Single-pass Testing Automata for LTL Model Checking

Ala Eddine Ben Salem

LRDE, EPITA, Le Kremlin-Bicêtre, France, ala@lrde.epita.fr

Abstract. *Testing Automaton* (TA) is a new kind of ω-automaton introduced by Hansen et al. [6] as an alternative to the standard Büchi Automata (BA) for the verification of stutter-invariant LTL properties. Geldenhuys and Hansen [5] shown later how to use TA in the automata-theoretic approach to LTL model checking. They propose a TA-based approach using a verification algorithm that requires two searches (two passes) and compare its performance against the BA approach. This paper improves their work by proposing a transformation of TA into a normal form (STA) that only requires a single one-pass verification algorithm. The resulting automaton is called *Single-pass Testing Automaton* (STA). We have implemented the STA approach in Spot model checking library. We are thus able to compare it with the "traditional" BA and TA approaches. These experiments show that STA compete well on our examples.

1 Introduction

The automata-theoretic approach [11] to LTL model checking relies on ω-automata (i.e., an extension of finite automata to infinite words). It starts by converting the negation of the LTL formula φ into an ω-automaton $A_{\neg\varphi}$, then composing that automaton with the state-space of a model M given as a Kripke structure $\mathcal{K}_M$ (a variant of ω-automaton), and finally checking the language emptiness of the resulting product automaton $A_{\neg\varphi} \otimes \mathcal{K}_M$. This operation tells whether $A_{\neg\varphi} \otimes \mathcal{K}_M$ accepts an infinite word, and can return such a word as a counterexample. The model M satisfies φ iff $\mathscr{L}(A_{\neg\varphi} \otimes \mathcal{K}_M) = \emptyset$.

As for any model checking process, the automata-theoretic approach suffers from the well known state explosion problem. In practice, it is the product automaton that can be very large, its size can reach $(|A_{\neg\varphi}| \times |\mathcal{K}_M|)$ states, which can make it impossible to be handled using the resources of modern computers.

The ω-automaton representing $A_{\neg\varphi}$ is usually a Büchi Automaton (BA). This paper focuses on improving another kind of ω-automaton called *Testing Automaton (TA)*. TA is a variant of an "extended" Büchi automaton introduced by Hansen et al. [6]. Instead of observing the valuations on states or transitions, the TA transitions only record the changes between these valuations. In addition, TA are less expressive than Büchi automata since they are able to represent only stutter-invariant [3] properties. Also they are often a lot larger than their equivalent Büchi automaton, but their high degree of determinism [6] often leads to a smaller product size [5].

In a previous work [1], we evaluated the use of TA for the model checking of stutter-invariant LTL properties. We have shown that the TA approach is efficient when the formula to be verified is violated (i.e., a counterexample exists). This is not the case when the property is satisfied since the product automaton ($A_{\neg\varphi} \otimes \mathcal{K}_M$) has to be visited twice during the the emptiness check.

In this work, we improve the TA approach in order to avoid the second pass of the emptiness check algorithm. To achieve this goal, we propose a transformation of TA into a normal form that does not require such a second pass, called *Single-pass Testing Automata* (STA). We have implemented the algorithms of STA approach in Spot [10] library. Our experimental comparisons between BA, TA and STA approaches show that the STA approach is statistically more efficient when no counterexample is found (i.e., the property is satisfied) because it does not require a second pass.

2 Existing Approaches

Let AP a set of atomic propositions, a valuation ℓ over AP is a function $\ell : AP \mapsto \{\bot, \top\}$ (i.e., an assignment of truth value to each atomic proposition). We denote by $\Sigma = 2^{AP}$ the set of all valuations over AP, where a valuation $\ell \in \Sigma$ is interpreted either as the set of atomic propositions that are true, or as a Boolean conjunction. For instance if $AP = \{a,b\}$, then $\Sigma = 2^{AP} = \{\{a,b\},\{a\},\{b\},\emptyset\}$ or equivalently $\Sigma = \{ab, a\bar{b}, \bar{a}b, \bar{a}\bar{b}\}$.

2.1 Büchi Automata (BA)

A Büchi Automaton (BA) is an ω-automaton [4] with valuations on transitions and acceptance conditions on states. Any LTL formula φ can be converted into a BA that accepts the same executions that satisfy φ [11].

Definition 1 (BA) *A* Büchi Automaton (BA) *over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{B} = \langle \mathcal{Q}, I, \delta, \mathcal{F} \rangle$ where:*

- *$\mathcal{Q}$ is a finite set of states, $I \subseteq \mathcal{Q}$ is a finite set of initial states,*
- *$\mathcal{F} \subseteq \mathcal{Q}$ is a finite set of accepting states ($\mathcal{F}$ is called the accepting set),*
- *$\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the transition relation where each transition is labeled by a letter ℓ of Σ, i.e., each element $(q,\ell,q') \in \delta$ represents a transition from state q to state q' labeled by a valuation $\ell \in 2^{AP}$.*

A run *of $\mathcal{B}$ over an infinite word $\sigma = \ell_0\ell_1\ell_2 \ldots \in \Sigma^\omega$ is an infinite sequence of transitions $r = (q_0,\ell_0,q_1)(q_1,\ell_1,q_2)(q_2,\ell_2,q_3)\ldots \in \delta^\omega$ such that $q_0 \in I$ (i.e., the infinite word is recognized by the run). Such a run is said to be accepting if $\forall i \in \mathbb{N}, \exists j \geq i, q_j \in \mathcal{F}$ (i.e., at least one accepting state is visited infinitely often). The infinite word σ is accepted by $\mathcal{B}$ if there exists an accepting run of $\mathcal{B}$ over σ.*

Figure 1 shows a BA recognizing the LTL formula $(a \,\mathsf{U}\, \mathsf{G} b)$. In this BA, the Boolean conjunctions labeling each transition are valuations over $AP = \{a,b\}$. The LTL formulas labeling each state represent the property accepted starting from this state of the automaton: they are shown for the reader's convenience but not used for model checking. As an illustration of Definition 1, the infinite word $ab; a\bar{b}; \bar{a}b; ab; \bar{a}b; ab; \ldots$ is accepted by the BA of Figure 1. A run over such infinite word must start in the initial state labeled by the formula $(a \,\mathsf{U}\, \mathsf{G} b)$ and remains in this state for the first two valuations $ab; a\bar{b}$, then it changes the value of a, so it has to take the transition labeled by the valuation $\bar{a}b$ to move to the second state labeled by the formula $(\mathsf{G} b)$. Finally, to be accepted, it must stay on this accepting state by executing infinitely the transitions labeled by $\{\bar{a}b, ab\}$.

