exists. From the above discussion it is clear that if $En \subseteq Ex$ holds, there exists a strongly fair cycle. In both these cases the algorithm is correct.

Now consider the case when $En \subseteq Ex$ does not hold. As a fair cycle cannot pass through any of the states $s \in R$, it is enough to look at the cycles of $C \setminus R$. These are all included in the maximal strongly connected components of $C \setminus R$. From this it follows that deciding the existence of a strongly fair accepting cycle for all of these components decides also the existence of a strongly fair accepting cycle in C.

For each $C' \in C$, the recursion terminates when either the above condition holds for some strongly connected component or no transitions are left in the component.

□

5. COMPARISON OF THE METHODS

In this section, we perform some empirical tests [9] of the methods described in this paper. For sake of brevity, we call the method based on monitor automata and depth first search the 2DFS method and the method based on strongly connected components the SCC method. In the following, we first take a look at the theoretical performance of the two methods and then compare this with the empirical results. The empirical comparison is performed on two protocols for mutual exclusion.

5.1. Theoretical evaluation

There are two issues that are of interest to us in the performance of the compared methods. Clearly, one is the time complexity of the actual algorithm for model checking, i.e., in the case of 2DFS the nested depth first search (Algorithm 14) and in the case of SCC the algorithm used for finding the strongly connected components and the algorithm of Lichtenstein and Pnueli (Algorithms 15, 16).

The complexity of 2DFS in this sense is easily determined. The nested depth first search is linear in the size of the product automaton. The worst case is that both of the searches go through all of the automaton once.

The SCC method is a bit less straightforward. Finding strongly connected components, in the first place, is easy and can be done in linear time, e.g. with Tarjan's algorithm. Also, it is easy to see that the cycle finding algorithm employed in SCC is linear in the case of weak fairness. Whether the condition discussed in Section 4.2 is fulfilled for a strongly connected component C can be determined by going through the component once.

Now let us consider strong fairness. Let k be the number of the automata for which fairness is assumed and let n be the size of a strongly connected component to be checked for cycles. Determining if the condition holds and, if this is not the case, finding the bad states can easily be implemented in $O(n)$ time. On every level of the recursion, all the states that have out transitions with some of the k automata are removed as bad. Thus the depth of the recursion is bound by k. This makes the time complexity of the algorithm $O(nk)$.

The other issue concerning the overall performance is what effect the method has on the size of the product automaton. In the SCC method, the product automaton $B^l \times B_{\overline{\varphi}}$ forms the input of the algorithm as is, i.e., there is no extra factor. In the 2DFS method, monitor automata are embedded which multiply the product automaton with some factor. This comes from incorporating the monitor automata and the use of the construction from [12] for turning a generalized Büchi automaton into a normal one. In the case of weak fairness, this factor is linear in k. In the case of strong fairness, however, this factor is $O(2^k)$. In the empirical study, it is of interest to us to what extent these theoretical factors show up in real life problems. From above, it is clear that the relative performance of the methods mostly depends on the factor k.

5.2. The empirical study

Now let us proceed with the description of the empirical study. The mutual exclusion problem is stated as follows: make sure that no two processes execute commands from a critical section at the same time, and also make sure that any process can execute the code of the critical section eventually if it wants to. A mutual exclusion protocol is said to be *live* if the latter condition is fulfilled. We look at two mutual exclusion protocols, Peterson's and a simple solution employing a semaphore. Both of these generalize to the case of arbitrarily many processes, for all of which we assume fairness; thus we can evaluate the effects of the increasing parameter k. We give the results in the number of states encountered during verification and the time that the verification took. The tests were run on a Pentium computer with 256 MB of central memory.

Peterson's protocol is a well-known solution to the problem at hand. It employs a queuing system for the processes wanting to enter the critical section and is live assuming weak fairness. The results are displayed in Fig. 1 (upper part). Note that for the 2DFS method and 4 processes the figures are just lower bounds, as the program ran out of memory after 2×10^6 states.

