Model Checking using Generalized Testing Automata

A.-E. Ben Salem[1,2], A. Duret-Lutz[1], and F. Kordon[2]

[1] LRDE, EPITA, Le Kremlin-Bicêtre, France
ala@lrde.epita.fr, adl@lrde.epita.fr
[2] LIP6, CNRS UMR 7606, Université P. & M. Curie~–~Paris~6, France
Fabrice.Kordon@lip6.fr

Abstract. Geldenhuys and Hansen showed that a kind of ω-automata known as *Testing Automata* (TA) can, in the case of stuttering-insensitive properties, outperform the Büchi automata traditionally used in the automata-theoretic approach to model checking~[10].
In previous work~[23], we compared TA against *Transition-based Generalized Büchi Automata* (TGBA), and concluded that TA were more interesting when counterexamples were expected, otherwise TGBA were more efficient.
In this work we introduce a new kind of automata, dubbed *Transition-based Generalized Testing Automata* (TGTA), that combine ideas from TA and TGBA. Implementation and experimentation of TGTA show that they outperform other approaches in most of the cases.

Keywords: testing automata, model checking, emptiness check.

1 Introduction

Context The automata-theoretic approach to model checking linear-time properties~[28] splits the verification process into four operations:

1. Computation of the state-space for the model M. This state-space can be seen as an ω-automaton A_M whose language, $\mathscr{L}(A_M)$, represents all possible infinite 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 infinite executions that would invalidate φ.
3. Synchronization of these automata. This constructs a product automaton $A_M \otimes A_{\neg\varphi}$ whose language, $\mathscr{L}(A_M) \cap \mathscr{L}(A_{\neg\varphi})$, is the set of executions of M invalidating φ.
4. Emptiness check of this product. This operation tells whether $A_M \otimes A_{\neg\varphi}$ accepts an infinite word, and can return such a word (a counterexample) if it does. The model M verifies φ iff $\mathscr{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$.

Problem Different kinds of ω-automata have been used with the above approach. In the most common case, a property expressed as an LTL (linear-time temporal logic) formula is converted into a Büchi automaton with state-based acceptance, and a Kripke structure is used to represent the state-space of the model.

In Spot~[17], our model checking library, we prefer to represent properties using *generalized* (i.e., multiple) Büchi acceptance conditions *on transitions* rather than on states~[7]. Any algorithm that translates LTL into a Büchi automaton has to deal with

generalized Büchi acceptance conditions at some point, and the process of *degeneralizing* the Büchi automaton often increases its size. Several emptiness-check algorithms can deal with generalized Büchi acceptance conditions, making such an a degeneralization unnecessary and even costly~[5]. Moving the acceptance conditions from the states to the transitions also reduces the size of the property automaton~[4, 13].

Unfortunately, having a smaller property automaton $A_{\neg\varphi}$ does not always imply a smaller product with the model ($A_M \otimes A_{\neg\varphi}$), and the size of this product really affects model checking efficiency. Thus, instead of targeting smaller property automata, some people have attempted to build automata that are *more deterministic*~[25]. However even this does not guarantee the product to be smaller.

Hansen et al.~[14] introduced a new kind of ω-automaton called *Testing Automaton* (TA). These automata are less expressive than Büchi automata since are tailored to represent *stuttering-insensitive* properties (such as any LTL property that does not use the X operator). They are often a lot larger than their equivalent Büchi automaton, but surprisingly, their high degree of determinism often lead to a smaller product~[10]. As a back-side, TA have two different modes of acceptance (Büchi-acceptance or livelock-acceptance), and their emptiness check may require two passes, mitigating the benefits of a having a smaller product.

Objectives The study of Geldenhuys and Hansen~[10] shows TA are statistically more efficient than Büchi automata. In a previous work~[23], we have extended their comparison to TGBA, and shown that TA are indeed better when the formula to be verified is violated (i.e., a counterexample is found), but this is not the case when the property is verified since the entire state space may have to be visited twice to check for each acceptance mode of a TA.

This paper introduces a new type of ω-automata, Transition-based Generalized Testing Automata (TGTA), that mixes features from both TA and TGBA. From TA, it reuses the labeling of transitions with changesets, and simplifications based on stuttering. From TGBA, it inherits the use of transition-based acceptance conditions. TGTA combine the advantages of TA and TGBA: it is still statistically more efficient than other ω-automata when the property is violated but does not require a second pass when no counterexample is found, thus remaining more efficient than other ω-automata in that situation.

We have implemented this new approach in Spot. This required little effort since TGTA reuse the emptiness check algorithm of TGBA. We are thus able to compare TGTA with the "traditional" algorithms we used on TA, BA and TGBA. These experiments show that TGTA compete well on our examples.

Contents Section~2 provides a brief summary of the three ω-automaton (BA, TGBA and TA) and pointers to their associated operations for model checking before Section~3 presents TGTA. Section~4 reports our experiments before a discussion in Section~5.

2 Presentation of Three Existing Approaches

Let AP designate the set of *atomic proposition* of the model. We use AP to build a linear-time property. Any state of the model is labeled by a valuation of these atomic propositions. We denote by $\Sigma = 2^{AP}$ the set of these valuations, which we interpret either as a set or as Boolean conjunctions. For instance if $AP = \{a, b\}$, then $\Sigma = 2^{AP} =$

$\{\{a,b\},\{a\},\{b\},\emptyset\}$ but we equivalently interpret it as $\Sigma = \{ab, a\bar{b}, \bar{a}b, \bar{a}\bar{b}\}$. All the executions of the model can be represented by a Kripke structure $\mathcal{K}$. An execution of the model is simply an infinite sequence of such valuations, i.e., an element from Σ^ω.

Definition 1 *A Kripke structure over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{K} = \langle S, I, R, L \rangle$. where:*

- *S is a finite set of states,*
- *$I \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is the transition relation,*
- *$L : S \to \Sigma$ is a state-labeling function.*

An execution $w = k_0 k_1 k_2 \ldots \in \Sigma^\omega$ is accepted by $\mathcal{K}$ if there exists an infinite sequence $s_0, s_1, \ldots \in S^\omega$ such that $s_0 \in I$ and $\forall i \in \mathbb{N}, L(s_i) = k_i \wedge (s_i, s_{i+1}) \in R$. The language accepted by $\mathcal{K}$ is the set $\mathscr{L}(\mathcal{K}) \subseteq K^\omega$ of executions it accepts.

A property can be seen as a set of sequences, i.e., a subset of Σ^ω. Among these properties, we want to distinguish those that are *stuttering-insensitive*:

Definition 2 *A property, or a language, i.e., a set of infinite sequences $\mathcal{P} \subseteq \Sigma^\omega$, is* stuttering-insensitive *iff any sequence $k_0 k_1 k_2 \ldots \in \mathcal{P}$ remains in $\mathcal{P}$ after repeating any valuation k_i or omitting duplicate valuations. Formally, $\mathcal{P}$ is stuttering-insensitive iff*

$$k_0 k_1 k_2 \ldots \in \mathcal{P} \iff k_0^{i_0} k_1^{i_1} k_2^{i_2} \ldots \in \mathcal{P} \text{~for any~} i_0 > 0, i_1 > 0 \ldots$$

Theorem 1 *Any LTL$\setminus$X formula (i.e., an LTL formula that does not use the* X *operator) describes a stuttering-insensitive property. Conversely any stuttering-insensitive property can be expressed as an LTL$\setminus$X formula*~[19].

The following sections presents three kinds of ω-automata~[8] that can be used to express properties in the automata-theoretic approach to model checking. *Transition-based Generalized Büchi Automata* and *Büchi Automata* can both express general properties, while *Testing Automata* are tailored to stuttering-insensitive properties.

2.1 Transition-based Generalized Büchi Automata

We begin by defining Transition-based Generalized Büchi Automata (TGBA), which are a generalization of the better known Büchi automata used for model checking~[13]. In our context, the TGBA represents the negation of the LTL property to verify.

Definition 3 *A TGBA over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{G} = \langle S, I, R, F \rangle$ where:*

- *S is a finite set of states,*
- *$I \subseteq S$ is the set of initial states,*
- *F is a finite set of acceptance conditions,*
- *$R \subseteq S \times 2^\Sigma \times 2^F \times S$ is the transition relation, where each element (s_i, K_i, F_i, d_i) represents a transition from state s_i to state d_i labeled by the non-empty set of valuation K_i, and a set of acceptance conditions F_i.*

An execution $w = k_0 k_1 k_2 \ldots \in \Sigma^\omega$ is accepted by $\mathcal{G}$ if there exists an infinite path $(s_0, K_0, F_0, s_1)(s_1, K_1, F_1, s_2)(s_2, K_2, F_2, s_3) \ldots \in R^\omega$ where:

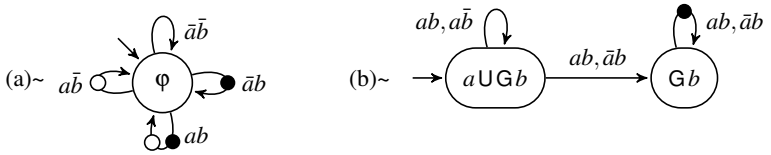


Fig. 1. (a) A TGBA with acceptance conditions $F = \{\bullet, \circ\}$ recognizing the LTL property $\varphi = \mathsf{G}\mathsf{F}a \wedge \mathsf{G}\mathsf{F}b$. (b) A TGBA with $F = \{\bullet\}$ recognizing the LTL property $a\mathsf{U}\mathsf{G}b$.

- $s_0 \in I$, *and* $\forall i \in \mathbb{N}, k_i \in K_i$ *(the execution is recognized by the path),*
- $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ *(each acceptance condition is visited infinitely often).*

The language accepted by $\mathcal{G}$ *is the set* $\mathscr{L}(\mathcal{G}) \subseteq \Sigma^\omega$ *of executions it accepts.*

Any LTL formula φ can be converted into a TGBA whose language is the set of executions that satisfy φ. Several algorithms exist to translate an LTL formula into a TGBA~[4, 9, 13, 1].

Figure~1 shows two examples of TGBA: one deterministic TGBA derived from the LTL formula $\mathsf{G}\mathsf{F}a \wedge \mathsf{G}\mathsf{F}b$, and one non-deterministic TGBA derived from $a\mathsf{U}\mathsf{G}b$. The LTL formulæ that label states 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 can be inferred from Fig.~1(a), an LTL formula such as $\bigwedge_{i=1}^{n} \mathsf{G}\mathsf{F}p_i$ can be represented by a one-state deterministic TGBA with n acceptance conditions.

Any infinite path in these examples is accepted if it visits all acceptance conditions (represented by colored dots on the transitions) infinitely often.