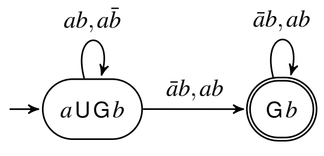


Fig. 1. A BA $\mathcal{B}$ for the LTL formula $a \mathsf{U} \mathsf{G} b$, with accepting states shown as double circles.

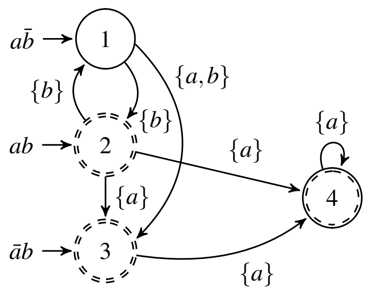


Fig. 2. A TA $\mathcal{T}$ for the LTL formula $a \mathsf{U} \mathsf{G} b$.

2.2 Model Checking using BA

The synchronous product of a BA $\mathcal{B}$ with a Kripke structure $\mathcal{K}$ is a BA $\mathcal{K} \otimes \mathcal{B}$ whose language is the intersection of both languages. Testing this product automaton (i.e., a BA) for emptiness amounts to the search of an accepting cycle that contains at least one accepting states (of $\mathcal{F}$).

Algorithm 1 presented below is an iterative version of the Couvreur's SCC-based algorithm [2] adapted to the emptiness check of BA. Algorithm 1 computes on-the-fly the Maximal Strongly Connected Components (MSCCs) of the BA representing the product $\mathcal{K} \otimes \mathcal{B}$: it performs a Depth-First Search (DFS) for SCC detection and then merges the SCCs belonging to the same Maximal SCC into a single SCC. After each merge, if the merged SCC contains an accepting state from $\mathcal{F}_{\otimes}$ (line 16), then an accepting run (i.e., a counterexample) is found (line 17) and the $\mathscr{L}(\mathcal{K} \otimes \mathcal{B})$ is not empty. *todo* is the DFS stack. It is used by the procedure `DFSpush` to push the states of the current DFS path and the set of their successors that have not yet been visited. H maps each visited state to its rank in the DFS order, and $H[s] = 0$ indicates that s is a dead state (i.e., s belongs to a maximal SCC that has been fully explored). The *SCC* stack stores a chain of partial SCCs found during the DFS. For each SCC the attribute *root* is the DFS rank (H) of the first state of the SCC, *acc* is the set of accepting states belonging to the SCC, and *rem* contains the fully explored states of the SCC.

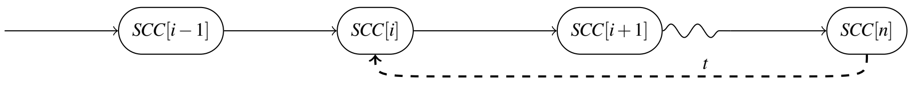


Fig. 3. *SCC* search stack and how the SCCs are merged.

1. The algorithm 1 begins by pushing in *SCC* each state s visited for the first time (line 12), as a trivial SCC with the set $acc = \{s\} \cap \mathcal{F}_{\otimes}$ (line 22).
2. Then, when the DFS explores a transition t between two states s and d, if d is in the SCC stack (line 14), then t closes a cycle passing through s and d in the product automaton. This cycle "strongly connects" all SCCs pushed in the *SCC* stack between $SCC[i]$ and $SCC[n]$: the two SCCs that respectively contains the states d and s ($SCC[n]$ is the top of the *SCC* stack).
3. All the SCCs between $SCC[i]$ and $SCC[n]$ are merged (line 15) into $SCC[i]$. The merge of SCCs is illustrated by Figure 3: a "back" transition t is found between $SCC[n]$ and $SCC[i]$, therefore the latest SCCs (from i to n) are merged.

```
1  Input: A BA $\mathcal{K} \otimes \mathcal{B} = \langle \mathcal{S}_\otimes, I_\otimes, \delta_\otimes, \mathcal{F}_\otimes \rangle$
2  Result: $\top$ if and only if $\mathscr{L}(\mathcal{K} \otimes \mathcal{B}) = \emptyset$
3  Data: $todo$: stack of $\langle state \in \mathcal{S}_\otimes, succ \subseteq \delta_\otimes \rangle$
         $SCC$: stack of
         $\langle root \in \mathbb{N}, acc \subseteq \mathcal{F}_\otimes, rem \subseteq \mathcal{S}_\otimes \rangle$
         $H$: map of $\mathcal{S}_\otimes \mapsto \mathbb{N}$
         $max \leftarrow 0$
4  begin
5    foreach $s^0 \in I_\otimes$ do
6      DFSpush($s^0$)
7      while $\neg todo$.empty() do
8        if $todo$.top().$succ = \emptyset$ then
9          DFSpop()
10       else
11         pick one $\langle s, \_, d \rangle$ off $todo$.top().$succ$
12         if $d \notin H$ then
13           DFSpush($d$)
14         else if $H[d] > 0$ then
15           merge($H[d]$)
16           if $SCC$.top().$acc \neq \emptyset$ then
17             return $\bot$
18   return $\top$

19 DFSpush($s \in \mathcal{S}_\otimes$)
20   $max \leftarrow max + 1$
21   $H[s] \leftarrow max$
22   $SCC$.push($\langle max, (\{s\} \cap \mathcal{F}_\otimes), \emptyset \rangle$)
23   $todo$.push($\langle s, \{\langle q, l, d \rangle \in \delta_\otimes \mid q = s\} \rangle$)
24 DFSpop()
25   $\langle s, \_ \rangle \leftarrow todo$.pop()
26   $SCC$.top().$rem$.insert($s$)
27   if $H[s] = SCC$.top().$root$ then
28     foreach $s \in SCC$.top().$rem$ do
29       $H[s] \leftarrow 0$
30     $SCC$.pop()
31 merge($t \in \mathbb{N}$)
32   $r \leftarrow \emptyset$
33   while $t < SCC$.top().$root$ do
34     $acc \leftarrow acc \cup SCC$.top().$acc$
35     $r \leftarrow r \cup SCC$.top().$rem$
36     $SCC$.pop()
37   $SCC$.top().$acc \leftarrow SCC$.top().$acc \cup acc$
38   $SCC$.top().$rem \leftarrow SCC$.top().$rem \cup r$
```