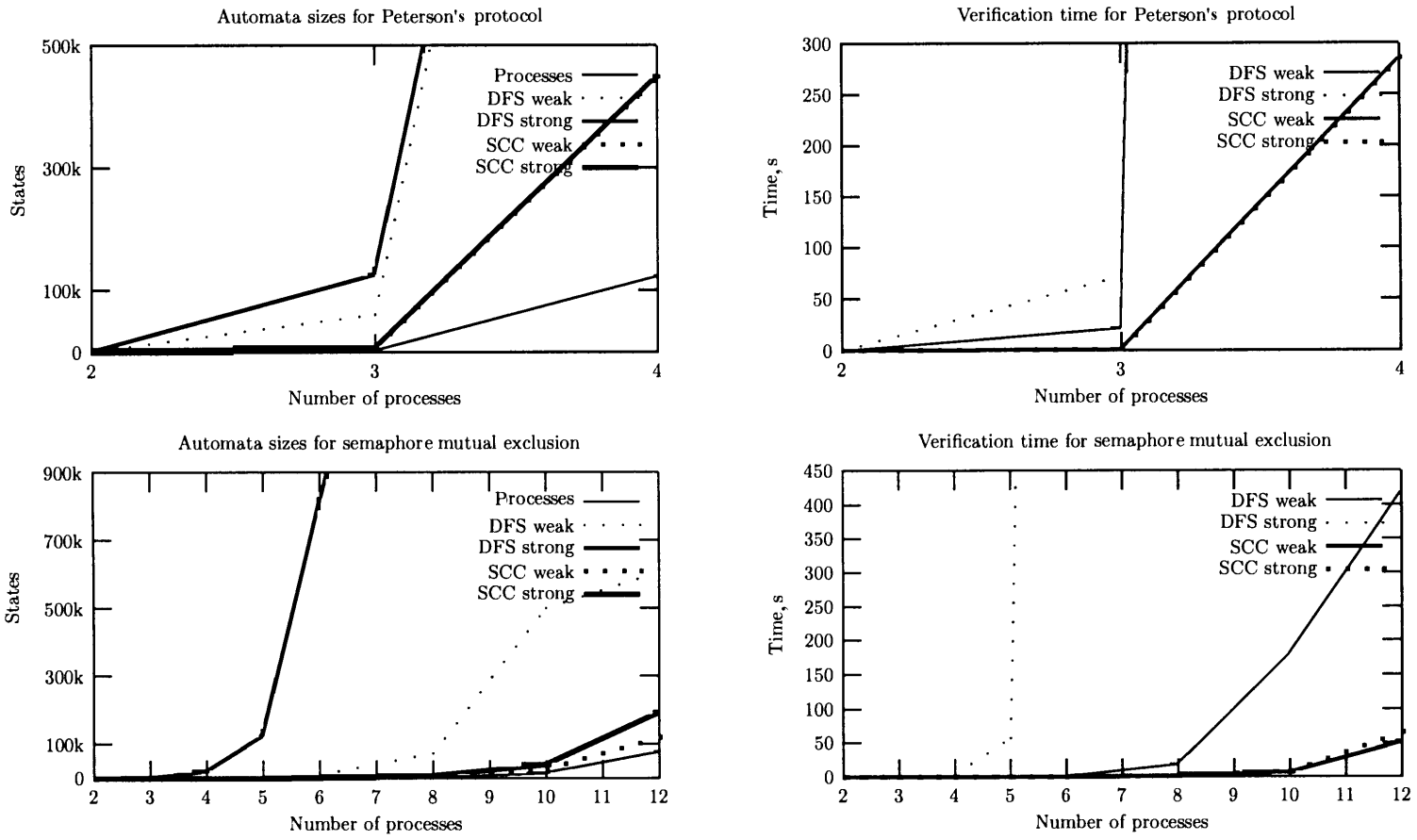

Fig. 1. The empirical results for Peterson's protocol and semaphore mutual exclusion.

