On-the-fly Emptiness Check of Transition-based Streett Automata

Alexandre Duret-Lutz[1], Denis Poitrenaud[2], and Jean-Michel Couvreur[3]

[1] EPITA Research and Development Laboratory (LRDE)
[2] Laboratoire d'Informatique de Paris 6 (LIP6)
[3] Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)

Abstract. In the automata theoretic approach to model checking, checking a state-space S against a linear-time property φ can be done in $\mathrm{O}(|S| \times 2^{\mathrm{O}(|\varphi|)})$ time. When model checking under n strong fairness hypotheses expressed as a Generalized Büchi automaton, this complexity becomes $\mathrm{O}(|S| \times 2^{\mathrm{O}(|\varphi|+n)})$. Here we describe an algorithm to check the emptiness of Streett automata, which allows model checking under n strong fairness hypotheses in $\mathrm{O}(|S| \times 2^{\mathrm{O}(|\varphi|)} \times n)$. We focus on transition-based Streett automata, because it allows us to express strong fairness hypotheses by injecting Streett acceptance conditions into the state-space without any blowup.

1 Introduction

The Automata Theoretic Approach to Model Checking [29, 28] is a way to check that a model M verifies some property expressed as a temporal logic formula φ, in other words: to check whether $M \models \varphi$. This verification is achieved in four steps, using automata over infinite words (ω-automata):

1. Computation of the state space of M. This graph can be seen as an ω-automaton A_M whose language $\mathscr{L}(A_M)$ is the set of all possible executions of M.
2. Translation of the temporal property φ into an ω-automaton $A_{\neg\varphi}$ whose language, $\mathscr{L}(A_{\neg\varphi})$, is the set of all executions that would invalidate φ.
3. Synchronized product of these two objects. This constructs an automaton $A_M \otimes A_{\neg\varphi}$ whose language is $\mathscr{L}(A_M) \cap \mathscr{L}(A_{\neg\varphi})$: the set of executions of the model M that invalidate the temporal property φ.
4. Emptiness check of this product. This operation tells whether $A_M \otimes A_{\neg\varphi}$ accepts an infinite word (a counterexample). The model M verifies φ iff $\mathscr{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$.

On-the-fly algorithms. In practice the above steps are usually tightly tied by the implementation, due to transversal optimizations that forbid a sequential approach. One such optimization is the *on-the-fly model checking*, where the computation of the product, state space, and formula automaton are all driven by the progression of the emptiness check procedure: nothing is computed before it is required.

Being able to work on-the-fly has three practical advantages:

– Large parts of A_M may not need to be built because of the constraints of $A_{\neg\varphi}$.
– The emptiness check may find a counterexample without exploring (and thus constructing) the entire synchronized product.

– To save memory we can throw away states that have been constructed but are not actually needed. We would rebuild them later should they be needed again. [13]

From an implementation standpoint *on-the-fly model checking* puts requirements on the interface of the automata representing the product, the state graph, and the formula. For instance the interface used in Spot [8] amounts to two functions: one to obtain the initial state of the automata, another to get the successors of a given state. It is common to say that an emptiness check *is* on-the-fly when it is *compatible* with such an interface. For instance Kosaraju's algorithm [2, §23.5] for computing strongly connected components (SCC) will not work on-the-fly because it has to know the entire graph to transpose it. The algorithms of Tarjan [25] and Dijkstra [5, 6] are more suited to compute SCCs on-the-fly, because they perform a single depth-first search.

Fairness hypotheses [10] is a way to restrict the verification to a subset of "fair" executions of the model. For instance if we have a model of two concurrent processes running on the same host, we might want to assume that the scheduler is fair and that both processes will get slices of CPU-time infinitely often. When considering all the possible executions of the model, this hypothesis amounts to discarding all executions in which a process is stuck.

Transition-based Büchi and Streett automata. We shall consider two kinds of ω-automata that are expressively equivalent: Büchi automata and Streett automata. Büchi automata are more commonly used because there exist simple translations from LTL formulæ to Büchi automata and there exist many emptiness check algorithms for these automata [4]. Readers familiar with Büchi automata might be surprised that we use transition-based acceptance conditions rather than state-based ones. As noted by several authors [19, 3, 11, 12, 4, 26] this allows LTL formulæ to be translated into smaller automata, and for our purpose it will be useful to show why *weak* (resp. *strong*) *fairness hypotheses* can be added to a Büchi (resp. Streett) automaton without any blowup.

Streett Automata can also be used as intermediate steps in some methods to complement Büchi automata [27]. For instance in the automata theoretic approach to model checking, we could want to express a property P to verify, not as an LTL formula, but as a (more expressive) Büchi automaton A_P (or equivalently, an ω-regular expression). To ensure that $M \models A_P$ we should check that $\mathscr{L}(A_M \otimes \neg A_P) = \emptyset$. One way to compute $\neg A_P$ is to use Safra's construction [21] to construct a Streett automaton, and then convert this Streett automaton back into a Büchi automaton.

Our objective is to introduce an on-the-fly emptiness check for transition-based Streett automata, in order to efficiently verify linear-time properties under strong fairness hypotheses, or simply to check the emptiness of $A_M \otimes \neg A_P$ without the cost of converting $\neg A_p$ into a Büchi automaton. Existing emptiness checks for Streett automata [20, 15] share the same asymptotic complexity, but are state-based and will not work on-the-fly.

Outline. Section 2 briefly reviews LTL and transition-based Büchi automata. Section 3 then introduces fairness hypotheses and Streett automata. We recall that weak fairness hypotheses are free and show that strong fairness hypotheses are less costly to express with Streett automata. Finally section 4 gives an on-the-fly algorithm to check the emptiness of a Streett automaton in a way that is only linearly slower (in the number of acceptance conditions) than the emptiness check of a Büchi automaton.

2 Background

2.1 Linear-time Temporal Logic (LTL)

An LTL formula is constructed from a set AP of atomic propositions, the usual boolean operators ($\neg$, $\vee$, $\wedge$, $\rightarrow$) and some temporal operators: X (next), U (until), F (eventually), G (globally). An LTL formula can express a property on the execution of the system to be checked. Because we focus on fairness properties we shall not be concerned with the full semantics of LTL [1, 18], it is enough to describe the following two idioms:

- G F p means that property p is true infinitely often (i.e., at any instant of the execution you can always find a later instant so that p is true),
- F G p means that property p is eventually true continuously (i.e., at some instant in the future p will stay true for the remaining of the execution).