Algorithm 1: Emptiness check algorithm for BA.

4. The set of accepting states of the merged SCC is equal to the union of $SCC[i].acc \cup SCC[i+1].acc \cup \cdots \cup SCC[n].acc$. If this union contains an accepting state of $\mathcal{F}_\otimes$, then the merged SCC is accepting and the algorithm return *false* (line 17): the product is not empty.

2.3 Testing Automata (TA)

Testing Automata were introduced by Hansen et al. [6] to represent stutter-invariant [3] properties. While a Büchi automaton observes the value of the atomic propositions AP, the basic idea of TA is to only detect the *changes* in these values, making TA particularly suitable for stutter-invariant properties; if a valuation of AP does not change between two consecutive valuations of an execution, the TA can stay in the same state, this kind of transitions are called stuttering transitions. To detect infinite executions that end stuck in the same TA state because they are stuttering, a new kind of accepting states is introduced: *livelock-accepting states.*

If A and B are two valuations, $A \oplus B$ denotes the symmetric set difference, i.e., the set of atomic propositions that differ (e.g., $a\bar{b} \oplus ab = \{b\}$).

Definition 2 (TA) *A TA over the alphabet* $\Sigma = 2^{AP}$ *is a tuple* $\mathcal{T} = \langle \mathcal{Q}, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$, *where:*
- $\mathcal{Q}$ *is a finite set of states,* $I \subseteq \mathcal{Q}$ *is a finite set of initial states,*
- $U : I \to 2^\Sigma$ *is a function mapping each initial state to a set of valuations (set of possible initial configurations),*

– *$\mathcal{F} \subseteq \mathcal{Q}$ is a set of Büchi-accepting states,*
– *$\mathcal{G} \subseteq \mathcal{Q}$ is a set of livelock-accepting states,*
– *$\delta \subseteq \mathcal{Q} \times (\Sigma \setminus \emptyset) \times \mathcal{Q}$ is the transition relation where each transition (s,k,d) is labeled by a* changeset*: $k \in \Sigma$ is interpreted as a non empty set of atomic propositions whose value must change between states s and d.*

An infinite word $\sigma = \ell_0\ell_1\ell_2 \ldots \in \Sigma^\omega$ is accepted by $\mathcal{T}$ iff *there exists an infinite sequence $r = (q_0, \ell_0 \oplus \ell_1, q_1)(q_1, \ell_1 \oplus \ell_2, q_2) \ldots (q_i, \ell_i \oplus \ell_{i+1}, q_{i+1}) \ldots \in (\mathcal{Q} \times \Sigma \times \mathcal{Q})^\omega$ such that:*
– *$q_0 \in I$ with $\ell_0 \in U(q_0)$,*
– *$\forall i \in \mathbb{N}$, either $(q_i, \ell_i \oplus \ell_{i+1}, q_{i+1}) \in \delta$ (the execution progresses in the TA), or $(\ell_i = \ell_{i+1}) \wedge (q_i = q_{i+1})$ (the execution is stuttering and the TA does not progress),*
– *either, $\forall i \in \mathbb{N}$, $(\exists j \geq i, \ell_j \neq \ell_{j+1}) \wedge (\exists l \geq i, q_l \in \mathcal{F})$ (the TA is progressing in a Büchi-accepting way), or, $\exists n \in \mathbb{N}$, $(q_n \in \mathcal{G} \wedge (\forall k \geq n, q_k = q_n \wedge \ell_k = \ell_n))$ (the sequence reaches a livelock-accepting state and then stays on that state because the execution is stuttering).*

The construction of a TA from a BA is detailed in [5, 1]. To illustrate Definition 2, let us consider Figure 2, representing a TA $\mathcal{T}$ for $a\,\mathsf{U}\,\mathsf{G}\,b$. In this figure, the initial states 1, 2 and 3 are labeled respectively by the set of valuations $U(1) = \{a\bar{b}\}$, $U(2) = \{ab\}$ and $U(3) = \{\bar{a}b\}$. Each transition of $\mathcal{T}$ is labeled with a changeset over the set of atomic propositions $AP = \{a, b\}$. In a TA, states with a double enclosure belong to either $\mathcal{F}$ or $\mathcal{G}$: states in $\mathcal{F} \setminus \mathcal{G}$ have a double solid line, states in $\mathcal{G} \setminus \mathcal{F}$ have a double dashed line (states 2 and 3 of $\mathcal{T}$), and states in $\mathcal{F} \cap \mathcal{G}$ use a mixed dashed/solid style (state 4).