Generally, the graphs show a very clear edge in performance for the SCC method; with the 2DFS method we could not solve the case of 4 processes. This is best explained by the high complexity of the protocol in itself: with 4 processes, it has over 10^5 states, which contributes to the high number of states in the product automaton of the 2DFS method. The verification times show this directly. One must note here that the extremely high time curve is mostly due to the paging involved. The worst case scenario seems realistic for 2DFS in the sense that the number of states is close to the theoretical maximum implied by the constructions used. One notes that the curves of the SCC method overlap here in the cases of weak and strong fairness. Thus in this case, no extra work needs to be done if one assumes strong fairness without actually needing it.

The next protocol we consider is a simple semaphore solution to the problem. The system consists of a number of processes which wait for their entry to the critical section at a semaphore. Weak fairness is not enough in this case as, waiting at the semaphore, the processes do not have any transitions enabled. Strong fairness guarantees the liveness of the protocol. Again, the results are shown in Fig. 1 (lower part). As the protocol is not live under the assumption of weak fairness, the figures show the number of states found and time spent before the first counterexample was found (except for the times of the SCC method for reasons discussed later). Again, the results for the case of 12 processes and weakly fair 2DFS and from 8 processes on and strongly fair 2DFS are only lower bounds, which explains the surprising lessening gradient. The sharp increase in time is again explained by paging.

In this case, SCC does not outperform 2DFS as clearly as with Peterson's protocol. The case of 10 processes could be found not live assuming weak fairness and the case of 6 processes verified using strong fairness. Because the system is essentially very simple, with only c. 77 000 states in the case of 12 processes, the insurmountable growth of the automaton in 2DFS takes place later than with Peterson's.

Let us then consider the performance of the SCC method. The times in the lower right graph in Fig. 1 and the case of weak fairness are the times it took to find all the counterexamples, i.e., all of the automaton was examined. This enables us to directly compare the time complexity of the $O(n)$ weak case and the $O(nk)$ strong case. Now consider the worst case scenario in the case of strong fairness. It will only take place when, for each strongly connected component to be checked, the recursion goes to its full depth k. As it may be that one removes the "bad" states from which there are transitions with several of the k automata, the recursion reaches depth k only when for each of the k automata the bad states are removed separately. Because of this, one would not expect the time to be multiplied quite with k, but still to be substantially worse than the linear weak case. Surprisingly, in our test runs, the performance of SCC is almost the same with both fairnesses. Thus, in the light of these experiments the factor k seems to be a theoretical worst case scenario.

6. DISCUSSION AND FUTURE WORK

We have described and performed empirical evaluation of two different methods for taking fairness assumptions into account in automata theoretic model checking. Both of the methods are known from the literature. We wanted to put together these methods, present them in a unified formal framework and to compare their usefulness in some problems.

Weak fairness is recognized as a practically applicable assumption. Indeed, widely used LTL model checkers such as SPIN [7] include verification under weak fairness as a standard option. Essentially, the method for weak fairness implemented in SPIN is the same as the 2DFS and monitor based method described in Section 3. As this paper also shows, the case of strong fairness is computationally harder. Thus, it is not surprising that this case is not so commonly implemented.

In the case of weak fairness, both approaches are rather efficient, though the SCC method performs better in both case studies. This is because of the extra states due to the monitor automaton and the construction of [12]. The case of strong fairness provides more distinction between the methods. The nondeterminism of the monitor automaton in the case of strong fairness causes an exponential blow-up of the product automaton; this happens as the number of automata for which fairness is assumed grows. This exponential growth showed to be insurmountable rather early in our experiments. On the other hand, the SCC method performed surprisingly well; this considering the fact that also in this case, theoretically one would have to pay a penalty of an added linear factor to the running time. This factor hardly showed in our experiments.

These results seem interesting, as the nested depth first search has been the main algorithm implemented in LTL model checking tools, such as SPIN. All in all, it would seem that strong fairness could be a more widely implemented option in automata theoretic verification.