Testing a TGBA for emptiness amounts to the search of a strongly connected component that contains at least one occurrence of each acceptance condition. This can be done in different ways~[5]. We are using Couvreur's SCC-based emptiness check algorithm~[4] because it needs to traverse the state-space only once, and its complexity does not depend on the number of acceptance conditions. This algorithm is detailed in Appendix~B.

2.2 Büchi Automata

Compared to TGBA, the more traditional Büchi Automata (BA) have only one state-based acceptance condition.

One common way to obtain a BA from an LTL formula is to first translate the formula into some Generalized Büchi Automata with multiple acceptance conditions (it could be a TGBA~[13, 9] or a state-based GBA~[12]) and then to *degeneralize* this automaton to obtain a single acceptance condition.

Definition 4 *A BA over the alphabet* $\Sigma = 2^{AP}$ *is a tuple* $\mathcal{B} = \langle S, I, R, F \rangle$ *where:*

- S *is a finite set states,*
- $I \subseteq S$ *is the set of initial states,*
- $F \subseteq S$ *is a finite set of acceptance states,*
- $R \subseteq S \times 2^\Sigma \times S$ *is the transition relation where each transition is labeled by a set of valuations.*

An execution $w = k_0 k_1 k_2 \ldots \in \Sigma^\omega$ *is accepted by* $\mathcal{B}$ *if there exists an infinite path* $(s_0, K_0, s_1)(s_1, K_1, s_2)(s_2, K_2, s_3)\ldots \in R^\omega$ *such that:*

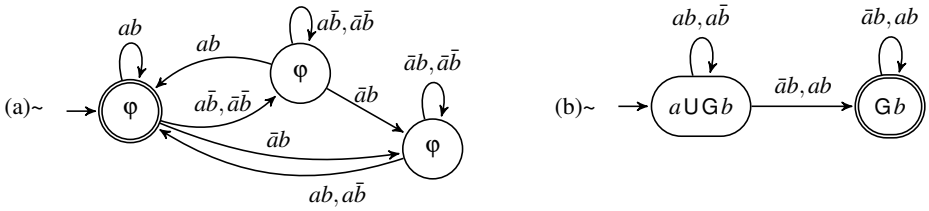


Fig. 2. Two examples of BA, with acceptance states shown as double circles. (a) A BA for the LTL property $\varphi = \mathsf{GF}a \wedge \mathsf{GF}b$ obtained by degeneralizing the TGBA for Fig.~1(a). (b) A BA for the LTL property $a\mathsf{U}\mathsf{G}b$.

- $s_0 \in I$, *and* $\forall i \in \mathbb{N}, k_i \in K_i$ *(the execution is recognized by the path),*
- $\forall i \in \mathbb{N}, \exists j \geq i, s_j \in F$ *(at least one acceptance state is visited infinitely often).*

The language accepted by $\mathcal{B}$ *is the set* $\mathscr{L}(\mathcal{B}) \subseteq \Sigma^\omega$ *of executions it accepts.*

Theorem 2 *TGBA and BA have the same expressive power: any TGBA can be converted into a language-equivalent BA and vice-versa*~[4, 13].

The process of converting a TGBA into a BA is called~*degeneralization*. In the worst case, a TGBA with s states and n acceptance conditions will be degeneralized into a BA with $s \times (n+1)$ states.

Figure~2 shows the same properties as~Fig.~1, but expressed as Büchi automata. The automaton from Fig.~2(a) was built by degeneralizing the TGBA from Fig.~1(a). The worst case of the degeneralization occurred here, since the TGBA with 1 state and n acceptance conditions was degeneralized into a BA with $n+1$ states. It is known that no BA with less than $n+1$ states can accept the property $\bigwedge_{i=1}^{n} \mathsf{GF}\,p_i$ so this Büchi automaton is optimal~[3]. The property $a\mathsf{U}\mathsf{G}b$, on the right hand side of the figure, is easier to express: the BA has the same size as the TGBA.

In the other way, a BA can be seen as a TGBA, by simply marking transitions leaving acceptance states as accepting, without adding states nor transitions. Algorithms that input TGBA can therefore be easily adapted to process BA. More importantly, BA can be checked for emptiness using the same one-pass emptiness-check algorithm.

2.3 Testing Automata

Testing Automata (TA) were introduced by Hansen et~al.~[14] to represent stuttering-insensitive properties. While a Büchi automaton observes the value of the atomic propositions AP, the basic idea of TA is to detect the *changes* in these values; if a valuation of AP does not change between two consecutive valuations of an execution, the TA can stay in the same state. To detect infinite executions that end stuck in the same TA state because they are stuttering, a new kind of acceptance states is introduced: *livelock-acceptance 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., $a\bar{b} \oplus ab = \{b\}$). Technically, this is implemented with an XOR operation (also denoted by the symbol $\oplus$).

Definition 5 *A TA over the alphabet* $\Sigma = 2^{AP}$ *is a tuple* $\mathcal{T} = \langle S, I, U, R, F, G \rangle$. *where:*

– *S is a finite set of states,*
– *$I \subseteq S$ is the 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),*
– *$R \subseteq S \times \Sigma \times S$ is the transition relation where each transition (s,k,d) is labeled by a* changeset*: $k \in \Sigma$ is interpreted as a (possibly empty) set of atomic propositions whose value must change between states s and d,*
– *$F \subseteq S$ is a set of Büchi-acceptance states,*
– *$G \subseteq S$ is a set of livelock-acceptance states.*

An execution $w = k_0 k_1 k_2 \ldots \in \Sigma^{\omega}$ is accepted by $\mathcal{T}$ if there exists an infinite sequence $(s_0, k_0 \oplus k_1, s_1)(s_1, k_1 \oplus k_2, s_2)\ldots(s_i, k_i \oplus k_{i+1}, s_{i+1})\ldots \in (S \times \Sigma \times S)^{\omega}$ such that:

– *$s_0 \in I$ with $k_0 \in U(s_0)$,*
– *$\forall i \in \mathbb{N}$, either $(s_i, k_i \oplus k_{i+1}, s_{i+1}) \in R$ (the execution progresses in the TA), or $k_i = k_{i+1} \wedge s_i = s_{i+1}$ (the execution is stuttering and the TA does not progress),*
– *Either, $\forall i \in \mathbb{N}, (\exists j \geq i, k_j \neq k_{j+1}) \wedge (\exists l \geq i, s_l \in F)$ (the TA is progressing in a Büchi-accepting way), or, $\exists n \in \mathbb{N}, (s_n \in G \wedge (\forall i \geq n, s_i = s_n \wedge k_i = k_n))$ (the sequence reaches a livelock-acceptance state and then stays on that state because the execution is stuttering).*

The language accepted by $\mathcal{T}$ is the set $\mathscr{L}(\mathcal{T}) \subseteq \Sigma^{\omega}$ of executions it accepts.

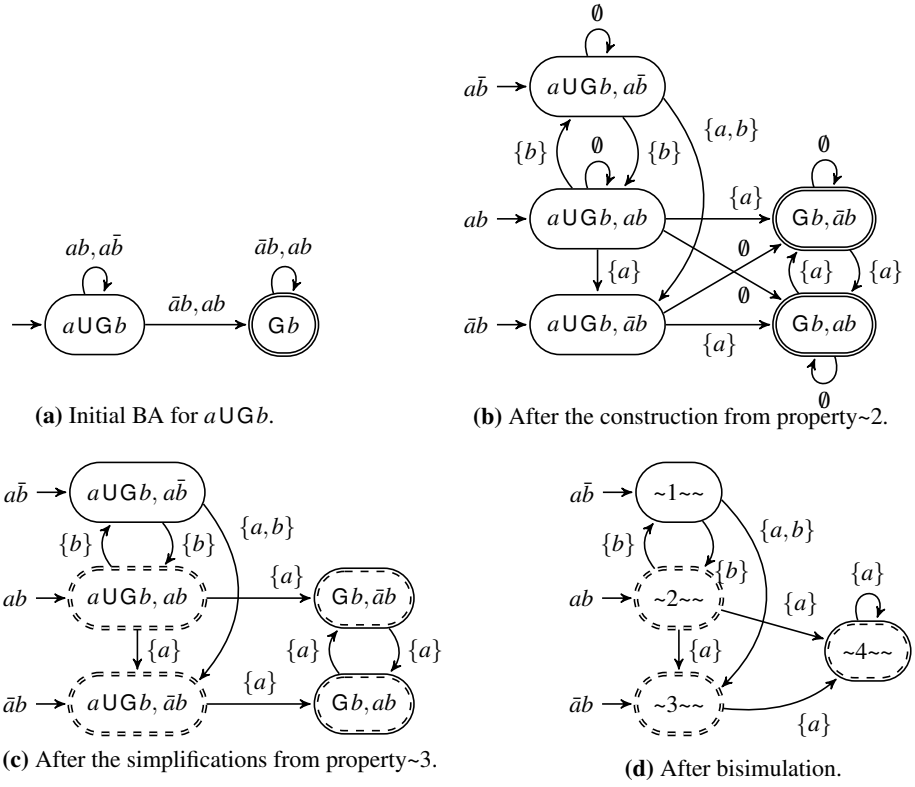
To illustrate this definition, consider Fig.~3d, representing a TA for $a \,\mathsf{U}\, \mathsf{G}\, b$.

– The execution $ab;\bar{a}b;ab;\bar{a}b;ab;\bar{a}b;ab;\ldots$ is Büchi accepting. A run recognizing such an execution 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 run state $4 \in F$ infinitely often, so they are Büchi accepting.
– The execution $ab;\bar{a}b;\bar{a}b;\bar{a}b;\ldots$ is livelock accepting. 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 $3 \in G$.
– The execution $ab;a\bar{b};ab;a\bar{b};ab;a\bar{b};\ldots$ 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 F state) nor livelock-accepting (it passes through state $2 \in G$, but does not stay into this state continuously).

Property 1 *The language accepted by a testing automaton is stuttering-insensitive.*

Proof. This follows from definition of accepted executions: a TA may not change its state when an execution stutters, so stuttering is always possible. □

Construction of a Testing Automaton from a Büchi Automaton Geldenhuys and Hansen~[10] have shown how to convert a BA into a TA by first converting the BA into an automaton with valuations on the states, and then converting this automaton into a TA by computing the difference between the labels of the source and destination of each transition. The next proposition implements these first two steps at once.



(a) Initial BA for $a \mathsf{U} \mathsf{G} b$.

(b) After the construction from property~2.

(c) After the simplifications from property~3.

(d) After bisimulation.

Fig. 3. Steps of the construction of a TA from a BA. States with a double enclosure belong to either F or G: states in $F \setminus G$ have a double plain line, states in $G \setminus F$ have a double dashed line, and states in $F \cap G$ use a mixed dashed/plain style.