The size $|\varphi|$ of an LTL formula φ is its number of operators plus atomic propositions.

2.2 Büchi Automata

Definition 1 *(TGBA) A* Transition-based Generalized Büchi Automaton [12] *over Σ is a Büchi automaton with labels and generalized acceptance conditions on transitions. It can be defined as a tuple $A = \langle \Sigma, \mathcal{Q}, q^0, \delta, \mathcal{F} \rangle$ where Σ is an alphabet, $\mathcal{Q}$ is a finite set of states, $q^0 \in \mathcal{Q}$ is a distinguished initial state, $\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the (non-deterministic) transition relation, $\mathcal{F} \subseteq 2^\delta$ is a set of sets of accepting transitions.*

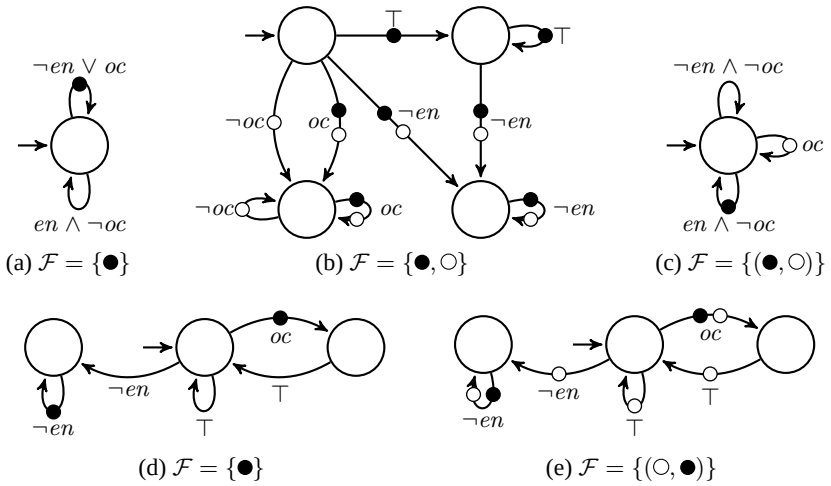
Graphically we represent the elements of $\mathcal{F}$ (which we call *acceptance conditions*) as small circles such as ● or ○ on Fig. 1a, 1b and 1d. We will also merge into a single transition all transitions between two states with identical acceptance conditions, as if the transition relation was actually in $\mathcal{Q} \times 2^\Sigma \times \mathcal{Q}$.[4]

For the purpose of model checking we have AP equal to the set of all atomic propositions that can characterize a configuration, and we use these automata with $\Sigma = 2^{AP}$ (i.e., each configuration of the system can be mapped into a letter of Σ). Graphically, with the aforementioned merging of transitions, it is therefore equivalent to label the transitions of the automata by propositional formulæ over AP.

An infinite word $\sigma = \sigma(0)\sigma(1)\cdots$ over the alphabet Σ is accepted by A if there exists an infinite sequence $\rho = (q_0, l_0, q_1)(q_1, l_1, q_2)\ldots$ of transitions of δ, starting at $q_0 = q^0$, and such that $\forall i \geqslant 0, \sigma(i) = l_i$, and $\forall f \in \mathcal{F}, \forall i \geqslant 0, \exists j \geqslant i, \rho(j) \in f$. That is, each letter of the word is recognized, and ρ traverses each acceptance condition infinitely often.

Given two TGBAs A and B, the synchronous product of A and B, denoted $A \otimes B$ is a TGBA that accepts only the words that are accepted by A and B. If we denote $|A|_s$ the number of accessible states of A, we have $|A \otimes B|_s \leq |A|_s \times |B|_s$. If we denote $|A|_t$ the number of transitions of A, we always have $|A|_t \leq |A|_s^2 \times |\Sigma|$. Also $|A \otimes B|_t \leq (|A|_s \times |B|_s)^2 \times |\Sigma| \leq |A|_t \times |B|_t$. Finally if a TGBA C has only one state and is deterministic, then $|A \otimes C|_s \leq |A|_s$ and $|A \otimes C|_t \leq |A|_t$.

[4] This optimization is pretty common in implementations; we only use it to simplify figures.



(a) $\mathcal{F} = \{\bullet\}$ (b) $\mathcal{F} = \{\bullet, \circ\}$ (c) $\mathcal{F} = \{(\bullet, \circ)\}$

(d) $\mathcal{F} = \{\bullet\}$ (e) $\mathcal{F} = \{(\circ, \bullet)\}$

Fig. 1: (a) A TGBA equivalent to the LTL formula $\mathsf{GF}((\neg en) \vee oc)$; (b),(d) two TGBAs equivalent to $\varphi = (\mathsf{GF}\, en \rightarrow \mathsf{GF}\, oc)$; (c),(e) two TSAs equivalent to φ. $\top$ denotes the *true* value.

3 Coping with Fairness Hypotheses

Fairness hypotheses are a way to filter out certain behaviors of the model that are deemed irrelevant. For instance when modeling a communication between two processes over a lossy channel, we might want to assume that any message will eventually reach its destination after a finite number of retransmissions. Although there is one behavior of the model in which the retransmitted message is always lost, we may want to ignore this possibility during verification.

3.1 Weak and Strong Fairness

Let us give a definition of fairness involving a pair of events en and oc in a model M. An event could be the progress of some process, the execution of a particular instruction of the model, or even the fact that an instruction is enabled (i.e., could be executed).

Definition 2 *(Unconditional fairness) An event oc is* unconditionally fair *if it will happen infinitely often, i.e., if $M \models \mathsf{GF}\, oc$.*

Definition 3 *(Weak fairness) A pair of events (en, oc) is* weakly fair *if whenever en occurs continuously, then oc will occur infinitely often: $M \models (\mathsf{FG}\, en \rightarrow \mathsf{GF}\, oc)$.*

Because we have $\mathsf{FG}\, en \rightarrow \mathsf{GF}\, oc \equiv \mathsf{GF}((\neg en) \vee oc)$ weak fairness can be handled like unconditional fairness.