– The infinite word $ab;\bar{a}b;ab;\bar{a}b;ab;\bar{a}b;ab;\ldots$ is accepted by a Büchi accepting run of $\mathcal{T}$. Indeed, a run recognizing such word must start in state 2, then it always changes the value of a, so it has to take transitions labeled by $\{a\}$. For instance it could be the run $2 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \cdots$ or the run $2 \xrightarrow{\{a\}} 3 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \cdots$ Both visit the state $4 \in \mathcal{F}$ infinitely often, so they are Büchi accepting.
– The infinite word $ab;\bar{a}b;\bar{a}b;\bar{a}b;\ldots$ is accepted by a livelock accepting run of $\mathcal{T}$. An accepting run starts in state 2, then moves to state 4, and stutters on this livelock-accepting state. Another possible accepting run goes from state 2 to state 3 and stutters in $3 \in \mathcal{G}$.
– The infinite word $ab;a\bar{b};ab;a\bar{b};ab;a\bar{b};\ldots$ is not accepted. It would correspond to a run alternating between states 2 and 1, but such a run is neither Büchi accepting (does not visit any $\mathcal{F}$ state) nor livelock-accepting (it passes through state $2 \in \mathcal{G}$, but does not stay into this state continuously).

2.4 Model Checking using TA

The product of a Kripke and a TA is not a TA: while a TA execution is allowed to stutter on any state, the product execution must execute an explicit stuttering transition.

Definition 3 (Synchronous Product of a TA with a Kripke structure) *For a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ and a TA $\mathcal{T} = \langle \mathcal{Q}, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$, the product $\mathcal{K} \otimes \mathcal{T}$ is a tuple $\langle \mathcal{S}_\otimes, I_\otimes, U_\otimes, \delta_\otimes, \mathcal{F}_\otimes, \mathcal{G}_\otimes \rangle$ where*
– *$\mathcal{S}_\otimes = \mathcal{S} \times \mathcal{Q}$, $\mathcal{F}_\otimes = \mathcal{S} \times \mathcal{F}$, $\mathcal{G}_\otimes = \mathcal{S} \times \mathcal{G}$,*
– *$I_\otimes = \{(s,q) \in \mathcal{S}_0 \times I \mid l(s) \in U(q)\}$ with $\forall (s,q) \in I_\otimes$, $U_\otimes((s,q)) = \{l(s)\}$,*

– $\delta_\otimes = \{((s,q),k,(s',q')) \mid (s,s') \in \mathcal{R}, (q,k,q') \in \delta, k = l(s) \oplus l(s')\}.$
$\cup \{((s,q),\emptyset,(s',q')) \mid (s,s') \in \mathcal{R}, q = q', l(s) = l(s')\}$

An execution $\sigma = \ell_0\ell_1\ell_2 \ldots \in \Sigma^\omega$ is accepted by $\mathcal{K} \otimes \mathcal{T}$ if there exists an infinite sequence $r = (s_0, \ell_0 \oplus \ell_1, s_1)(s_1, \ell_1 \oplus \ell_2, s_2) \ldots (s_i, \ell_i \oplus \ell_{i+1}, s_{i+1}) \ldots \in (\mathcal{S}_\otimes \times \Sigma \times \mathcal{S}_\otimes)^\omega$ such that:

- *$s_0 \in \mathcal{S}^0_\otimes$ with $\ell_0 \in U_\otimes(s_0)$,*
- *$\forall i \in \mathbb{N}, (s_i, \ell_i \oplus \ell_{i+1}, s_{i+1}) \in \delta_\otimes$ (we are always progressing in the product)*
- *Either, $\forall i \in \mathbb{N}$, $(\exists j \geq i, \ell_j \neq \ell_{j+1}) \wedge (\exists l \geq i, s_l \in \mathcal{F}_\otimes)$ (the automaton is progressing in a Büchi-accepting way), or, $\exists n \in \mathbb{N}, \forall k \geq n, (\ell_k = \ell_n) \wedge (s_k \in \mathcal{G}_\otimes)$ (a suffix of the execution stutters in $\mathcal{G}_\otimes$).*

We have $\mathscr{L}(\mathcal{K} \otimes \mathcal{T}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{T})$ by construction.

Figure 4 shows an example of a synchronous product between a Kripke structure $\mathcal{K}$ and a TA $\mathcal{T}$ recognizing the LTL formula $\mathsf{F}\,\mathsf{G}\,p$. Each state of $\mathcal{K}$ is numbered and labeled with a valuation of atomic propositions (over $AP = \{p\}$) that hold in this state. In the product $\mathcal{K} \otimes \mathcal{T}$, states are labeled with a pairs of the form (s,q) where s is a state of $\mathcal{K}$ and q of $\mathcal{T}$, and the livelock accepting states are denoted by a double dashed circle.

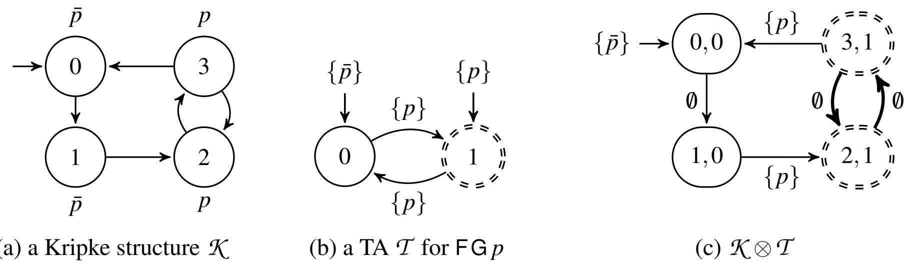


Fig. 4. Example of a product between a Kripke structure $\mathcal{K}$ and a TA $\mathcal{T}$ of $\mathsf{F}\,\mathsf{G}\,p$. The bold cycle of $\mathcal{K} \otimes \mathcal{T}$ is livelock-accepting.