Property 2 (Converting a BA into a TA~[10]) *For any BA $\mathcal{B} = \langle S_\mathcal{B}, I_\mathcal{B}, R_\mathcal{B}, F_\mathcal{B} \rangle$ over the alphabet $\Sigma = 2^{AP}$ and such that $\mathscr{L}(\mathcal{B})$ is stuttering insensitive, let us define the TA $\mathcal{T} = \langle S_\mathcal{T}, I_\mathcal{T}, U_\mathcal{T}, R_\mathcal{T}, F_\mathcal{T}, \emptyset \rangle$ with $S_\mathcal{T} = S_\mathcal{B} \times \Sigma$, $I_\mathcal{T} = I_\mathcal{B} \times \Sigma$, $F_\mathcal{T} = F_\mathcal{B} \times \Sigma$ and*

- $\forall (s,k) \in I_\mathcal{T}, U_\mathcal{T}((s,k)) = \{k\}$
- $\forall (s,k) \in S_\mathcal{T}, \forall (s',k') \in S_\mathcal{T}$,
  $((s,k), k \oplus k', (s',k')) \in R_\mathcal{T} \iff \exists K \in 2^\Sigma, ((s,K,s') \in R_\mathcal{B}) \wedge (k \in K)$

Then $\mathscr{L}(\mathcal{B}) = \mathscr{L}(\mathcal{T})$.

Figure~3b shows the result of applying this construction to the example Büchi automaton shown for $a \mathsf{U} \mathsf{G} b$. This testing automaton does not yet use livelock-acceptance states (the G set). The next property, again from Geldenhuys and Hansen~[10], shows how filling G allows to remove all transitions labeled by $\emptyset$.

Property 3 (Using G to simplify a TA~[10]) *Let $\mathcal{T} = \langle S, I, U, R, F, G \rangle$ be TA. By combining the first three of the following operations we can remove all transitions of the form $(s, \emptyset, s')$ (i.e. stuttering-transitions) from a TA. The fourth simplification can be performed along the way.*

1. *If $Q \subseteq S$ is a Strongly Connected Component (SCC) such that $Q \cap F \neq \emptyset$ (it is Büchi accepting), and any two states $q, q' \in Q$ can be connected using a non-empty sequence of stuttering-transitions $(q,\emptyset,q_1)\cdot(q_1,\emptyset,q_2)\cdots(q_n,\emptyset,q') \in R^*$, then the testing automaton $\mathcal{T}' = \langle S,I,U,F,G\cup Q\rangle$ is such that $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$. Such a component Q is called an **accepting Stuttering-SCC**.*
2. *If there exists a transition $(s,\emptyset,s') \in R$ such that $s' \in G$, the automaton $\mathcal{T}'' = \langle S,I,R\setminus\{(s,\emptyset,s')\},F,G\cup\{s\}\rangle$ is such that $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.*
3. *If $\mathcal{T}$ does not contain any accepting Stuttering-SCC, and there exists a transition $(s,\emptyset,s') \in R$ such that s' cannot reach any state from G using only transitions labeled by $\emptyset$, then these transitions can be removed. I.e., the automaton $\mathcal{T}''' = \langle S,I,R\setminus\{(s,\emptyset,s')\},F,G\rangle$ is such that $\mathscr{L}(\mathcal{T}''') = \mathscr{L}(\mathcal{T})$.*
4. *Any state from which one cannot reach a Büchi-accepting cycle nor a livelock-acceptance state can be removed without changing the automaton's language.*

The resulting TA can be further simplified by merging bisimilar states (two states are bisimilar if the automaton can accept the same executions starting for either of these states). This can be achieved using any algorithm based on partition refinement~[e.g., 27], taking $\{F\cap G, F\setminus G, G\setminus F, S\setminus(F\cup G)\}$ as initial partition.

Fig.~3 shows how a BA denoting the LTL formula $a\,\mathsf{U}\,\mathsf{G}\,b$ is transformed into a TA by applying prop.~2, prop.~3, and finally merging bisimilar states.

A TA for $\mathsf{GF}a \wedge \mathsf{GF}b$ is too big to be shown: even after simplifications it has 11 states and 64 transitions.

An unfortunate consequence of having two different ways of accepting executions (livelock or Büchi), is that the emptiness-check algorithm required during model checking must perform two passes on the whole state space in the worst case. Geldenhuys and Hansen~[10] have devised a heuristic that often renders the second pass useless when the formula is violated. Another optimization we present in Appendix~D is to omit the second pass when no livelock-accepting states is encountered during the first pass.

3 Transition-based Generalized Testing Automata

This section introduces a new kind of automaton that combines features from both TA and TGBA. From TA, we take the idea of labeling transitions with changesets, however we remove the use of livelock-acceptance (because it may require a two-pass emptiness check), and the implicit stuttering. From TGBA, we inherit the use of transition-based generalized acceptance conditions.

The resulting Chimera, which we call *Transition-based Generalized Testing Automaton* (TGTA), accepts only stuttering-insensitive languages like TA, and inherits advantages from both TA and TGBA: it has a simple one-pass emptiness-check procedure (the same as the one for TGBA), and can benefit from reductions based on the stuttering of the properties pretty much like a TA. Livelock acceptance states, which are no longer supported, can be emulated using states with a Büchi accepting self-loop labeled by $\emptyset$.

Definition 6 *A TGTA over the alphabet Σ is a tuple $\mathcal{T} = \langle S,I,U,R,F\rangle$ where:*
- *S is a finite set of states,*

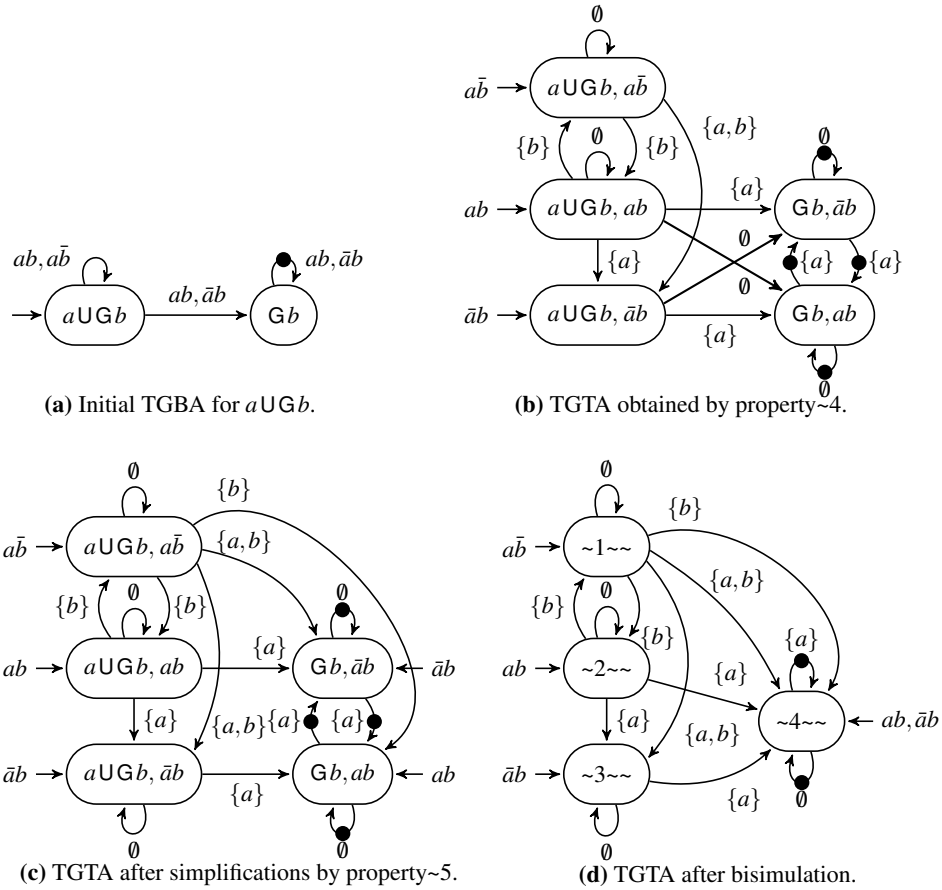


Fig. 4. TGTA obtained after various steps while translating the TGBA representing $a\,\mathsf{U}\,\mathsf{G}\,b$, into a TGTA with $F = \{\bullet\}$.

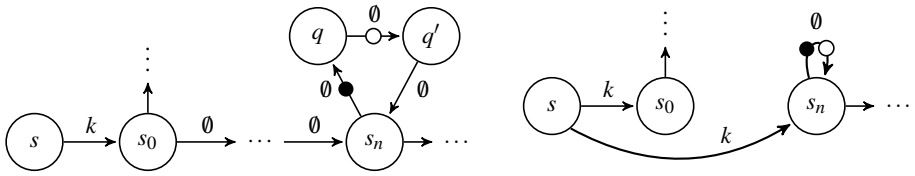
- *$I \subseteq S$ is the set of initial states,*
- *$U : I \to 2^{\Sigma}$ is a function mapping each initial state to a set of symbols of Σ*
- *F is a finite set of acceptance conditions,*
- *$R \subseteq S \times \Sigma \times 2^{F} \times S$ is the transition relation, where each element (s_i, k_i, F_i, d_i) represents a transition from state s_i to state d_i labeled by a* changeset *k_i interpreted as a (possibly empty) set of atomic propositions whose value must change between states s_i and d_i, and the set of acceptance conditions F_i.*

An execution $w = k_0 k_1 k_2 \ldots \in \Sigma^{\omega}$ is accepted by $\mathcal{T}$ if there exists an infinite path $(s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2)(s_2, k_2 \oplus k_3, F_2, s_3) \ldots \in R^{\omega}$ where:

- *$s_0 \in I$ with $k_0 \in U(s_0)$ (the execution is recognized by the path),*
- *$\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i,\ f \in F_j$ (each acceptance condition is visited infinitely often).*

The language accepted by $\mathcal{T}$ is the set $\mathscr{L}(\mathcal{T}) \subseteq \Sigma^{\omega}$ of executions it accepts.

Figure~4d shows a TGTA constructed for $a\,\mathsf{U}\,\mathsf{G}\,b$ in the same way as we did for Fig.~3d. The only accepting runs are those that see $\bullet$ infinitely often. The reader can



(a) Before stuttering-transitions ($\emptyset$) reduction. **(b)** After reduction.

Fig. 5. Using stuttering-transitions to simplify a TGTA.

verify that all the executions taken as example in section~2.3 are still accepted, but not always with the same runs (for instance $ab;\bar{a}b;\bar{a}b;\bar{a}b;\ldots$ is accepted by the run $2,4,4,4,\ldots$, but not by the run $2,3,3,3,\ldots$). This small difference is due to the way we emulate livelock-accepting states, as we shall describe later (in Property ~5).