Fig. 1a shows an example of a 1-state TGBA recognizing $\mathsf{GF}((\neg en) \vee oc)$. This TGBA is deterministic: for any configuration given by a set of truth values of en and oc, there is only one transition that can be followed. In fact, any formula of the form $\bigwedge_{i=1}^{n}(\mathsf{FG}\, en_i \rightarrow \mathsf{GF}\, oc_i)$, representing a combination of n weak (or unconditional)

fairness hypotheses, can be translated into a 1-state deterministic TGBA with 2^n transitions. Note that this "1-state determinism" property holds both because we are considering *generalized* automata and *transition-based* acceptance conditions, it would not not hold for *state-based* acceptance conditions.

Definition 4 *(Strong fairness) A pair of events* (en, oc) *is* strongly fair *if whenever* en *occurs infinitely often, then* oc *will occur infinitely often:* $M \models (\mathsf{G}\,\mathsf{F}\,en \rightarrow \mathsf{G}\,\mathsf{F}\,oc)$.

Fig. 1b and 1d show two TGBAs corresponding to the formula $\mathsf{G}\,\mathsf{F}\,en \rightarrow \mathsf{G}\,\mathsf{F}\,oc$. The first, bigger automaton is produced by LTL-to-Büchi translation algorithms such those of Couvreur [3] or Tauriainen [26]. The smaller one is a TGBA adaptation of an automaton shown by Kesten et al. [14]; we do not know of any general LTL-to-Büchi translation algorithm that would produce this automaton. Attempts to construct automata for conjunctions of strong fairness hypotheses, i.e. formulæ of the form $\bigwedge_{i=1}^{n}(\mathsf{G}\,\mathsf{F}\,en_i \rightarrow \mathsf{G}\,\mathsf{F}\,oc_i)$, will lead to a nondeterministic automaton that has either $3^n + 1$ or 3^n states depending on whether we base the construction on Fig. 1b or 1d. These automata have $2^{\mathrm{O}(n)}$ transitions.

3.2 Fairness in the Automata Theoretic Approach

Given a model M and an LTL formula φ, we can check whether $M \models \varphi$ by checking whether the automaton $A_M \otimes A_{\neg\varphi}$ accepts any infinite word (such a word would be a counterexample). Because $|A_{\neg\varphi}|_t = 2^{\mathrm{O}(|\varphi|)}$, we have $|A_M \otimes A_{\neg\varphi}|_t \leq |A_M|_t \times 2^{\mathrm{O}(|\varphi|)}$. Checking the emptiness of a TGBA can be done in linear time with respect to its size, regardless of the number of acceptance conditions [4], so the whole verification process requires $\mathrm{O}(|A_M|_t \times 2^{\mathrm{O}(|\varphi|)})$ time.

Verifying φ under some fairness hypothesis represented as an LTL formula ψ amounts to checking whether $M \models (\psi \rightarrow \varphi)$, i.e., φ should hold only for the runs where ψ also holds. We can see that $A_M \otimes A_{\neg(\psi\rightarrow\varphi)} = A_M \otimes A_{\psi\wedge\neg\varphi} = A_M \otimes A_\psi \otimes A_{\neg\varphi}$. In other words, a fairness hypothesis could be represented by just an extra synchronized product before doing the emptiness check.

Weak fairness. We have seen that n weak fairness hypotheses can be represented by a 1-state deterministic TGBA. This means that the operation $A_M \otimes A_\psi$ is basically free: it will not add new states to those of A_M. In practice each transition of A_M would be labelled during its on-the-fly construction with the acceptance conditions of A_ψ. Model checking under n week fairness hypotheses is therefore independent of n[5] and requires $\mathrm{O}(|A_M|_t \times 2^{\mathrm{O}(|\varphi|)})$ time.

Strong fairness. Model checking under n strong fairness hypotheses is costly with Büchi automata: we have seen that these n hypotheses can be represented by a TGBA with $2^{\mathrm{O}(n)}$ transitions, the verification therefore requires $\mathrm{O}(|A_M|_t \times 2^{\mathrm{O}(|\varphi|+n)})$ time.

3.3 Streett Automata

Definition 5 *(TSA) A* Transition-based Streett Automaton *is a kind of TGBA in which acceptance conditions are paired. It can be also be defined as a tuple* $A = \langle \Sigma, \mathcal{Q}, q^0, \delta, \mathcal{F}\rangle$

[5] This is because we assume that we are using a generalized emptiness check [4].

where $\mathcal{F} = \{(l_1, u_1), (l_2, u_2), \ldots, (l_r, u_r)\}$ *is a set of pairs of acceptance conditions with* $l_i \subseteq \delta$ *and* $u_i \subseteq \delta$.

The difference between TSA and TGBA lies in the interpretation of $\mathcal{F}$. An infinite word σ over the alphabet Σ is accepted by A if there exists an infinite sequence $\rho = (q_0, l_0, q_1)(q_1, l_1, q_2)\ldots$ of transitions of δ, starting at $q_0 = q^0$, and such that $\forall i \geqslant 0, \sigma(i) = l_i$, and $\forall (l, u) \in \mathcal{F}, (\forall i \geqslant 0, \exists j \geqslant i, \rho(j) \in l) \implies (\forall i \geqslant 0, \exists j \geqslant i, \rho(i) \in u)$. That is, each letter of the word is recognized, and for each pair (l, u) of acceptance conditions, if ρ encounters l infinitely often, then it encounters u infinitely often.

Given two TSA A and B, it is also possible to define their synchronous product $A \otimes B$ such that $|A \otimes B| = \mathrm{O}(|A| \times |B|)$ and $\mathscr{L}(A \otimes B) = \mathscr{L}(A) \cap \mathscr{L}(B)$.

Büchi and Streett automata are known to be expressively equivalent [21]. Obviously a TGBA with acceptance conditions $\mathcal{F} = \{u_1, u_2, \ldots, u_n\}$ can be translated into an equivalent TSA without changing its structure: we simply use the acceptance conditions $\mathcal{F}' = \{(\mathcal{Q}, u_1), \ldots, (\mathcal{Q}, u_n)\}$. For instance Fig. 1e shows the TSA resulting from this rewriting applied to the TGBA of Fig. 1d.