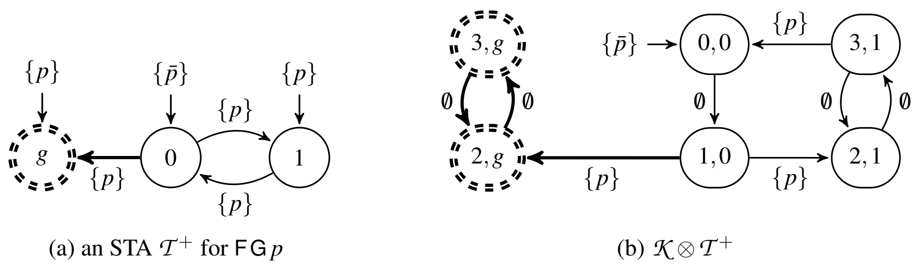


Fig. 5. Impact on the product of using STA $\mathcal{T}^+$ instead of TA $\mathcal{T}$. Bold states and transitions are addition relative to Figure 4.

A two-pass emptiness check algorithm In this section, we present a two-pass algorithm for the emptiness check of the synchronous product between a TA and a Kripke structure. In model checking approach using TA, the emptiness check requires a dedicated algorithm because according to the Definition 3, there are two ways to detect an accepting cycle in the product:

- Büchi accepting: a cycle containing at least one Büchi-accepting state ($\mathcal{F}_\otimes$) and at least one non-stuttering transition (i.e., a transition (s,k,s') with $k \neq \emptyset$),
- livelock accepting: a cycle composed only by stuttering transitions and livelock accepting states ($\mathcal{G}_\otimes$).

A straightforward emptiness check would have the following two passes: a first pass to detect Büchi accepting cycles and a second pass to detect livelock accepting cycles.

```
1  Input: $\mathcal{K}\otimes\mathcal{T} = \langle \mathcal{S}_\otimes, I_\otimes, U_\otimes, \delta_\otimes, \mathcal{F}_\otimes, \mathcal{G}_\otimes \rangle$
2  Result: $\top$ if and only if $\mathscr{L}(\mathcal{K}\otimes\mathcal{T}) = \emptyset$
3  Data: $todo$: stack of $\langle state \in \mathcal{S}_\otimes, succ \subseteq \delta_\otimes \rangle$
         $SCC$: stack of $\langle root \in \mathbb{N}, lk \in 2^{AP}, k \in 2^{AP}, acc \subseteq \mathcal{F}_\otimes, rem \subseteq \mathcal{S}_\otimes \rangle$
         $H$: map of $\mathcal{S}_\otimes \mapsto \mathbb{N}$
         $max \leftarrow 0$, $Gseen \leftarrow false$
4  begin
5      if $\neg$ first-pass() then return $\bot$ ;
6      if $Gseen$ then return second-pass() ;
7  first-pass()
8      foreach $s^0 \in I_\otimes$ do
9          DFSpush1($\emptyset$, $s^0$)
10         while $\neg todo.empty()$ do
11             if $todo.top().succ = \emptyset$ then
12                 DFSpop()
13             else
14                 pick one $\langle s,k,d \rangle$ off $todo$.top().$succ$
15                 if $d \notin H$ then
16                     DFSpush1(k, d)
17                 else if $H[d] > 0$ then
18                     merge1(k, $H[d]$)
19                     if $(SCC.top().acc \neq \emptyset) \wedge$
                       $(SCC.top().k \neq \emptyset)$ then return $\bot$;
20                     if $(d \in \mathcal{G}_\otimes) \wedge (SCC.top().k = \emptyset)$
                       then return $\bot$ ;
21     return $\top$
22 DFSpush1($lk \in 2^{AP}$, $s \in \mathcal{S}_\otimes$)
23     $max \leftarrow max + 1$
24     $H[s] \leftarrow max$
25     if $s \in \mathcal{F}_\otimes$ then
26         $SCC$.push($\langle max, lk, \emptyset, \{s\}, \emptyset \rangle$)
27     else
28         $SCC$.push($\langle max, lk, \emptyset, \emptyset, \emptyset \rangle$)
29     $todo$.push($\langle s, \{\langle q,k,d \rangle \in \delta_\otimes \mid q = s\} \rangle$)
30     if $s \in \mathcal{G}_\otimes$ then
31         $Gseen \leftarrow true$
32 merge1($lk \in 2^{AP}$, $t \in \mathbb{N}$)
33     $acc \leftarrow \emptyset$
34     $r \leftarrow \emptyset$
35     $k \leftarrow lk$
36     while $t < SCC.top().root$ do
37         $acc \leftarrow acc \cup SCC$.top().$acc$
38         $k \leftarrow k \cup SCC$.top().$k \cup SCC$.top().$lk$
39         $r \leftarrow r \cup SCC$.top().$rem$
40         $SCC$.pop()
41     $SCC$.top().$acc \leftarrow SCC$.top().$acc \cup acc$
42     $SCC$.top().$k \leftarrow SCC$.top().$k \cup k$
43     $SCC$.top().$rem \leftarrow SCC$.top().$rem \cup r$
```

Algorithm 2: The first-pass of the Emptiness check algorithm for TA products.