Construction of a TGTA from a TGBA We now describe how to build a TGTA starting from a TGBA. The construction is inspired by the one presented in section~2.3 to construct a TA from a BA. In future work we plan to implement a direct translation from LTL to TGTA, but the construction presented below is enough to show the benefits of using TGTAs, and makes it easier to understand how TGTAs relates from TGBAs.

Our first property is the counertpart of Prop.~2: we can construct a TGTA from a TGBA by moving labels to states, and labeling each transition by the set difference between the labels of its source and destination states. While doing so, we keep the generalized acceptance conditions on the transitions. An example is shown on Fig~4b.

Property 4 (Converting TGBA into TGTA) *For any TGBA $\mathcal{G} = \langle S_\mathcal{G}, I_\mathcal{G}, R_\mathcal{G}, F\rangle$ over the alphabet $\Sigma = 2^{AP}$ and such that $\mathscr{L}(\mathcal{G})$ is stuttering insensitive, let us define the TGTA $\mathcal{T} = \langle S_\mathcal{T}, I_\mathcal{T}, U_\mathcal{T}, R_\mathcal{T}, F\rangle$ with $S_\mathcal{T} = S_\mathcal{G} \times \Sigma$, $I_\mathcal{T} = I_\mathcal{G} \times \Sigma$ and*
(i) $\forall (s,k) \in I_\mathcal{T}, U_\mathcal{T}((s,k)) = \{k\}$
(ii) $\forall (s,k) \in S_\mathcal{T}, \forall (s',k') \in S_\mathcal{T}$,
$((s,k), k \oplus k', F_1, (s',k')) \in R_\mathcal{T} \iff \exists K \in 2^\Sigma, ((s,K,F_1,s') \in R_\mathcal{G}) \wedge (k \in K)$
Then $\mathscr{L}(\mathcal{G}) = \mathscr{L}(\mathcal{T})$. (See appendix~E for a proof.)

The next property is the pendent of Prop.~3 to simplify the automaton by removing stuttering-transitions. Here we cannot remove self-loop transitions labeled by $\emptyset$, but we can remove all others. The intuition behind this simplification is illustrated on Fig~5a: s_0 is reachable from state s by a non-stuttering transition, but s_0 can reach an accepting stuttering-cycle by following only stuttering transitions. In the context of TA we would have to declare s_0 as being a livelock-accepting state. For TGTA, we replace the accepting stuttering-cycle by adding a self-loop labeled by all acceptance conditions on s_n, then the predecessors of s_0 are connected to s_n as in Fig.~5b.

Property 5 (Using stuttering-transitions to simplify a TGTA) *Let $\mathcal{T} = \langle S, I, U, R, F\rangle$ be TGTA such that $\mathscr{L}(\mathcal{T})$ is stuttering insensitive. By combining the first three of the following operations, we can remove all stuttering-transitions that are not self-loop (see Fig.~5). The fourth simplification can be performed along the way.*

1. *If $Q \subseteq S$ is a SCC such that any two states $q, q' \in Q$ can be connected using a sequence of stuttering-transitions $(q,\emptyset,F_0,q_1)(q_1,\emptyset,F_1,q_2)\cdots(q_n,\emptyset,F_n,q') \in R^*$ with $F_0 \cup F_1 \cup \cdots \cup F_n = F$, then we can add an accepting stuttering self-loop $(q,\emptyset,F,q)$ on each state $q \in Q$. I.e., the TGTA $\mathcal{T}' = \langle S, I, U, R \cup \{(q,\emptyset,F,q) \mid q \in Q\}, F\rangle$ is such that $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$. Let us call such a component Q an **accepting Stuttering-SCC**.*
2. *If there exists an accepting Stuttering-SCC Q and a sequence of stuttering-transitions $(s_0,\emptyset,F_1,s_1)(s_1,\emptyset,F_2,s_2)\cdots(s_{n-1},\emptyset,F_n,s_n) \in R^*$ such that $s_n \in Q$ and $s_0, s_1, \ldots s_{n-1} \notin Q$ (Fig.~5a), then:*
   - *For any non-stuttering transition, $(s,k,f,s_0) \in R$ going to s_0 and such that $k \neq \emptyset$, the TGTA $\mathcal{T}'' = \langle S, I, U, R \cup \{(s,k,f,s_n)\}, F\rangle$ is such that $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.*
   - *If $s_0 \in I$, the TGTA $\mathcal{T}'' = \langle S, I \cup \{s_n\}, U'', R, F\rangle$ with $\forall s \neq s_n$, $U''(s) = U(s)$ and $U''(s_n) = U(s_n) \cup U(s_0)$, is such that $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.*
3. *Let $\mathcal{T}^\dagger = \langle S, I^\dagger, U^\dagger, R^\dagger, F\rangle$ be the TGTA obtained after repeating the previous two operations as much as possible (i.e., $\mathcal{T}^\dagger$ contains all the transitions and initial states that can be added by the above two operations). Then, we can add non-accepting stuttering self-loops $(s,\emptyset,\emptyset,s)$ to all states that did not have an accepting stuttering self-loop because $\mathcal{T}$ describes a stuttering invariant property. Also we can remove all stuttering-transitions that are not self-loops since stuttering can be captured by self-loops after the previous two operations. More formally, the automaton $\mathcal{T}''' = \langle S, I^\dagger, U^\dagger, R''', F\rangle$ with $R''' = \{(s,k,f,d) \in R^\dagger \mid k \neq \emptyset \vee (s = d \wedge f = F)\} \cup \{(s,\emptyset,\emptyset,s) \mid (s,\emptyset,F,s) \notin R^\dagger\}$ is such that $\mathscr{L}(\mathcal{T}''') = \mathscr{L}(\mathcal{T}^\dagger) = \mathscr{L}(\mathcal{T})$.*
4. *Any state from which one cannot reach a Büchi-accepting cycle can be removed from the automaton without changing its language.* (See appendix~E for proofs.)

Here again, an additional optimization is to merge bisimilar states, this can be achieved using the same algorithm used to simplify a TA, taking S as initial partition and taking into account the acceptance conditions of the outgoing transitions. All these steps are shown on Fig.~4.

We can think of a TGTA as a TGBA whose transitions are labeled by changesets instead of atomic proposition valuations. When checking a TGBA for emptiness, we are looking for an accepting cycle that is reachable from an initial state. When checking a TGTA for emptiness, we are looking exactly for the same thing. The same emptiness check algorithm can be used, because emptiness check algorithms do not look at transition labels.

This is a nice feature of TGTA, not only because it gives us a one-pass emptiness check, but also because it eases the implementation of the TGTA approach in a TGBA-based model checker. We need only to implement the conversion of TGBA to TGTA and the product between a TGTA and a Kripke structure. We discuss our implementation in the next section.

4 Experimentation

This section presents our experimentation of the various types of automata within our tool Spot~[17]. We first present the Spot architecture and the way the variation on the model checking algorithm was introduced. Then we present our benchmarks (formulæ and models) prior to the description of our experiments.

4.1 Implementation on top of Spot

Spot is a model-checking library offering several algorithms that can be combined to build a model checker~[7]. Figure~6 shows the building blocks we used to implement the three approaches.

One point that we did not discuss so far is that in the automata-theoretic approach, the automaton used to represent the property to check has to be synchronized with a Kripke structure representing the model. Depending on the kind of automaton (TGBA, BA, TA, TGTA), this *synchronized product* has to be defined differently. Only the TGBA and BA approaches can share the same product definition. The definitions of these different products follow naturally from the definition of the runs on each automata. We refer the reader to Appendix~A for a definition of all these products.

The TGBA, BA, and TGTA approaches share the same emptiness check, while a dedicated algorithm is required by the TA approach. In Fig.~6, no direct translation is provided from LTL to TGTA (this is also true for BA and TA). This could be investigated later, the need being, so far, to assess their interest before optimizing the translation process.

In order to evaluate our approach on "realistic" models, we decided to couple the Spot library with the CheckPN tool~[7]. CheckPN implements Spot's Kripke structure interface in order to build the state space of a Petri net on the fly. This Kripke structure is then synchronized with an ω-automaton (TGBA, BA, TA or TGTA) on the fly, and fed to the suitable emptiness check algorithm. The latter algorithm drives the on-the-fly construction: only the explored part of the product (and the associated states of the Kripke structure) will be constructed.

Constructing the state space on-the-fly is a double-edged optimization. Firstly, it saves memory, because the state-space is computed as it is explored and thus, does not need be stored. Secondly, it also saves time when a property is violated because the emptiness check can stop as soon as it has found a counterexample. However, on-the-fly exploration is costlier than browsing an explicit graph: an emptiness check algorithm such as the one for TA that does two traversals of the full state-space in the worst case (e.g. when the property holds) will pay twice the price of that construction.

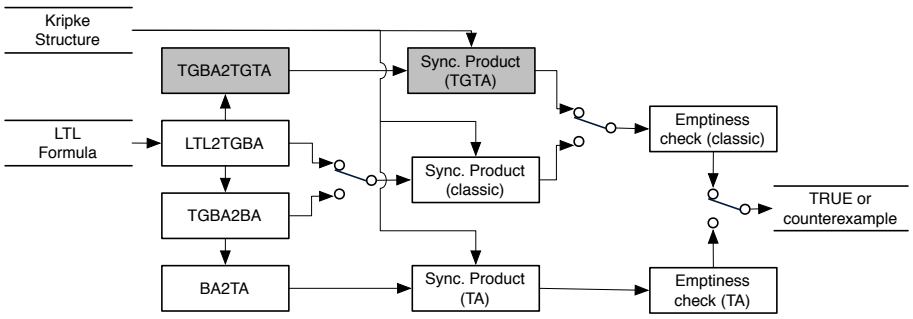


Fig. 6. The experiment's architecture in SPOT. Three command-line switches control which one of the approaches is used to verify an LTL formula on a Kripke structure. The new components required by the TGTA approach are outlined in Gray.

In the CheckPN implementation of the Kripke structure, the Petri Net marking are compressed to save memory. The marking of a state has to be uncompressed every time we compute its successors, or when we compute the value of the atomic properties on this state. These two operations often occur together, so there is a one-entry cache that prevents the marking from being uncompressed twice in a row.

4.2 Benchmark Inputs

We selected some Petri net models and formulæ to compare these approaches.

Case Studies The following two bigger models, were taken from actual cases studies. They come with some *dedicated* properties to check.