The converse operation, translating Streett automata to Büchi, induces an exponential blowup of the number of states [22]. For instance Löding [17] shows how to translate a state-based Streett automaton of $|\mathcal{Q}|$ states and n pairs of acceptance conditions into a state-based Büchi automaton with $|\mathcal{Q}| \times (4^n - 3^n + 2)$ states (and 1 acceptance condition). The following construction shows how to translate a TSA of $|\mathcal{Q}|$ states and n pairs acceptance conditions of into a TGBA of $|\mathcal{Q}| \times (2^n + 1)$ states and n acceptance conditions. (The same construction could be achieved for state-based automata: here the gain is only due to the use of generalized acceptance conditions.)

Given a TSA $A = \langle \Sigma, \mathcal{Q}, q^0, \mathcal{F}, \delta \rangle$ with $\mathcal{F} = \{(l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n)\}$, let $N = \{1, 2, \ldots, n\}$, and for any $(S, t) \in 2^N \times \delta$ let $\mathit{pending}(S, t) = (S \cup \{i \in N \mid t \in l_i\}) \backslash \{i \in N \mid t \in u_i\}$. Now define the TGBA $A' = \langle \Sigma, \mathcal{Q}', q^0, \delta', \mathcal{F}' \rangle$ where $\mathcal{Q}' = \mathcal{Q} \cup (\mathcal{Q} \times 2^N)$, $\delta' = \delta \cup \{(s, g, (d, \emptyset)) \mid (s, g, d) \in \delta\} \cup \{((s, S), g, (d, \mathit{pending}(S, (s, g, d))) \mid (s, g, d) \in \delta, S \in 2^N\}$, and $\mathcal{F}' = \{f_i \mid i \in 2^N\}$ with $f_i = \{((s, S), l, (d, D) \in \delta' \mid N \setminus S = i\}$. Then $\mathscr{L}(A) = \mathscr{L}(A')$.

The justification behind this construction is that any run accepted by a Streett automaton can be split in two parts: a finite prefix, where any transition can occur, followed by a infinite suffix where it is guaranteed that any transition in l_i will be eventually followed by a transitions in u_i. The original TGBA is therefore cloned $2^n + 1$ times to construct the corresponding TSA. The first clone, using $\mathcal{Q}$ and δ, is where the prefix is read. From there the automaton can non-deterministically switch to the clone that is using states in $\mathcal{Q} \times \{\emptyset\}$. From now on the automaton has to remember which u_i it has to expect: this is the purpose of the extra set added to the state. An automaton is in state (s, S) that follows a transition in l_i will therefore reach state $(s, S \cup \{i\})$, and conversely, following a transition in u_i will reach state $(s, S \setminus \{i\})$. The function $\mathit{pending}(S, t)$ defined above computes those pending u_is. The acceptance conditions are defined to complement the set of pending u_is, to be sure they are eventually fulfilled.

3.4 Strong Fairness with Streett Automata

The TSA of Fig. 1e is however not the most compact way to translate a strong fairness formula: Fig. 1c shows how it can be done with a 1-state deterministic TSA.

Actually any LTL formula $\bigwedge_{i=1}^{n} \mathsf{GF}\, en \rightarrow \mathsf{GF}\, oc$ representing n strong fairness hypotheses can be translated into a 1-state deterministic TSA with n pairs of acceptance conditions and 4^n transitions. It is the TSA $A = \langle 2^{AP}, \{q\}, q, \delta, \mathcal{F}\rangle$ where $AP = \{oc_1, oc_2, \ldots, oc_n, en_1, en_2, \ldots en_n\}$, $\delta = \{\langle q, E, q\rangle \mid E \in 2^{AP}\}$, and $\mathcal{F} = \{(l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n)\}$ with $l_i = \{(q, E, q) \in \delta \mid en_i \in E\}$ and $u_i = \{(q, E, q) \in \delta \mid oc_i \in E\}$. Again this "1-state determinism" would not hold for state-based Streett acceptance condition.

Combining this automaton with the construction of section 3.3, we can represent n strong fairness hypotheses using a TGBA of $2^n + 1$ states (and $4^n(2^n + 1)$ transitions). This is better than the TGBA of 3^n states presented in section 3.1, but the complexity of the verification would remain in $\mathrm{O}(|A_M|_t \times 2^{\mathrm{O}(|\varphi|+n)})$ time.

As when model checking under weak fairness hypotheses, the Streett acceptance conditions representing strong fairness hypotheses can be injected in the automaton A_M during its on-the-fly generation: any transition of A_M labelled by $E \in 2^{AP}$ receives the acceptance conditions $\alpha(E)$. The verification under n strong fairness hypotheses amounts to checking the emptiness of a TSA of size $\mathrm{O}(|A_M|_t \times 2^{\mathrm{O}(|\varphi|)})$, with n pairs of acceptance conditions.

We now show how to check this TSA emptiness in $\mathrm{O}(|A_M|_t \times n \times 2^{\mathrm{O}(|\varphi|)})$ time by adapting an algorithm by Couvreur [3, 4] that was originally designed for the emptiness check of TGBA.