The first-pass of Algorithm 2 is similar to Algorithm 1, it detects all Büchi-accepting cycles, and with line 20 included in this algorithm, it detects also some livelock-accepting cycles. Since in certain cases it may fail to report some livelock-accepting cycles, a second pass is required to look for possible livelock-accepting cycles. However, if no livelock-accepting state is visited during the first pass (i.e., the product does not contain livelock-accepting states), then the second pass can be disabled: this is the purpose of variable *Gseen* of Algorithm 2 (line 6), where *Gseen* is a flag that records if a livelock-accepting state is detected during the exploration of the product by the first pass (line 30). This first-pass is based on the BA emptiness check algorithm presented in Algorithm 1 with the following changes:

- In each item *scc* of the *SCC* stack: the new field *scc.lk* stores the *change-set* labeling the transition coming from the previous SCC, and *scc.k* contains the union of all *change-sets* in *scc* (lines 38 and 42).

– After each merge, *SCC*.top() is checked for Büchi-acceptance (line 19) or livelock-acceptance (line 20) depending on the emptiness of *SCC*.top().*k*.

Figure 4 illustrates how the `first-pass` of Algorithm 2 can fail to detect the livelock accepting cycle in a product $\mathcal{K} \otimes \mathcal{T}$ as defined in Definition 3. In this example, $\mathcal{G}_{\mathcal{T}} = \{1\}$ therefore $(3,1)$ and $(2,1)$ are livelock-accepting states, and $C_2 = [(3,1) \rightarrow (2,1) \rightarrow (3,1)]$ is a livelock-accepting cycle.

However, the `first-pass` may miss this livelock-accepting cycle depending on the order in which it processes the outgoing transitions of $(3,1)$. If the transition $t_1 = ((3,1),\{p\},(0,0))$ is processed before $t_2 = ((3,1),\emptyset,(2,1))$, then the cycle $C_1 = [(0,0) \rightarrow (1,0) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (0,0)]$ is detected and the four states are merged in the same SCC before exploring t_2. After this merge (line 18), this SCC is at the top of the SCC stack. Subsequently, when the DFS explores t_2, the merge caused by the cycle C_2 does not add any new state to the SCC, and the *SCC* stack remains unchanged. Therefore, the test line 20 still return false because the union $SCC.top().k$ of all change-sets labeling the transitions of the SCC is not empty (it includes for example t_1's label: $\{p\}$). Finally, `first-pass` algorithm terminates without reporting any accepting cycle, missing C_2.

On the other side, if the `first-pass` had processed t_2 before t_1, it would have merged the states $(3,1)$ and $(2,1)$ in an SCC, and would have detected it to be livelock-accepting.

In general, to report a livelock-accepting cycle, the first-pass computes the union of all change-sets of the SCC containing this cycle. However, this union may include non-stuttering transitions belonging to other cycles of the SCC. In this case, the `second-pass` is required to search for livelock-accepting cycles, ignoring the non-stuttering transitions that may belong to the same SCC. In the next section, we propose a Single-pass Testing Automata STA, which allows to obtain a synchronous product in which such mixing of non-stuttering and stuttering transitions will never occur in SCCs containing livelock-accepting cycles, making the `second-pass` unnecessary.

It is important to say that in the experiments presented in the sequel, we implemented Algorithm 2 including an heuristic proposed by Geldenhuys and Hansen [5] to detect more livelock-accepting cycles during the first pass. However, when properties are satisfied, the second pass is always required because this heuristic fails to report some livelock-accepting cycles [5]. We don't present the details of this heuristic because we show in the next sections other solutions that allow to detect all the livelock-accepting cycles during the first pass and therefore remove the second pass (in all cases).

3 Converting a TA into a Single-pass Testing Automaton (STA)

In this section, we introduce STA, a transformation of TA into a normal form such that livelock-accepting states have no successors, and therefore STA approach does not need the second pass of the emptiness check of TA approach. This contribution improves the efficiency of the model checking (this will be experimentally evaluated in section 4). In addition, STA simplify the implementation (and the optimization) of the emptiness check algorithm as it renders unnecessary the implementation of the second pass.

Definition 4 (STA) *A Single-pass Testing Automaton (STA) is a Testing Automaton* $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$ *over* Σ *such that* $\delta \cap (\mathcal{G} \times \Sigma \times Q) = \emptyset$. *In other words, an STA is a TA in which every livelock-accepting state has no successors.*

3.1 Construction of an STA from a TA

Property 1 formalizes the construction of an STA from a TA. We can transform a TA into an STA by adding an unique livelock-accepting state g (i.e., in STA, $G = \{g\}$), and adding a transition (q,k,g) for any transition (q,k,q') that goes into a livelock-accepting state $q' \in \mathcal{G}$ of the original automaton. In addition, if q' has no successors then q' can be removed, since it is bisimilar to the new state g.

Property 1 *Let $\mathcal{T} = \langle \mathcal{Q}, I, U, \delta, \mathcal{F}, \mathcal{G}\rangle$ be a TA, we construct an equivalent STA $\mathcal{T}' = \langle \mathcal{Q}', I', U', \delta', \mathcal{F}, \{g\}\rangle$ such that $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$ by the following:*

- *$\mathcal{Q}' = (\mathcal{Q} \setminus G_{\emptyset}) \cup \{g\}$ where $G_{\emptyset} = \{q \in \mathcal{G} \mid (\{q\} \times \Sigma \times \mathcal{Q}) \cap \delta = \emptyset\}$ is the set of states of $\mathcal{G}$ that have no successors, and $g \notin \mathcal{Q}$ is a new state,*
- *$I' = I \cup \{g\}$ if $\mathcal{G} \cap I \neq \emptyset$, $I' = I$ otherwise,*
- *$\delta' = (\delta \setminus (\mathcal{Q} \times \Sigma \times G_{\emptyset})) \cup \{(q,k,g) \mid (q,k,q') \in \delta,\ q' \in \mathcal{G}\}$,*
- *$\forall q \in I$, $U'(q) = U(q)$ and $U'(g) = \bigcup\limits_{q \in (\mathcal{G} \cap I)} U(q)$.*

Figure 5a shows how the TA from Figure 4b was transformed into an STA using Property 1. The idea behind this transformation is that any livelock-accepting execution of $\mathcal{T}$ will be mapped to an execution of $\mathcal{T}^+$ that is captured by the new state g. The new g state has an impact on the product (Figure 5b): the strongly connected components of this new product no longer mix non-stuttering transitions and livelock-accepting cycles: this renders the `second-pass` useless. The objective of STA is to isolate in the product the exploration of the parts that are composed only by livelock-accepting states and stuttering transitions, like the bold part of the product represented in the Figure 5b.