PolyORB models the core of the μbroker component of a middleware~[16] in an implementation using a Leader/Followers policy~[21]. It is a Symmetric Net and, since CheckPN processes P/T nets only, it was unfolded into a P/T net. The resulting net, for a configuration involving three sources of data, three simultaneous jobs and two threads (one leader, one follower) is composed of 189 places and 461 transitions. Its state space contains 61 662 states[3]. The authors propose to check that once a job is issued from a source, it must be processed by a thread (no starvation). It corresponds to:

$$\Phi_1 = \mathsf{G}(MSrc_1 \rightarrow \mathsf{F}(DOSrc_1)) \wedge \mathsf{G}(MSrc_2 \rightarrow \mathsf{F}(DOSrc_2)) \wedge \mathsf{G}(MSrc_3 \rightarrow \mathsf{F}(DOSrc_3))$$

MAPK models a biochemical reaction: Mitogen-activated protein kinase cascade~[15]. For a scaling value of 8 (that influences the number of tokens in the initial marking), it contains 22 places and 30 transitions. Its state space contains 6.11×10^6 states. The authors propose to check that from the initial state, it is necessary to pass through states *RafP*, *MEKP*, *MEKPP* and *ERKP* in order to reach *ERKPP*. In LTL:

$$\begin{aligned}\Phi_2 = \neg((\neg RafP)\,\mathsf{U}\,MEKP) \wedge \neg((\neg MEKP)\,\mathsf{U}\,MEKPP) \wedge \\ \neg((\neg MEKPP)\,\mathsf{U}\,ERKP) \wedge \neg((\neg ERKP)\,\mathsf{U}\,ERKPP)\end{aligned}$$

Toy Examples A first class of four models were selected from the Petri net literature~[2, 20]: the flexible manufacturing system (FMS), the Kanban system, the Peterson algorithm, and the slotted-ring system. All these models have a parameter n. For the Peterson algorithm, 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 chose values for n in order to get state space having between 2×10^5 to 3×10^6 nodes except for Peterson that is 6.3×10^8 nodes. The objective is to have non trivial state spaces to be synchronized.

Types of Formulæ As suggested by Geldenhuys and Hansen~[10], the type of formula may affect the performances of the various algorithms. In addition to the formulæ Φ_1 and Φ_2 above, we consider two classes of formulæ:

[3] This is a rather small value compared to MAPK but, due to the unfolding, each state is a 189-value vector. PolyORB with three sources of data, three simultaneous jobs and three threads would generate 1 137 096 states with 255-value vectors, making the experiment much too slow.

– *RND*: randomly generated LTL formulæ (without X operator). Since random formulæ are very often trivial to verify (the emptiness check needs to explore only a handful of states), for each model we selected only random formulæ that required to explore more than 2000 states with the three approaches.
– *WFair*: properties of the form $(\bigwedge_{i=1}^{n} \mathsf{GF}\, p_i) \rightarrow \varphi$, where φ is a randomly generated LTL formula. This represents the verification of φ under the weak-fairness hypothesis $\bigwedge_{i=1}^{n} \mathsf{GF}\, p_i$. The automaton representing such a formula has at least n acceptance conditions which means that the BA will in the worst case be $n+1$ times bigger than the TGBA. For the formulæ we generated for our experiments we have $n \approx 3.23$ on the average.

All formulæ were translated into automata using Spot, which was shown experimentally to be very good at this job~[22, 6]. The time spent doing the conversion from LTL to TGBA and then to TGTA (bisimulation included) is not measured in this benchmark. This translation process is almost instantaneous (<0.1s), and even if its runtime could be improved (for instance with a direct translation from LTL to TGTA) it is clearly a non significant part of the run time of the different model checking approaches, where all the time is spent performing the emptiness check of the product (built on-the-fly) between the Kripke structure and the property automaton.

4.3 Results

Table~1 shows how for TGBA, TA and TGTA approaches deal with toy models and random formulæ. We omit data for BA since they are always outperformed by TGBA. For space reason, we also omit the table showing toy models against weak-fairness formulæ~[23], because it shows results similar to those of table~1.

Table~2 shows the results of the two cases studies against random, weak-fairness, and dedicated formulæissued from the studies.

These tables separate cases where formulæ are verified from cases where they are violated. In the former (left sides of the tables), no counterexample are found and the full state space had to be explored; in the latter (right sides) the on-the-fly exploration of the state space stopped as soon as the existence of a counterexample could be computed.

All values shown in all tables are averaged over 100 different formulas (except for the lines Φ_1 and Φ_2 in Table~2, where only one formula is used). For instance we checked Peterson5 against 100 random formulæ that had no counterexample, and against 100 random formulæ that had a counterexample. The average and maximum are computed separately on these two sets of formulæ.

Column-wise, these tables show the average and maximum sizes (states and transitions) of: (1) the automata $A_{\neg\varphi_i}$ expressing the properties φ_i; (2) the products $A_{\neg\varphi_i} \otimes A_M$ of the property with the model; and (3) the subset of this product that was actually explored by the emptiness check. For verified properties, the emptiness check of TGBA and BA always explores the full product so these sizes are equal, while the emptiness check of TA always performs two passes on the full product so it shows double values. On violated properties, the emptiness check aborts as soon as it finds a counterexample, so the explored size is usually significantly smaller than the full product.

The emptiness check values show a third column labeled "T": this is the time (in hundredth of seconds, a.k.a. centiseconds) spent doing that emptiness check, includ-

			Verified properties (no counterexample)							Violated properties (a counterexample exists)							
			Automa.		Full product		Emptiness check			Automa.		Full product		Emptiness check			
			st.	tr.	st.	tr.	st.	tr.	T	st.	tr.	st.	tr.	st.	tr.	T	
Peterson5	TGBA	avg	6	74	2 489 651	10 706 255	2 489 651	10 706 255	18 623	6	98	4 378 750	18 950 326	447 532	1 664 897	3 429	**Peterson5**
		max	15	368	8 648 954	51 231 247	8 648 954	51 231 247	66 305	19	452	15 758 813	68 376 074	3 464 205	16 063 131	33 813	
	TA	avg	21	333	2 364 394	9 208 893	3 820 489	14 883 673	29 299	23	400	4 530 886	17 667 955	477 568	1 709 098	3 626	
		max	86	2 976	6 794 050	26 853 438	13 588 100	53 706 876	102 304	96	2 384	15 665 643	61 047 330	4 356 432	16 974 570	33 646	
	TGTA	avg	18	384	**2 327 423**	**9 070 953**	**2 327 423**	**9 070 953**	**17 675**	20	473	**4 338 504**	**16 958 774**	**421 538**	**1 504 616**	**3 270**	
		max	63	2 541	6 802 901	26 886 243	6 802 901	26 886 243	50 992	89	3 931	15 665 643	61 102 382	3 171 366	12 373 040	27 590	
Ring6	TGBA	avg	6	62	671 254	4 279 403	671 254	4 279 403	986	7	96	1 534 755	10 264 597	247 215	1 257 494	349	**Ring6**
		max	18	360	2 788 267	27 749 420	2 788 267	27 749 420	5 348	25	347	5 378 284	33 010 296	1 882 843	12 472 859	3 613	
	TA	avg	18	213	**525 472**	**3 040 856**	1 020 062	5 908 454	1 686	24	366	1 410 442	8 308 311	191 563	1 054 280	379	
		max	61	1 396	2 218 432	14 265 568	4 436 864	28 531 136	7 513	79	2 453	4 591 370	26 900 864	1 583 757	9 167 989	3 849	
	TGTA	avg	16	266	538 121	3 168 068	**538 121**	**3 168 068**	**892**	21	443	**1 349 758**	**8 084 652**	**176 745**	**979 121**	**323**	
		max	57	1 874	2 260 160	14 894 816	2 260 160	14 894 816	3 799	73	2 257	3 864 570	22 690 780	1 380 951	8 353 228	2 935	
FMS5	TGBA	avg	6	68	551 742	3 993 810	551 742	3 993 810	814	5	69	6 883 843	62 106 759	54 518	214 399	55	**FMS5**
		max	21	462	7 302 624	65 806 572	7 302 624	65 806 572	12 288	16	280	23 726 488	244 007 764	1 921 058	11 918 449	2 391	
	TA	avg	15	186	**405 588**	**3 161 894**	482 687	3 786 904	976	17	251	6 686 607	55 303 413	31 632	165 756	52	
		max	51	931	6 570 291	53 742 054	9 018 660	74 274 828	19 268	53	1 615	20 819 154	179 323 867	1 356 842	10 590 558	2 814	
	TGTA	avg	13	231	407 130	3 184 980	**407 130**	**3 184 980**	**788**	15	305	**6 397 882**	**53 832 859**	**30 372**	**155 899**	**46**	
		max	37	1 147	6 570 291	53 742 054	6 570 291	53 742 054	12 522	52	1 852	19 040 412	167 542 101	1 356 842	10 590 558	2 551	
Kanban5	TGBA	avg	5	48	837 446	8 350 628	837 446	8 350 628	1 254	6	66	5 974 582	61 036 827	45 690	165 487	31	**Kanban5**
		max	12	238	9 500 904	135 774 933	9 500 904	135 774 933	18 934	20	268	16 729 695	220 725 750	915 486	4 055 483	860	
	TA	avg	13	133	575 668	5 450 687	575 668	5 450 687	1 099	20	225	5 659 936	54 667 538	21 839	104 243	25	
		max	55	1 170	5 970 944	60 234 328	5 970 944	60 234 328	11 567	66	1 096	16 091 600	161 286 146	439 088	3 579 035	866	
	TGTA	avg	11	166	**560 409**	**5 302 142**	**560 409**	**5 302 142**	**1 015**	17	265	**5 153 380**	**50 559 085**	**19 503**	**97 543**	**22**	
		max	51	1 343	5 970 944	60 234 328	5 970 944	60 234 328	11 013	58	1 246	13 940 304	144 001 788	439 056	3 578 982	819	

Table~1. Comparison of the three approaches on toy examples with random formulæ, when counterexamples do not exist (left) or when they do (right).