4 Emptiness Check for Streett Automata

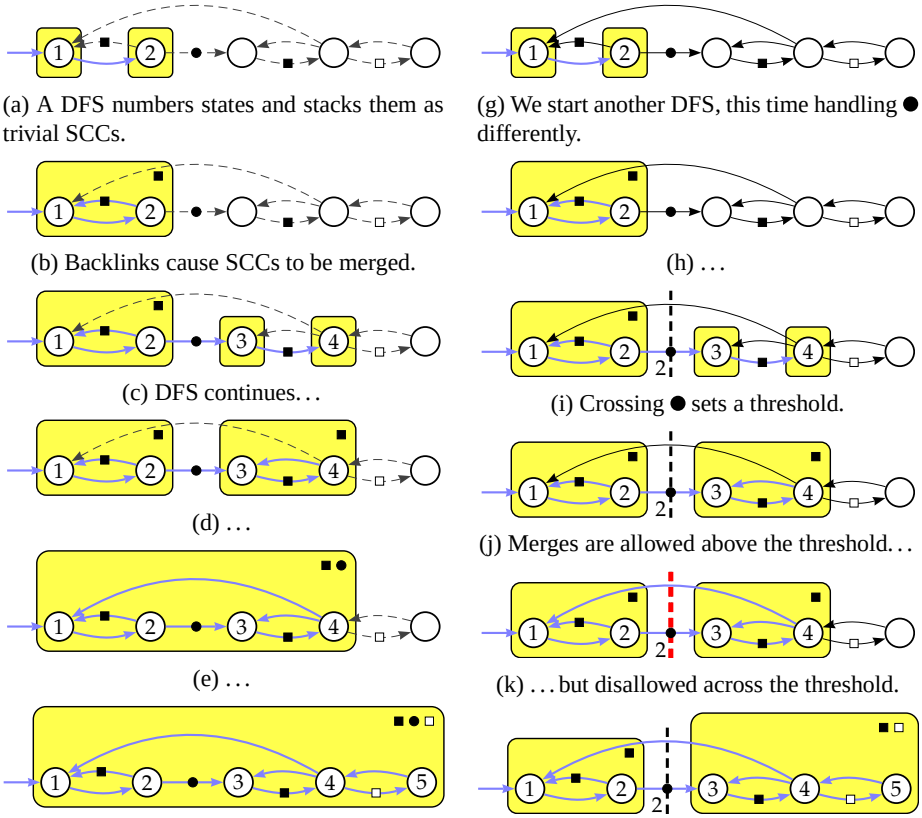
The behavior of the algorithm is illustrated on Fig. 2 on a TSA with 2 pairs of acceptance conditions: $(\bullet, \circ)$ and $(\blacksquare, \square)$. We are looking for runs that visit $\circ$ (resp. $\square$) infinitely often if they visit $\bullet$ (resp. $\blacksquare$) infinitely often.

As its older brother (for TGBA [3, 4]) this algorithm performs a DFS to discover strongly connected components (SCC). Each SCC is labelled with the set of acceptance conditions that can be found on its edges, and will stop as soon as it finds an SCC whose label verifies $(\bullet \rightarrow \circ) \wedge (\blacksquare \rightarrow \square)$. Figures 2a–2f show the first steps until a terminal SCC (i.e. with no outgoing transition) is found. Let us denote $\mathcal{F} = \{(l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n)\}$ the set of acceptance conditions of the Streett automaton, and $acc \subseteq \mathcal{F}$ the set of acceptance conditions of the terminal SCC encountered. When such a terminal SCC is found we can be in one of the three following cases.

1. Either the SCC is trivial (i.e. has no loops): it cannot be accepting and all its states can be ignored from now on.
2. Or the SCC is accepting: $\forall i, l_i \in acc \implies u_i \in acc$.
   In that case the algorithm terminates and reports the existence of an accepting run. It is better to check this condition any time a non-trivial SCC is formed, not only for terminal SCC: this gives the algorithm more chance to terminate early.
3. Or $\exists i, l_i \in acc \wedge u_i \notin acc$.
   In that case we cannot state whether the SCC is accepting or not. Maybe it contains an accepting run that does not use any transition of l_i. Fig. 2f is an instance of this case: $\mathcal{F} = \{(\bullet, \circ), (\blacksquare, \square)\}$ and $acc = \{\bullet, \blacksquare, \square\}$ so the algorithm cannot conclude immediately.

To solve third case, the algorithm will revisit the whole SCC, but avoiding transitions t such that $\exists i, t \in l_i \wedge u_i \notin acc$. Practically, we define the set $avoid = \{l_i \in acc \mid u_i \notin acc\}$ of l_i that cannot be satisfied, all the states from the SCC are removed from the hash table of visited states, and the algorithm makes another DFS with the following changes:

- amongst the outgoing transitions of a state, those who carry acceptance condition of $avoid$ are visited last
- crossing a transition labelled by an avoided acceptance condition sets up a threshold (denoted by a dashed vertical line on Fig. 2i)
- if a transition going out from a SCC goes back to another SCC in the search stack, then the two SCC will be merged only if the two SCC are behind the last threshold set. Fig. 2j shows one case where merging has been allowed, while Fig 2k shows a forbidden attempt to merge two SCCs.



(a) A DFS numbers states and stacks them as trivial SCCs.

(b) Backlinks cause SCCs to be merged.

(c) DFS continues...

(d) ...

(e) ...

(f) This terminal SCC could hide an accepting run that visits ● only finitely often.

(g) We start another DFS, this time handling ● differently.

(h) ...

(i) Crossing ● sets a threshold.

(j) Merges are allowed above the threshold...

(k) ...but disallowed across the threshold.

(l) The right SCC is accepting; we stop.

Fig. 2: Running the emptiness check on a TSA with $\mathcal{F} = \{(●, ○), (■, □)\}$.

This new visit will construct smaller SCCs instead of the original terminal SCC. The only way to merge these smaller SCCs would be to accept a cycle using a transition from an acceptance condition (of $avoid$) that cannot be satisfied. For each of these smaller SCCs we can then decide whether they are trivial, accepting, or if they contain acceptance conditions (not already listed in $avoid$) that cannot be satisfied. In the latter case $avoid$ is augmented and the process is repeated. This recursion cannot exceed $|\mathcal{F}|$ levels since we complete $avoid$ at each step with at least one pair of $\mathcal{F}$.

Compared to the original emptiness check for TGBA that visits each state and transitions only once, this variant will in the worst case visit each state and transitions $|\mathcal{F}| + 1$ times. On a TSA A this algorithm therefore works in $\mathrm{O}(|A|_t \times |\mathcal{F}|)$ time.