The STA emptiness check algorithm is the `first-pass` of the TA emptiness check algorithm without the `second-pass` procedure. In other words, in STA approach, the emptiness check is only Algorithm 2 (page 7) without line 6.

3.2 Correctness of the One-pass Emptiness Check using STA

In the following, $\mathcal{K}$, $\mathcal{T}$, $\mathcal{T}^+$ denote respectively a Kripke structure, a TA and an STA.

The `first-pass` is an SCC-based algorithm, it computes the set of all MSCCs (i.e., Maximal SCCs) of the product automaton. Therefore, in order to prove that the `first-pass` is sufficient to detect all livelock-accepting cycles, we prove that in $\mathcal{K} \otimes \mathcal{T}^+$, searching for all livelock-accepting cycles is equivalent to searching for all MSCCs that are only composed of stuttering transitions and livelock-accepting states. In Algorithm 2, line 20 allows to detect this kind of MSCCs.

Lemma 1 *In a product $\mathcal{K} \otimes \mathcal{T}$: **if** one MSCC M contains a product state (s,q) such that q is a livelock-accepting state that has no successors in $\mathcal{T}$, **then** M is only composed of stuttering transitions and livelock-accepting states.*

Proof. q has no successors in the TA $\mathcal{T}$, therefore from q, a run of $\mathcal{T}$ can only execute stuttering transitions: it stays in the same livelock-accepting state q. Consequently, all product states of M are connected by stuttering transitions. In addition, they have the same livelock-accepting state as TA component (q), therefore by Definition 3 all states of M are livelock-accepting.

Lemma 2 *In a product $\mathcal{K} \otimes \mathcal{T}^+$: one MSCC M contains a livelock-accepting state* ***if and only if*** *M is only composed of stuttering transitions and livelock-accepting states.*

Proof. ($\Longrightarrow$) If an MSCC M contains a livelock-accepting state (s,q) of $\mathcal{K} \otimes \mathcal{T}^+$, then q is a livelock-accepting state that has no successors in $\mathcal{T}^+$ because in STA every livelock-accepting state has no successors. The proof follows from Lemma 1 applied to $\mathcal{K} \otimes \mathcal{T}^+$.
($\Longleftarrow$) Any state of M is livelock-accepting.

The difference between Lemma 1 and Lemma 2 is that the livelock-accepting states of STA have no successors, while those of TA can. The following lemma is only for STA.

Lemma 3 *In the product automaton $\mathcal{K} \otimes \mathcal{T}^+$: there exists at least one livelock-acceptance cycle C* ***if and only if*** *there exists at least one non trivial MSCC M such that M is only composed of stuttering transitions and livelock-accepting states.*

Proof. ($\Longrightarrow$): The cycle C contains at least one livelock-accepting state, therefore applying Lemma 2 with M is the MSCC containing C allows us to conclude.
($\Longleftarrow$): M is non-trivial (i.e., it contains at least two states or a single state with a self-loop), therefore M contains at least one non-trivial cycle only composed of stuttering transitions and livelock-accepting states. This cycle is the livelock-accepting cycle C.

In Algorithm 2, the `first-pass` computes all MSCCs and line 20 allows to detect only the MSCCs satisfying Lemma 3. Thus, the STA emptiness check algorithm reports one cycle **if and only if** this cycle is a livelock-accepting or a Büchi-accepting cycle.

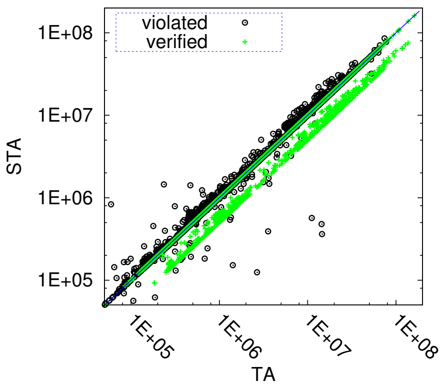
STA optimization The goal of this optimization is to reduce the number of transitions in STA, by exploiting the fact that the livelock-accepting states ($q' \in \mathcal{G}$) that are also Büchi-accepting ($q' \in \mathcal{F}$) do not require the second pass. Indeed, during the TA to STA transformation described by Property 1, it was unnecessary to add artificial transitions (q,k,g) for any transition (q,k,q') where $q' \in (\mathcal{G} \cap \mathcal{F})$, because any MSCC containing q' is necessarily an accepting MSCC and it is detected by the `first-pass` of Algorithm 2.

4 Experimental evaluation of STA

This section presents our experimentation conducted under the same conditions as our previous work [1], i.e., within the same tools Spot and CheckPN. We selected some Petri net models and formulas to compare the three approaches: BA, TA and STA.

The models are from the Petri net literature [9], we selected the following models: the flexible manufacturing system (FMS), the Kanban system, the Peterson algorithm, the slotted-ring system, the dining philosophers and the Round-robin mutex. We also used two models from actual case studies: **PolyORB** [8] and **MAPK** [7]. All these models have a parameter n. For the dining philosophers, the Peterson algorithm, the Round-robin, and the slotted-ring, the models are composed of n 1-safe subnets. For FMS and Kanban, n only influences the number of tokens in the initial marking. We selected two instances for each model: $n = 4/5$ for Peterson, FMS and Kanban; $n = 6/5$ for slotted-ring; $n = 9/10$ for dining philosophers; $n = 14/15$ for Round-robin.