				Automaton		Full product		Emptiness check			Automaton		Full product		Emptiness check		
				st.	tr.	st.	tr.	st.	tr.	T	st.	tr.	st.	tr.	st.	tr.	T
PolyORB 3/3/2	RND	TGBA	avg	13	438	69 141	182 821	69 141	182 821	592	10	438	**100 245**	243 147	50 886	122 212	436
			max	55	2 852	341 835	986 034	341 835	986 034	3 115	52	2 136	411 157	1 076 730	227 290	848 061	2 087
		TA	avg	83	4 043	**64 086**	**138 821**	104 415	226 731	895	53	3 474	103 035	222 697	52 455	117 020	465
			max	411	46 769	295 389	653 208	590 778	1 306 416	5 350	469	62 602	336 393	726 570	221 833	478 119	2 136
		TGTA	avg	67	4 121	65 108	141 407	**65 108**	**141 407**	**558**	43	3 910	102 151	**221 808**	**46 907**	**104 919**	**409**
			max	334	47 567	317 676	703 265	317 676	703 265	2 731	393	66 517	363 075	775 320	193 890	503 543	1 677
	WFair	TGBA	avg	4	38	65 276	153 015	65 276	153 015	509	4	36	73 483	166 933	57 295	130 841	437
			max	12	128	184 986	791 568	184 986	791 568	1 713	12	130	193 995	578 037	135 785	401 864	1 169
		TA	avg	55	641	79 602	170 480	83 446	178 828	630	51	593	155 237	333 349	91 714	201 664	718
			max	164	2 102	184 986	397 458	269 934	603 126	2 384	195	3 162	379 629	826 305	197 846	433 800	1 724
		TGTA	avg	21	264	**62 469**	**133 825**	**62 469**	**133 825**	**480**	20	243	**71 784**	**153 927**	**55 984**	**122 560**	**434**
			max	52	681	184 986	397 458	184 986	397 458	1 618	52	906	184 986	398 520	116 737	253 999	1 014
	Φ_1	TGBA	–	7	576	345 241	760 491	**345 241**	760 491	**1 921**							
		TA	–	80	14 590	**342 613**	**742 815**	685 226	1 485 630	3 858							
		TGTA	–	79	17 153	345 277	753 798	345 277	**753 798**	1 925							
MAPK 8	RND	TGBA	avg	5	60	1 095 515	11 940 964	1 095 515	11 940 964	2 211	5	62	15 364 530	214 264 214	44 621	147 170	48
			max	14	256	11 408 161	147 987 445	11 408 161	147 987 445	26 963	14	278	57 275 770	785 403 825	1 657 316	7 858 321	2 860
		TA	avg	14	173	812 843	10 119 485	812 843	10 119 485	2 228	15	192	15 531 840	202 637 377	22 071	53 569	22
			max	48	969	9 793 932	125 681 958	9 793 932	125 681 958	27 197	55	916	44 955 300	579 348 912	318 571	832 689	345
		TGTA	avg	12	226	**812 456**	**10 116 066**	**812 456**	**10 116 066**	**2 173**	13	236	**14 360 794**	**189 226 133**	**21 080**	**51 231**	**19**
			max	47	1 304	9 793 932	125 681 958	9 793 932	125 681 958	26 208	44	1 141	43 481 872	569 374 833	318 570	832 687	308
	WFair	TGBA	avg	3	20	2 649 730	30 385 763	2 649 730	30 385 763	5 602	4	33	9 711 621	125 560 263	43 651	**195 459**	**47**
			max	8	84	16 962 176	268 267 694	16 962 176	268 267 694	45 042	12	136	33 023 267	477 330 020	2 248 882	13 302 369	3 148
		TA	avg	31	296	1 779 991	21 870 089	1 779 991	21 870 089	5 033	44	471	20 285 910	260 574 207	33 160	201 268	58
			max	103	1 270	13 254 682	172 100 144	13 254 682	172 100 144	43 217	131	1 786	44 841 684	585 288 808	1 130 354	12 156 893	3 039
		TGTA	avg	13	131	**1 684 025**	**20 831 377**	**1 684 025**	**20 831 377**	**4 660**	18	208	**8 596 802**	**110 988 663**	**31 661**	198 070	52
			max	43	591	13 254 682	172 100 144	13 254 682	172 100 144	37 989	60	871	21 684 220	284 699 450	1 129 783	12 155 921	2 849
	Φ_2	TGBA	–	6	165	46 494	302 350	46 494	302 350	53							
		TA	–	9	293	33 376	289 235	33 376	289 235	55							
		TGTA	–	8	452	**33 376**	**289 235**	**33 376**	**289 235**	**53**							

Table~2. Comparison of the three approaches for the case studies when counterexamples do not exist (left) or when they do (right).

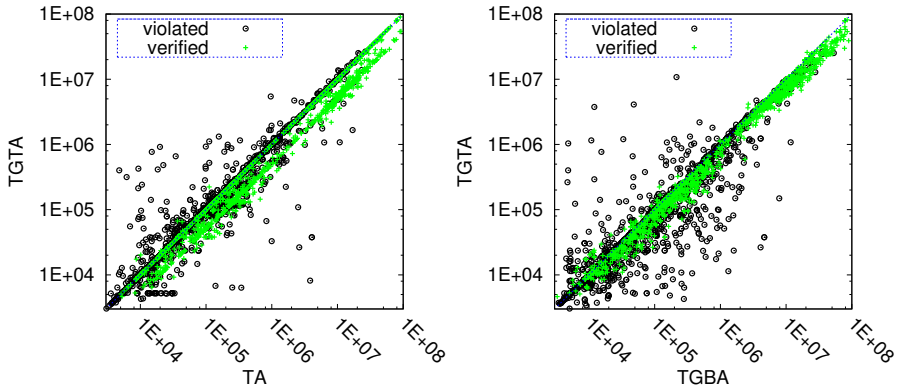


Fig. 7. Performance (transitions explored by the emptiness check) of TGTA against TA and TGBA.

ing the on-the-fly computation of the subset of the product that is explored. The time spent constructing the property automata from the formulæ is not shown (it is negligible compared to that of the emptiness check).

Figure~7 compares the number of visited transitions when running the emptiness check; plotting TGTA against TA and TGBA. This gives an idea of their relative performance. Each point corresponds to one of the 4100 evaluated formulas (2050 violated with counterexample as black circles, and 2050 verified having no counterexample as green crosses). Each point below the diagonal is in favor of TGTA while others are in favor of the other approach. Axes are displayed using a logarithmic scale. No comparison is presented with BA since they are less efficient than TGBA~[23].

All these tests were run on a 64bit Linux system running on an Intel Core 2 Quad Processor Q9400 at 2.66GHz, with 4GB of RAM.

5 Discussion

Although the state space of cases studies can be very different from random state spaces~[18], a first look at our results confirms two facts already observed in previous studies~[10]: (1) although the TA constructed from properties are usually a lot larger than TGBA (and even larger than BA~[23]), the average size of the full product is smaller thanks to the more deterministic nature of the TA. (2) For violated properties, the TA approach explores less states and transitions on the average than TGBA or BA.

We complete this picture by showing run times, by separating verified properties from violated properties, and by also evaluating the TGBA approach.

It should be noted that our implementation has been improved since our previous experiments~[23] where the cost of computing labels in the Kripke structure was higher than it is now (we use a cache). This change mainly benefit to testing automata, because they query two labels by transition of the Kripke structure (to compute an xor between source label and destination label) while other approaches query only one label.

For weak-fairness formulæ, we show only the results for cases studies because for toy examples we obtain similar results as random formulæ.

On verified properties the results are very straightforward to interpret when looking at the number of transitions explored by the emptiness check. TA outperform TGBA except for both Random and weak-fairness properties against Peterson, Ring and Poly-ORB. These are typical cases where the TA emptiness check has to perform two passes: this can be observed in the tables~1 and~2 when the number of transitions visited by the emptiness check is on the average twice the number of transitions of the product.

In these three cases, the TGTA approach, with its single-pass emptiness check, is a clear improvement over TA. On the left scatter plots of Fig.~7, these cases where the TGTA approach is twice faster than TA's, appear as a linear cloud of green crosses below the diagonal (because the axes are displayed using a logarithmic scale).

In the other where TA need only one pass on the average (e.g. Kanban, MAPK), TGTA and TA have similar performance, with a slight advantage for TGTA because the products are smaller.

As a consequence the TGTA approach outperforms TGBA and TA in all cases on verified properties.

On violated properties, it is harder to interpret these tables because the emptiness check will return as soon as it finds a counterexample. Changing the order in which non-deterministic transitions of the property automaton are iterated is enough to change the number of states and transitions to be explored before a counterexample is found: in the best case the transition order will lead the emptiness check straight to an accepting cycle; in the worst case, the algorithm will explore the whole product until it finally finds an accepting cycle. Although the emptiness check algorithms for the three approaches share the same routines to explore the automaton, they are all applied to different kinds of property automata, and thus provide different transition orders.

We believe that the TA and TGTA, since they are more deterministic~[10], are less sensitive to this ordering. Also, in all of our experiments the TA approach has always found the counterexample in the first pass of the emptiness check algorithm. This supports Geldenhuys and Hansen's claim that the second pass was seldom needed for violated properties (less than 0.005% of the cases in their experiments~[10]). Finally, in the tables~1 and~2, we observe that the TGTA approach explores the smallest products on the average.

6 Conclusion

This paper is the sequel of a preliminary work~[23] experimenting LTL model checking of stuttering-insensitive properties with various techniques: Büchi automata (BA), Transition-based Generalized Büchi Automata and Testing Automata~[10]. At this time, conclusions were that TA outperformed BA and sometimes TGBA for unverified properties (i.e., when a counterexample was found). However, this was not the case when no counterexample was computed since the entire state space may 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 by proposing a new type of ω-automaton: Transition-based Generalized Testing Automata (TGTA). It inherits from TA the labeling of transitions by changesets and, from TGBA, the use of transition-based acceptance conditions. The idea is to combine advantages observed on both TA and TGBA.

TGTA have been implemented in Spot easily, because only two new algorithms are required: the conversion of a TGBA into a TGTA, and a new definition of a product between a TGTA and a Kripke structure.

We have run benchmarks to assess their interest. Experiments reported that, in most cases, TGTA outperform TA and TGBA when no counterexample is found in the system and are comparable when the property is violated.

We conclude that there is nothing to lose by using TGTA to verify stuttering-insensitive properties, since they are always at least as good as TA and TGBA.

Future work We plan additional work to enable symbolic model checking with TGTA, thus allowing us to tackle much larger state spaces than in explicit model checking. Another idea would be to provide a direct conversion of LTL to TGTA, without the intermediate TGBA step. We believe a tableau construction such as the one of Couvreur~[4] could be easily adapted to produce TGTA.