Relation to other algorithms. The basic idea of using strongly connected components to check strong fairness is old [16, 9], and has been declined in a few algorithms to

```
1  Input: A Streett automaton A = ⟨Σ, Q, q⁰, δ, F⟩
2  Output: ⊤ iff 𝓛(A) = ∅
3  Data: SCC: stack of
         ⟨state ∈ Q, root ∈ ℕ, la ⊆ F, acc ⊆ F, rem ⊆ Q, succ ⊆ δ, fsucc ⊆ δ⟩
         H: map of Q ↦ ℕ
         avoid: stack of ⟨root ∈ ℕ, acc ⊆ F⟩
         min: stack of ℕ
         max ← 0
4  begin
5      min.push(0)
6      avoid.push(⟨1, ∅⟩)
7      DFSpush(∅, q⁰)
8      while ¬SCC.empty() do
9          if SCC.top().succ = ∅ then
10             if SCC.top().fsucc ≠ ∅ then
11                 swap(SCC.top().succ, SCC.top().fsucc)
12                 min.push(max)
13             else
14                 DFSpop()
15         else
16             pick one ⟨s, e, d⟩ off SCC.top().succ
17             a ← {f ∈ F | (s, e, d) ∈ f}
18             if d ∉ H then
19                 DFSpush(a, d)
20             else if H[d] > min.top() then
21                 merge(a, H[d])
22                 acc ← SCC.top().acc
23                 if ∀⟨l, u⟩ ∈ F, (l ∈ acc) ⟹ (u ∈ acc) then return ⊥
24     return ⊤
25 end
```

Fig. 3: Emptiness check of a Streett automaton (continued on next page).

```
26 DFSpush(a ⊆ F, q ∈ Q)
27     max ← max + 1
28     H[q] ← max
29     SCC.push(⟨q, max, a, ∅, ∅, {⟨s, l, a, d⟩ ∈ δ | s = q, a ∩ avoid.top().acc = ∅},
                                  {⟨s, l, a, d⟩ ∈ δ | s = q, a ∩ avoid.top().acc ≠ ∅}⟩)
30 end
31 DFSpop()
32     ⟨q, n, la, acc, rem, _, _⟩ ← SCC.pop()
33     max ← n − 1
34     if n ≤ min.top() then
35         min.pop()
36     old_avoid ← avoid.top().acc
37     if n = avoid.top().root then
38         avoid.pop()
39     new_avoid ← old_avoid ∪ {l | ⟨l, u⟩ ∈ F, l ∩ acc ≠ ∅, u ∩ acc = ∅}
40     if new_avoid ≠ old_avoid then
41         foreach s ∈ rem do
42             delete H[s]
43         avoid.push(n, new_avoid)
44         DFSpush(la, q)
45     else
46         foreach s ∈ rem do
47             H[s] ← 0
48 end
49 merge(a ⊆ F, t ∈ ℕ)
50     r ← ∅
51     s ← ∅
52     f ← ∅
53     while t < SCC.top().root do
54         a ← a ∪ SCC.top().acc ∪ SCC.top().la
55         r ← r ∪ SCC.top().rem ∪ SCC.top().state
56         s ← s ∪ SCC.top().succ
57         f ← f ∪ SCC.top().fsucc
58         SCC.pop()
59     SCC.top().acc ← SCC.top().acc ∪ a
60     SCC.top().rem ← SCC.top().rem ∪ r
61     SCC.top().succ ← SCC.top().succ ∪ s
62     SCC.top().fsucc ← SCC.top().fsucc ∪ f
63 end
```

Fig. 3: (Continued from previous page.)

check the emptiness of (state-based) Streett automata [20, 15]. But these algorithms modify the graph before visiting it again, hindering on-the-fly computations.

At a high level, our algorithm is close to the one presented by Latvala and Heljanko [15], who suggests using any algorithm to compute SCCs. However we have more than implementation detail differences. Our algorithm is targeted to transition-based acceptance conditions, actually shows how to make the emptiness check on-the-fly, and uses two tricks that are dependent on the algorithm used to compute SCC. As mentioned in the introduction, there exists two similar algorithms to compute SCCs on-the-fly: Tarjan's [25] and Dijkstra's [5, 6]. The latter is less known, but better suited to model checking (it has less overhead and can abort earlier). Our trick to use a threshold to prevent SCC merges could work with either algorithms, but for the emptiness-check to be correct we also need to perform the DFS in terms on SCCs instead of working in terms of states. This ordering is possible with Dijkstra's algorithm, but not Tarjan's.

Implementation. Fig. 3 presents the algorithm. Its structure mimics that of the emptiness check for TGBA of Couvreur et al. [4], especially it profits from the idea of performing the DFS in terms of SCCs rather than states: the stack SCC serves both as a stack of connected components and as the DFS stack. The constituents of each entry are $state$ (the root state of the SCC), $root$ (its DFS number), la (the acceptance conditions of the incoming transition to $state$), acc (the acceptance conditions of the cycles inside the SCC), rem (the other states of the SCC), $succ$ and $fsucc$ (the unexplored successors of the SCC).

These unexplored successors are split into $succ$ and $fsucc$ to ensure a proper ordering with respect to avoided acceptance conditions. When a state is pushed down on SCC at line 29, $fsucc$ is loaded with all transitions in acceptance conditions that must be avoided, while $succ$ receive the others. The latter will be visited first: the algorithm always pick the next successor to visit from $succ$ (line 16) and will swap $fsucc$ and $succ$ once $succ$ is empty (lignes 9–11).

Thresholds, meant to prevent merging SCCs using a cycle that would use an unsatisfiable acceptance condition, are represented by the number (in DFS order) of the last state of the SCC from which the threshold transition is going out (that is 2 on our example). These numbers form the min stack; they are used line 20 before deciding whether to merge; they are pushed when $fsucc$ and $succ$ are swapped line 12, and are popped when the state of that number is removed line 35.

The acceptance conditions to avoid are pushed on top of a stack called $avoid$ which is completed anytime the algorithm needs to revisit an SCC (line 43). Each element of this stack is a pair $(ar, \overline{acc})$ where $root$ is the number of the first state of the SCC starting at which acceptance conditions in $\overline{acc}$ should be avoided. This stack is popped when the SCC rooted at $root$ has been visited and has to be removed (lines 37–38).

Correctness. Termination is guaranteed by the DFS and the fact that the number of avoided acceptance conditions cannot exceed $|\mathcal{F}|$. By lack of space, we only give the scheme of our proof that this algorithm will return $\bot$ if an accepting run exists in the input TSA, and will return $\top$ otherwise. (A complete proof is available in French [7].)