For each selected model instance, we generated 200 verified formulas (i.e., no counterexample in the product) and 200 violated formulas (i.e., a counterexample exists):



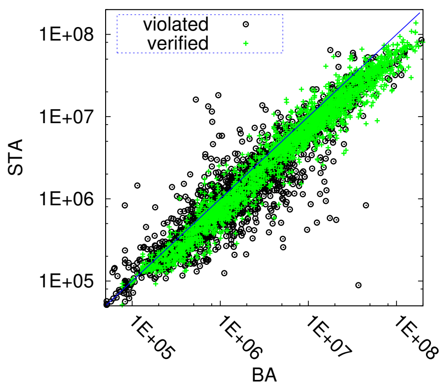


Fig. 6. Performance (transitions explored by the emptiness check) of STA against TA and BA.

100 random (length 15) and 100 weak-fairness [1] (length 30) of the two cases of formulas. Since generated formulas are very often trivial to verify (the emptiness check needs to explore only a handful of states), we selected for each model only those formulas requiring more than one second of CPU for the emptiness check in all approaches.

4.1 Results

Figure 6 compares the number of visited transitions when running the emptiness check; plotting STA against TA and BA. This gives an idea of their relative performance. Each point corresponds to one of the 5600 evaluated formulas (2800 violated with counterexample as black circles, and 2800 verified having no counterexample as grey crosses). Each point below the diagonal is in favor of STA while others are in favor of the other approach. Axes are displayed using a logarithmic scale.

4.2 Discussion

On verified properties, the results are very straightforward to interpret when looking at the number transitions explored by the emptiness check in Figure 6 .

STA significantly improve TA in all cases where a second pass was necessary. In these cases, the STA approach, with its single-pass emptiness check, is a clear improvement over TA. These cases where the STA approach is twice faster than TA's, appear as a linear cloud of grey crosses below the diagonal in the scatter plot of Figure 6 (we recall that the axes are displayed using a logarithmic scale). Otherwise, they have the same performance because if no livelock-acceptance states are detected in the product then the TA and STA approaches explore exactly the same product (these cases correspond to the grey crosses on the diagonal).

In the scatter plot comparing STA against BA, in most cases the grey crosses appear below the diagonal, i.e., the points where STA is better. Therefore, STA outperform BA for verified properties.

On violated properties, it is harder to interpret the results because they depend on the order in which non-deterministic transitions of the property automaton are explored. In the best case, the order of transitions leads the emptiness check straight to a counterexample; in the worst case, the algorithm explores the whole product until it finally finds a counterexample. The different kinds of property automata BA, TA and STA provide different orders of transitions and therefore change the number of states and transitions to be explored by the emptiness check before a counterexample is found.

5 Conclusion

In a preliminary work presented in [1], we experiment LTL model checking of stuttering-insensitive properties with various techniques: Büchi automata (BA), Transition-based Generalized Büchi Automata and Testing Automata (TA) [5]. At this time, conclusions were that TA has good performance for violated properties (i.e. when a counterexample was found). However, this was not the case when no counterexample was computed since the entire product had to be visited twice to check for each acceptance mode of a TA (Büchi acceptance or livelock-acceptance).

This paper extends the above work to avoid the second pass of the emptiness check algorithm in TA approach. It proposes a transformation of TA into STA, a Single-pass Testing Automata that avoids the need for a second pass.

The STA approach have been implemented in Spot, our model checking library and used on several benchmark models including large models issued from case studies. Experimentation with Spot reported that, STA remain good for violated properties, and also beat TA and BA in most cases when properties exhibit no counterexample.

References

1. Ben Salem, A.E., Duret-Lutz, A., Kordon, F.: Generalized Büchi automata versus testing automata for model checking. In: Proc. of SUMo'11. vol. 726, pp. 65–79. CEUR (June 2011)
2. Couvreur, J.M.: On-the-fly verification of temporal logic. In: Proc. of FM'99. LNCS, vol. 1708, pp. 253–271. Springer-Verlag (Sep 1999)
3. Etessami, K.: Stutter-invariant languages, ω-automata, and temporal logic. In: Proc. of CAV'99. LNCS, vol. 1633, pp. 236–248. Springer-Verlag (1999)
4. Farwer, B.: Automata logics, and infinite games, LNCS, vol. 2500, pp. 3–21. Springer (2002)
5. Geldenhuys, J., Hansen, H.: Larger automata and less work for LTL model checking. In: Proc. of SPIN'06. LNCS, vol. 3925, pp. 53–70. Springer (2006)
6. Hansen, H., Penczek, W., Valmari, A.: Stuttering-insensitive automata for on-the-fly detection of livelock properties. In: Proc. of FMICS'02. ENTCS, vol. 66(2). Elsevier (Jul 2002)
7. Heiner, M., Gilbert, D., Donaldson, R.: Petri nets for systems and synthetic biology. In: Proc. of SFM'08. LNCS, vol. 5016, pp. 215–264. Springer (2008)
8. Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Barrir, S., Vergnaud, T.: On the formal verification of middleware behavioral properties. In: Proc. of FMICS'04. ENTCS, vol. 133, pp. 139–157. Elsevier (Sep 2004)
9. Kordon, F., Linard, A., Buchs, D., Colange, M., Evangelista, S., Lampka, K., Lohmann, N., Paviot-Adet, E., Thierry-Mieg, Y., Wimmel, H.: Report on the model checking contest at petri nets 2011. T. Petri Nets and Other Models of Concurrency 6, 169–196 (2012)
10. MoVe/LRDE: The Spot home page: `http://spot.lip6.fr` (2014)
11. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Proc. of Banff'94. LNCS, vol. 1043, pp. 238–266. Springer-Verlag (1996)