References

[1] Babiak, T., Křetínský, M., Řeehák, V., Strejček, J.: LTL to Büchi automata translation: Fast and more deterministic. In: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12). Lecture Notes in Computer Science, vol. 7214, pp. 95–109. Springer (2012)

[2] Ciardo, G., Lüttgen, G., Siminiceanu, R.: Efficient symbolic state-space construction for asynchronous systems. In: Nielsen, M., Simpson, D. (eds.) Proceedings of the 21st International Conference on Application and Theory of Petri Nets (ICATPN'00). Lecture Notes in Computer Science, vol. 1825, pp. 103–122. Springer-Verlag (2000)

[3] Cichoń, J., Czubak, A., Jasiński, A.: Minimal Büchi automata for certain classes of LTL formulas. In: Proceedings of the Fourth International Conference on Dependability of Computer Systems (DEPCOS'09). pp. 17–24. IEEE Computer Society (2009)

[4] Couvreur, J.M.: On-the-fly verification of temporal logic. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99). Lecture Notes in Computer Science, vol. 1708, pp. 253–271. Springer-Verlag, Toulouse, France (Sep 1999)

[5] Couvreur, J.M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized Büchi automata. In: Godefroid, P. (ed.) Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05). Lecture Notes in Computer Science, vol. 3639, pp. 143–158. Springer (Aug 2005)

[6] Duret-Lutz, A.: LTL translation improvements in Spot. In: Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'11). Electronic Workshops in Computing, British Computer Society, Tunis, Tunisia (Sep 2011), http://ewic.bcs.org/category/15853

[7] Duret-Lutz, A., Poitrenaud, D.: SPOT: an extensible model checking library using transition-based generalized Büchi automata. In: Proceedings of the 12th

IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04). pp. 76–83. IEEE Computer Society Press, Volendam, The Netherlands (Oct 2004)

[8] Farwer, B.: Automata logics, and infinite games, Lecture Notes in Computer Science, vol. 2500, chap. ω-automata, pp. 3–21. Springer-Verlag (2002), http://dl.acm.org/citation.cfm?id=938135.938137

[9] Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01). Lecture Notes in Computer Science, vol. 2102, pp. 53–65. Springer-Verlag, Paris, France (2001)

[10] Geldenhuys, J., Hansen, H.: Larger automata and less work for LTL model checking. In: Proceedings of the 13th International SPIN Workshop (SPIN'06). Lecture Notes in Computer Science, vol. 3925, pp. 53–70. Springer (2006)

[11] Geldenhuys, J., Valmari, A.: Tarjan's algorithm makes on-the-fly LTL verification more efficient. In: Jensen, K., Podelski, A. (eds.) Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04). Lecture Notes in Computer Science, vol. 2988, pp. 205–219. Springer (2004)

[12] Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of the 15th Workshop on Protocol Specification Testing and Verification (PSTV'95). pp. 3–18. Chapman & Hall, Warsaw, Poland (1996), http://citeseer.nj.nec.com/gerth95simple.html

[13] Giannakopoulou, D., Lerda, F.: From states to transitions: Improving translation of LTL formulæ to Büchi automata. In: Peled, D., Vardi, M. (eds.) Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02). Lecture Notes in Computer Science, vol. 2529, pp. 308–326. Houston, Texas (Nov 2002)

[14] Hansen, H., Penczek, W., Valmari, A.: Stuttering-insensitive automata for on-the-fly detection of livelock properties. In: Cleaveland, R., Garavel, H. (eds.) Proceedings of the 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02). Electronic Notes in Theoretical Computer Science, vol. 66(2). Elsevier, Málaga, Spain (Jul 2002)

[15] Heiner, M., Gilbert, D., Donaldson, R.: Petri nets for systems and synthetic biology. In: Proceedings of the 8th International School on Formal Methods for the Design of Computer (SFM'08). Lecture Notes in Computer Science, vol. 5016, pp. 215–264. Springer (2008)

[16] Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Barrir, S., Vergnaud, T.: On the formal verification of middleware behavioral properties. In: Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04). Electronic Notes in Theoretical Computer Science, vol. 133, pp. 139–157. Elsevier Science Publishers (Sep 2004)

[17] MoVe/LRDE: The Spot home page: http://spot.lip6.fr (2012)

[18] Pelánek, R.: Properties of state spaces and their applications. International Journal on Software Tools for Technology Transfer (STTT) 10(5), 443–454 (2008)

[19] Peled, D., Wilke, T.: Stutter-invariant temporal properties are expressible without the next-time operator. Information Processing Letters 63(5), 243–246 (Sep 1995)
[20] Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)
[21] Pyarali, I., Spivak, M., Cytron, R., Schmidt, D.C.: Evaluating and optimizing thread pool strategies for RT-CORBA. In: Proceeding of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES'00). pp. 214–222. ACM (2000)
[22] Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'07). Lecture Notes in Computer Science, vol. 4595, pp. 149–167. Springer (2007)
[23] Salem, A.E.B., Duret-Lutz, A., Kordon, F.: Generalized Büchi automata versus testing automata for model checking. In: Procedings of the 2nd workshop on Scalable and Usable Model Checking for Petri Nets and other models of Concurrency (SUMo'11). vol. 726, pp. 65–79. CEUR, Newcastle, UK (June 2011)
[24] Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L. (eds.) Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05). Lecture Notes in Computer Science, vol. 3440. Springer, Edinburgh, Scotland (Apr 2005)
[25] Sebastiani, R., Tonetta, S.: "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In: Goos, G., Hartmanis, J., van Leeuwen, J. (eds.) Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03). Lecture Notes in Computer Science, vol. 2860, pp. 126–140. Springer-Verlag, L'Aquila, Italy (Oct 2003)
[26] Tauriainen, H.: Automata and Linear Temporal Logic: Translation with Transition-based Acceptance. Ph.D. thesis, Helsinki University of Technology, Espoo, Finland (Sep 2006)
[27] Valmari, A.: Bisimilarity minimization in $O(m \log n)$ time. In: Proceedings of the 30th International Conference on the Applications and Theory of Petri Nets (ICATPN'09). Lecture Notes in Computer Science, vol. 5606, pp. 123–142. Springer (2009), http://dx.doi.org/10.1007/978-3-642-02424-5_9
[28] Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G.M. (eds.) Proceedings of the 8th Banff Higher Order Workshop (Banff'94). Lecture Notes in Computer Science, vol. 1043, pp. 238–266. Springer-Verlag, Banff, Alberta, Canada (1996)

A Product Definitions

A.1 Product of TGBA (or BA) with a Kripke Structure

The product of a TGBA with a Kripke structure is a TGBA whose language is the intersection of both languages.

Definition 7 *For a Kripke structure $\mathcal{K} = \langle S_\mathcal{K}, I_\mathcal{K}, R_\mathcal{K}, L_\mathcal{K}\rangle$ and a TGBA $\mathcal{G} = \langle S_\mathcal{G}, I_\mathcal{G}, R_\mathcal{G}, F_\mathcal{G}\rangle$ the product $\mathcal{K}\otimes\mathcal{G}$ is the TGBA $\langle S, I, R, F\rangle$ where*

- $S = S_\mathcal{K} \times S_\mathcal{G}$,
- $I = I_\mathcal{K} \times I_\mathcal{G}$,
- $R = \{((s,s'), L_\mathcal{K}(s), f, (d,d')) \mid (s,d) \in R_\mathcal{K}, (s',k,f,d') \in R_\mathcal{G}, L_\mathcal{K}(s) \in k\}$
- $F = F_\mathcal{G}$.

Property 6 *We have $\mathscr{L}(\mathcal{K}\otimes\mathcal{G}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{G})$ by construction.*

Since a BA can be seen as a TGBA with a unique acceptance set, and all state-based acceptance conditions pushed to the outgoing transitions, the same construction can be used to make a product between a Kripke structure and a BA.

A.2 Product of a TA with a Kripke Structure

For TGBA (or BA) the synchronized product with a Kripke structure can be defined as another TGBA (or BA). In the case of testing automata, the product of a Kripke and a TA is not a TA: while an execution in a TA is allowed to stutter on any state, the execution in a product must always progress.

Definition 8 *For a Kripke structure $\mathcal{K} = \langle S_\mathcal{K}, I_\mathcal{K}, R_\mathcal{K}, L_\mathcal{K}\rangle$ and a TA $\mathcal{T} = \langle S_\mathcal{T}, I_\mathcal{T}, U_\mathcal{T}, R_\mathcal{T}, F_\mathcal{T}, G_\mathcal{T}\rangle$, the product $\mathcal{K}\otimes\mathcal{T}$ is an automaton $\langle S, I, U, R, F, G\rangle$ where*

- $S = S_\mathcal{K} \times S_\mathcal{T}$,
- $I = \{(s,s') \in I_\mathcal{K} \times I_\mathcal{T} \mid L_\mathcal{K}(s) \in U_\mathcal{T}(s')\}$,
- $\forall (s,s') \in I, U((s,s')) = \{L_\mathcal{K}(s)\}$,
- $R = \{((s,s'),k,(d,d')) \mid (s,d) \in R_\mathcal{K}, (s',k,d') \in R_\mathcal{T}, k = L_\mathcal{K}(s) \oplus L_\mathcal{K}(d)\}$
  $\cup \{((s,s'),\emptyset,(d,d')) \mid (s,d) \in R_\mathcal{K}, s' = d', L_\mathcal{K}(s) = L_\mathcal{K}(d)\}$
- $F = S_\mathcal{K} \times F_\mathcal{T}$, *and* $G = S_\mathcal{K} \times G_\mathcal{T}$.

An execution $w = k_0k_1k_2\ldots \in K^\omega$ is accepted by $\mathcal{K}\otimes\mathcal{T}$ if there exists an infinite sequence $(s_0, k_0\oplus k_1, s_1)(s_1, k_1\oplus k_2, s_2)\ldots(s_i, k_i\oplus k_{i+1}, s_{i+1})\ldots \in (S\times K\times S)^\omega$ such that:

- $s_0 \in I$ *with* $k_0 \in U(s_0)$,
- $\forall i \in \mathbb{N}, (s_i, k_i\oplus k_{i+1}, s_{i+1}) \in R$ *(we are always progressing in the product)*
- *Either,* $\forall i \in \mathbb{N}, (\exists j \geq i, k_j \neq k_{j+1}) \wedge (\exists l \geq i, s_l \in F)$ *(the automaton is progressing in a Büchi-accepting way), or,* $\exists n \in \mathbb{N}, \forall i \geq n, (k_i = k_n) \wedge (s_i \in G)$ *(a suffix of the execution stutters in G).*

Property 7 *We have $\mathscr{L}(\mathcal{K}\otimes\mathcal{T}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{T})$ by construction.*

A.3 Product of a TGTA with a Kripke Structure

The product of a TGTA with a Kripke structure is a TGTA.

Comparing this definition with the previous two products shows the double inheritance of TGTA. This product is similar to the product between a TA and a Kripke structure, expect it does not deal with livelock acceptance states and implicit stuttering. It is also similar to the product of a TGBA with a Kripke structure, except for the use of changesets on transitions, and the initial labels (U).