Let us use the following notations to describe the state of the algorithm:

$$\begin{aligned} SCC =& \langle state_0, root_0, la_0, acc_0, rem_0, succ_0, fsucc_0\rangle \\ & \langle state_1, root_1, la_1, acc_1, rem_1, succ_1, fsucc_1\rangle \\ & \vdots \\ & \langle state_n, root_n, la_n, acc_n, rem_n, succ_n, fsucc_n\rangle \\ min =& min_0 min_1 \dots min_p \\ avoid =& \langle ar_0, \overline{acc_0}\rangle \langle ar_1, \overline{acc_1}\rangle \dots \langle ar_r, \overline{acc_r}\rangle \end{aligned}$$

Furthermore, let us denote $\mathfrak{S}_i$ the set of states represented by $SCC[i]$, and $\varphi(x)$ the index of the SCC containing the state numbered x:

$$\begin{aligned} \mathfrak{S}_i =& \{s \in \mathcal{Q} \mid root_i \leq H[s] < root_{i+1}\} \qquad \text{for } 0 \leq i < n \\ \mathfrak{S}_n =& \{s \in \mathcal{Q} \mid root_n \leq H[s]\} \\ \varphi(x) =& \max\{i \mid root_i \leq x\} \end{aligned}$$

Lemma 1 *At any time between the execution of lines 8–15, for any pair $\langle ar_i, \overline{acc_i}\rangle$ on the $avoid$ stack, there exists a unique entry $\langle state_j, root_j, la_j, acc_j, rem_j, succ_j, fsucc_j\rangle$ on the SCC stack such that $ar_i = root_j$. In other words, the $avoid$ entries are always associated to roots of SCCs.*

Lemma 2 *When line 16 is run to pick a state amongst the successors of the top of SCC, the value of $\overline{acc_r}$ is the same as when this set of successors was created at line 29.*

Lemma 3 *The values of $(root_i)_{i\in[\![0,n]\!]}$ are strictly increasing and we have $root_n \leq max$ at all times between the execution of lines 8–15.*

Lemma 4 *Let us call n' the value of n at a moment right after lines 11–12 have been run. The sets $succ_{n'}$ and $fsucc_{n'}$ will never increase.*

Lemma 5 *The function g that to any $i \in \{0, ..., p\}$ associates $g(i) = \varphi(min_i)$ is injective. In other words, two states numbered min_{i_1} and min_{i_2} (with $i_1 \neq i_2$) cannot belong to the same SCC. Furthermore, if $n > min_p$, $root_{\varphi(min_p)+1} = min_p + 1$. In other words, min_p is the number of the last state of the SCC whose $root$ has the number $root_{\varphi(min_p)}$. Finally, $root_{\varphi(min_p)} \leq min_p \leq max$.*

The state set $\mathcal{Q}$ of the TSA to check can be partitioned in three sets:

- *active states* are those which appear in H associated to a non-null value,
- *removed states* are those which appear in H with a null value,
- *unexplored states* are not yet in H.

The algorithm can move a state from the *unexplored* set to the *active* set, and from there it can move it either to the *removed* set or back to the *unexplored* set (lines 41–42).

The following invariants are preserved by all the lines of the main function (lines 8–15). They need to be proved together as their proofs are interdependent.

Invariant 1 *For all $i \leq n$, the subgraph induced by the states of $\mathfrak{S}_i$ is a SCC. Furthermore there exists a cycle in this SCC that visits all acceptance conditions of acc_i. Finally $\mathfrak{S}_0, \mathfrak{S}_1, \dots, \mathfrak{S}_n$ is a partition of the set of* active states.

Invariant 2 $\forall i < n, \exists s \in \mathfrak{S}_i, \exists s' \in \mathfrak{S}_{i+1}, \exists p \in 2^{\Sigma}, \{f \in \mathcal{F} \mid (s,p,s') \in f\} = la_{i+1}$. *I.e., there exists a transition between the SCCs indexed by i and $i+1$ that is in all that acceptance conditions of la_{i+1}.*

Invariant 3 *There is exactly max* active states*. No state of H is associated to a value greater than max. If two different states are associated to the same value in H, this value is 0. In particular, this means that for any value v between 1 and max, there exists a unique* active state *s such that $H[s] = v$.*

Invariant 4 *For all integer $i \leq n$, the set rem_i holds all the states of $\mathfrak{S}_i \setminus \{state_i\}$.*

Invariant 5 *Any* removed state *q cannot be part of an accepting run.*

Invariant 6 *There is no state accessible from $state_n$ from which we could find an accepting cycle using a transition in an acceptance condition from $\overline{acc_r}$.*

Invariant 7 *All transitions going from $\mathfrak{S}_{\varphi(min_p)}$ to $\mathfrak{S}_{\varphi(min_p)+1}$ are labelled by an acceptance condition of $\overline{acc_r}$. (In particular, $la_{\varphi(min_p)+1} \cap \overline{acc_r} \neq \emptyset$.)*

Invariant 8 $\forall j \geq \varphi(min_p), acc_j \cap \overline{acc_r} = \emptyset$ *and* $\forall j > \varphi(min_p)+1, la_j \cap \overline{acc_r} = \emptyset$. *In other words, the SCC built after the last threshold, and the transitions between them, are not in acceptance conditions from $\overline{acc_r}$, except for the first transition visited after the last threshold (in $la_{\varphi(min_p)+1}$).*

The first two invariants imply that if the algorithm finds an i such that $\forall (l,u) \in \mathcal{F}, acc_i \in l \implies acc_i \in u$, then $SCC[i]$ is an accepting SCC (inv. 1) that is accessible (inv. 1 & 2), so the algorithm can terminate with $\bot$. Invariant 5 assures that no accepting run exists once all states have been *removed*: the algorithm therefore terminates with $\top$.

5 Conclusion

We have introduced a new algorithm for the on-the-fly emptiness check of transition-based Streett automata (TSA), that generalizes the algorithm for transition-based Büchi automata of Couvreur [3]. This algorithm checks the emptiness of a TSA A with $|A|_t$ transitions and $|\mathcal{F}|$ acceptance pairs in $\mathrm{O}(|A|_t \times |\mathcal{F}|)$ time. We have seen that this algorithm allows us to check a linear-time property on a model A_M under n strong fairness hypotheses in $\mathrm{O}(|A_M|_t \times 2^{\mathrm{O}(|\varphi|)} \times n)$ time instead of the $\mathrm{O}(|A_M|_t \times 2^{\mathrm{O}(|\varphi|+n)})$ we would have using Büchi automata.