Looking at the figure in Section 5, it is striking how differently the 2DFS and SCC methods perform in the case of strong fairness, this despite the fact that the same problem is solved in both cases. This proposes some future work: it would be interesting to know whether the strong fairness assumption can be somehow incorporated into nested depth first search, as it has been done in SPIN for weak fairness. Also, approaching the problem from another perspective, there might be ways to traverse and store the states of the product automaton in a more subtle manner. The authors wish to thank Bengt Jonsson for pointing out these ideas. Other future work could include a more thorough empirical evaluation of the SCC method, on some realistic communication protocol, for instance.

REFERENCES

1. Vardi, M. and Wolper, P. An automata-theoretic approach to automatic program verification. In *Proc. of 1st Annual IEEE Symp. on Logic in Computer Science,*

LICS'86 (Cambridge, Mass., June 1986). IEEE CS Press, Washington, DC, 1986, 322–331.
2. Lichtenstein, O. and Pnueli, A. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of 12th ACM Symposium on Principles of Programming Languages, POPL'85 (New Orleans, Louisiana, Jan. 1985)*. ACM Press, New York, 1985, 97–107.
3. Thayse, A. (ed.). *From Modal Logic to Deductive Databases: Introducing a Logic Based Approach to Artificial Intelligence*. Wiley, Chichester, 1989.
4. Walker, D. Automated analysis of mutual exclusion algorithms using CCS. *Form. Asp. Comput.*, 1989, **1**, 279–292.
5. Aggarwal, S., Courcoubetis, C. and Wolper, P. Adding liveness properties to coupled finite-state machines. *ACM Trans. Program. Lang. Syst.*, 1990, **12**, 303–339.
6. Courcoubetis, C., Vardi, M., Wolper, P. and Yannakakis, M. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1992, **1**, 275–288.
7. Holzmann, G. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 1997, **23**, 279–295.
8. Gerth, R., Peled, D., Vardi, M. and Wolper, P. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. of 15th IFIP WG6.1 Int. Symp. on Protocol Specification, Testing and Verification (Warsaw, June 1995)* (Dembinski, P. and Sredniawa, M., eds.). Chapman and Hall, London, 1995, 3–18. (IFIP Conference Proceedings, **38**.)
9. Malinen, T. Taking fairness into account in automata theoretic verification algorithms of concurrent systems (in Finnish). Master's thesis, Department of Computer Science, University of Helsinki, October 2001.
10. Büchi, J. On a decision method in restricted second-order arithmetic. In *Proc. of 1st Int. Congr. on Logic, Methodology and Philosophy of Science (Stanford, Calif., 1960)* (Nagel, E., ed.). Stanford University Press, Stanford, Calif., 1962, 1–12.
11. Thomas, W. Automata on infinite objects. In *Handbook on Theoretical Computer Science, Vol. A*: *Algorithms and Complexity* (van Leeuwen, J., ed.). Elsevier, Amsterdam, 1990, 133–192.
12. Choueka, Y. Theories of automata on omega-tapes: a simplified approach. *J. Comput. Syst. Sci.*, 1974, **8**, 117–141.
13. Sistla, A. P., Vardi, M. T. and Wolper P. The complementation problem for Büchi automata with appplications to temporal logic. *Theor. Comput. Sci.*, 1987, **49**, 217–237.
14. Francez, N. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1986.
15. Manna, Z. and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.
16. Tarjan, R. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1972, **1**, 146–160.

Õiglus automaaditeoreetilises mudelkontrollis

Tuomo Malinen ja Matti Luukkainen

On kirjeldatud ja empiiriliselt hinnatud kaht meetodit õigluseelduste arvestamiseks automaaditeoreetilises mudelkontrollis. Mõlemad meetodid on kirjandusest tuntud, kuigi sel kujul pole neid esitatud. Üks meetoditest põhineb õigluseelduse kaasamisel süsteemimudelisse; teisel juhul arvestatakse õigluseeldust algoritmiliselt. Artikli eesmärk on panna meetodid kokku, esitada need unifitseeritud formaalses raamistikus ning võrrelda nende suhtelist tõhusust mõne näite najal.