It should be noted that since Büchi automata can be seen as Streett automata without any structural change, this very same algorithm can also be used to check the emptiness of Büchi automata. In that case SCCs will never have to be revisited (the $avoid$ stack stays empty) and the algorithms performs the same operations as the original algorithm for Büchi automata.

Using Streett automata could also be useful to translate some LTL properties that look like strong fairness properties. For instance Sebastiani et al. [24] give the following LTL formula as an example of a property whose negation is hard to translate to Büchi

automata (most of the tools blow up):

$$\Big((\mathsf{GF}\,p_0 \to \mathsf{GF}\,p_1) \wedge (\mathsf{GF}\,p_2 \to \mathsf{GF}\,p_0) \wedge \\ (\mathsf{GF}\,p_3 \to \mathsf{GF}\,p_2) \wedge (\mathsf{GF}\,p_4 \to \mathsf{GF}\,p_2) \wedge \\ (\mathsf{GF}\,p_5 \to \mathsf{GF}\,p_3) \wedge (\mathsf{GF}\,p_6 \to \mathsf{GF}(p_5 \vee p_4)) \wedge \\ (\mathsf{GF}\,p_7 \to \mathsf{GF}\,p_6) \wedge (\mathsf{GF}\,p_1 \to \mathsf{GF}\,p_7)\Big) \to \mathsf{GF}\,p_8$$

Spot's LTL-to-Büchi translator [8] produces a TGBA with 1731 states for the negation of this formula. With a dedicated algorithm Sebastiani et al. were able to produce a 1281-state Generalized Büchi automaton. However this formula has the form $\psi \to \varphi$ where ψ is a combinaison of 8 strong fairness hypotheses, and $\neg\varphi$ can be expressed as a Büchi automaton with 2 states and no acceptance condition. The whole formula can therefore be expressed as a transition-based Streett automaton with two states and 8 pairs of acceptance conditions.[6] This reduction should not be a surprise since Streett automata are exponentially more succinct than Büchi automata [23], however this example shows that it would be useful to have an efficient algorithm to translate LTL formulæ to Streett automata. Unfortunately we are not aware of any published work in this area.

Bibliography

[1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[3] J.-M. Couvreur. On-the-fly verification of temporal logic. In *Proc. FM'99*, vol. 1708 of *LNCS*, pages 253–271, Toulouse, France, Sept. 1999. Springer-Verlag.

[4] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. SPIN'05*, vol. 3639 of *LNCS*, pages 143–158. Springer-Verlag, Aug. 2005.

[5] E. W. Dijkstra. EWD 376: Finding the maximum strong components in a directed graph. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF, May 1973.

[6] E. W. Dijkstra. Finding the maximal strong components in a directed graph. In *A Discipline of Programming*, chapter 25, pages 192–200. Prentice-Hall, 1976.

[7] A. Duret-Lutz. *Contributions à l'approche automate pour la vérification de propriétés de systèmes concurrents*. PhD thesis, Université Pierre et Marie Curie (Paris 6), July 2007.

[8] A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Proc. MASCOTS'04*, pages 76–83, Volendam, The Netherlands, Oct. 2004. IEEE Computer Society.

[9] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, June 1987.

[6] Combining this with the TSA to TGBA construction from section 3.3 yields a TGBA of $2 \times (2^8 + 1) = 514$ states that is even smaller than that of Sebastiani et al.

[10] N. Francez. *Fairness*. Springer-Verlag, 1986.
[11] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. CAV'01*, vol. 2102 of *LNCS*, pages 53–65, Paris, France, 2001. Springer-Verlag.
[12] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In *Proc. FORTE'02*, vol. 2529 of *LNCS*, pages 308–326, Houston, Texas, Nov. 2002. Springer-Verlag.
[13] P. Godefroid, G. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, Nov. 1995.
[14] Y. Kesten, A. Pnueli, and M. Y. Vardi. Verification by augmented abstraction: The automata-theoretic view. *Journal of Computer and System Sciences*, 62(4): 668–690, 2001.
[15] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):1–19, 2000.
[16] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. the 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107. ACM, 1985.
[17] C. Löding. Methods for the transformation of ω-automata: Complexity and connection to second order logic. Diploma thesis, Institue of Computer Science and Applied Mathematics, 1998.
[18] S. Merz. Model checking: A tutorial overview. In *Proc. MOVEP'00*, vol. 2067 of *LNCS*, pages 3–38. Springer-Verlag, 2001.
[19] M. Michel. Algèbre de machines et logique temporelle. In *Proc. STACS'84*, vol. 166 of *LNCS*, pages 287–298, Paris, Apr. 1984.
[20] M. Rauch Henzinger and J. A. Telle. Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In *Proc. SWAT'96*, vol. 1097 of *LNCS*, pages 16–27, Reykjavík, Iceland, July 1996. Springer-Verlag.
[21] S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, Mar. 1989.
[22] S. Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition. In *Proc. STOC'92*. ACM, May 1992.
[23] S. Safra and M. Y. Vardi. On ω-automata and temporal logic (preliminary report). In *Proc. STOC'89*, pages 127–137. ACM, 1989.
[24] R. Sebastiani, S. Tonetta, and M. Y. Vardi. Symbolic systems, explicit properties: on hybrid approches for LTL symbolic model checking. In *Proc. CAV'05*, vol. 3576 of *LNCS*, pages 350–363, Edinburgh, July 2005. Springer-Verlag.
[25] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
[26] H. Tauriainen. *Automata and Linear Temporal Logic: Translation with Transition-based Acceptance*. PhD thesis, Helsinki University of Technology, Espoo, Finland, Sept. 2006.
[27] M. Y. Vardi. The Büchi complementation saga. In *Proc. STACS'07*, Aachen, Germany, Feb. 2007. Invited paper.
[28] M. Y. Vardi. Automata-theoretic model checking revisited. In *Proc. VMCAI'07*, vol. 4349 of *LNCS*, Nice, France, Jan. 2007. Springer-Verlag. Invited paper.
[29] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. Banff'94*, vol. 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.