Definition 9 *For a Kripke structure $\mathcal{K} = \langle S_\mathcal{K}, I_\mathcal{K}, R_\mathcal{K}, L_\mathcal{K} \rangle$ and a TGTA $\mathcal{T} = \langle S_\mathcal{T}, I_\mathcal{T}, U_\mathcal{T}, R_\mathcal{T}, F_\mathcal{T} \rangle$, the product $\mathcal{K} \otimes \mathcal{T}$ is a TGTA $\langle S, I, U, R, F \rangle$ where*

- $S = S_\mathcal{K} \times S_\mathcal{T}$,
- $I = \{(s,s') \in I_\mathcal{K} \times I_\mathcal{T} \mid L_\mathcal{K}(s) \in U_\mathcal{T}(s')\}$,
- $\forall (s,s') \in I, U((s,s')) = \{L_\mathcal{K}(s)\}$,
- $R = \{((s,s'),k,f,(d,d')) \mid (s,d) \in R_\mathcal{K}, (s',k,f,d') \in R_\mathcal{T}, k = L_\mathcal{K}(s) \oplus L_\mathcal{K}(d)\}$
- $F = F_\mathcal{T}$.

Property 8 *We have $\mathscr{L}(\mathcal{K} \otimes \mathcal{T}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{T})$ by construction.*

B Model Checking Using TGBA

When doing model checking with TGBA the two important operations are the translation of the linear-time property φ into a TGBA $A_{\neg\varphi}$ and the emptiness check of the product of the Kripke structure $\mathcal{K}$ with $A_{\neg\varphi}$: this product $\mathcal{K} \otimes A_{\neg\varphi}$ is a TGBA. Numerous algorithms translate LTL formulæ into TGBA~[13, 4, 1, 26]. We use Couvreur's one~[4] with some optimizations~[6].

Testing a TGBA for emptiness amounts to the search of a strongly connected component that contains at least one occurrence of each acceptance condition. It can be done in two different ways: either with a variation of Tarjan or Dijkstra algorithm~[4] or using several nested depth-first searches to save memory~[26]. The latter proved to be slower~[5], so we are using Couvreur's SCC-based emptiness check algorithm~[4]. Another advantage of the SCC-based algorithm is that their complexity does not depend on the number of acceptance conditions.

Algorithm.~1 presents an iterative version of Couvreur's algorithm~[4]. This algorithm computes on the fly the maximal Strongly Connected Components: 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 union of all acceptance conditions occurring in the merged SCC is equal to F, then an accepting run is found. *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). Figure~9 illustrates a run of this algorithm on a small example.

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 all acceptance conditions belonging to the SCC, *la* is the acceptance conditions of the transition between the previous and the current SCC, and *rem* contains the fully explored states of the SCC. Figure~8 shows how *acc* and *la* are used in the SCC search stack.

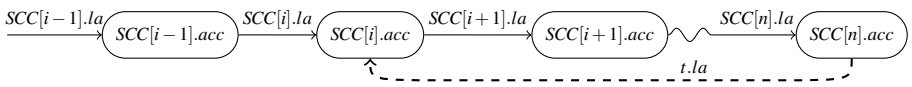


Fig. 8. *SCC* stack: the use of the SCCs fields *la* and *acc*.

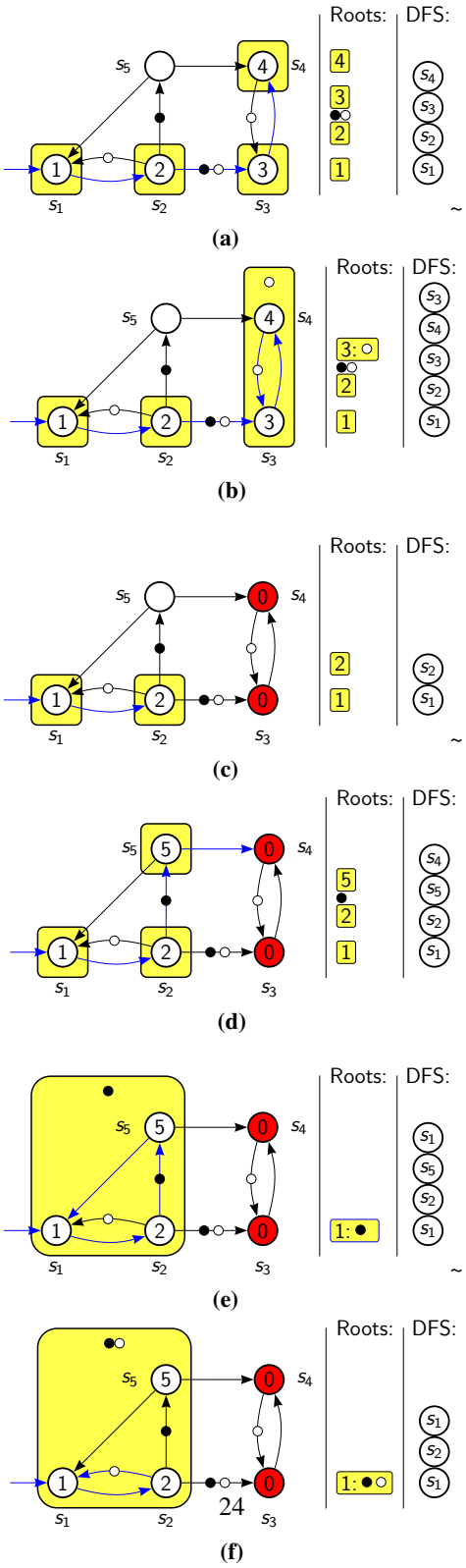


Fig. 9. Six intermediate steps in a run of algorithm~1. The states $s_1, \ldots, s_5$ are labeled by their value in H. The stack of roots of SCCs (the *root* stack in the algorithm) and the DFS search stack (induced by the recursive calls to `DFSpush()`) are displayed on the side. An interpretation of the SCC stack in term of SCCs is given as yellow blobs on the automaton.

(a) Initially the algorithm performs a DFS search by declaring each newly encountered

```
1  Input: A product TGBA G = ⟨S, I, R, F⟩
2  Result: true if and only if 𝒟(G) = ∅
3  Data: todo: stack of ⟨state ∈ S, succ ⊆ R⟩
         SCC: stack of
         ⟨root ∈ ℕ, la ⊆ F, acc ⊆ F, rem ⊆ S⟩
         H: map of S ↦ ℕ
         max ← 0
4  begin
5    foreach s⁰ ∈ I do
6      DFSpush(∅, s⁰)
7      while ¬todo.empty() do
8        if todo.top().succ = ∅ then
9          DFSpop()
10       else
11         pick one ⟨s, _, a, d⟩ off todo.top().succ
12         if d ∉ H then
13           DFSpush(a, d)
14         else if H[d] > 0 then
15           merge(a, H[d])
16           if SCC.top().acc = F then
17             return false
18   return true

19 DFSpush(la ⊆ F, s ∈ S)
20   max ← max + 1
21   H[s] ← max
22   SCC.push(⟨max, la, ∅, ∅⟩)
23   todo.push(⟨s, {⟨q, l, a, d⟩ ∈ R | q = s}⟩)
24 DFSpop()
25   ⟨s, _⟩ ← todo.pop()
26   SCC.top().rem.insert(s)
27   if H[s] = SCC.top().root then
28     foreach s ∈ SCC.top().rem do
29       H[s] ← 0
30     SCC.pop()
31 merge(la ⊆ F, t ∈ ℕ)
32   r ← ∅
33   acc ← la
34   while t < SCC.top().root do
35     acc ← acc ∪ SCC.top().acc
                 ∪ SCC.top().la
36     r ← r ∪ SCC.top().rem
37     SCC.pop()
38   SCC.top().acc ← SCC.top().acc ∪ acc
39   SCC.top().rem ← SCC.top().rem ∪ r
```

Algorithm 1: Emptiness check algorithm for TGBA.

The algorithm begins by pushing in *SCC* each state visited for the first time (line~12), as a trivial SCC with an empty *acc* set (line~22). Then, when the DFS explores a transition t between two states s and d, if d is in the SCC stack (line~14), therefore 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). All the SCCs between $SCC[i]$ and $SCC[n]$ are merged (line~15) into $SCC[i]$. The merge of acceptance conditions is illustrated by Fig.~8: a "back" transition t is found between $SCC[n]$ and $SCC[i]$, therefore the latest SCCs (from i to n) are merged. The acceptance conditions of the merged SCC is equal to the union of $SCC[i].acc \cup SCC[i+1].la \cup SCC[i+1].acc \cup \cdots \cup SCC[n].la \cup SCC[n].acc \cup t.la$. If this union is equal to F, then the merged SCC is accepting and the algorithm return *false* (line~17): the product is not empty.

C Model Checking Using BA

Since a BA can be seen as a TGBA by pushing acceptance conditions from states to outgoing transitions, the emptiness check from Algorithm.~1 also works. Other algorithms, specific to BA, are based on two nested depth-first searches. The comparison of these different emptiness checks raised many studies~[11, 24, 5], and for this work we only consider the SCC-based algorithm presented here.

D Emptiness Check Using TA

Testing Automata require a dedicated algorithm because there are two ways to detect an accepting cycle in the product:

- Büchi acceptance: a cycle containing at least one Büchi-acceptance state (F) and at least one non-stuttering transition (i.e., a transition (s,k,s') with $k \neq \emptyset$),
- livelock acceptance: a cycle composed only of stuttering transitions and livelock acceptance states (G).

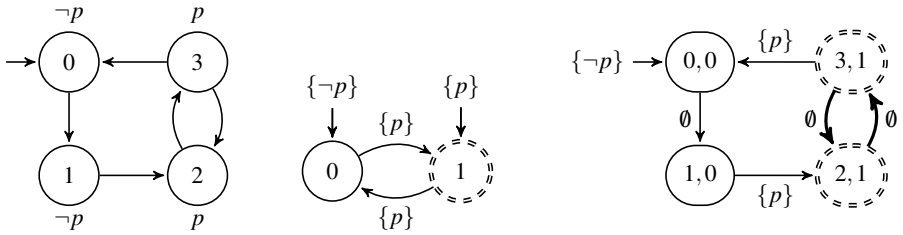
A straightforward emptiness check would have two passes: a first pass to detect Büchi acceptance cycles, it corresponds to Algorithm.~2 without the test at line~21 and a second pass presented in Algorithm.~3 to detect livelock acceptance cycles. It is not possible to merge these two passes into a single DFS: the first DFS requires the full product exploration while the second one must consider stuttering transitions only. These two passes are an inconvenient when the property is satisfied (no counterexample) since the entire state-space has to be explored twice.

```
1  Input: A product 𝒦 ⊗ 𝒯 = ⟨S, I, U, R, F, G⟩
2  Result: true if and only if ℒ(T) = ∅
3  Data: todo: stack of ⟨state ∈ S, succ ⊆ R⟩
         SCC: stack of ⟨root ∈ ℕ, lk ∈ 2^AP, k ∈
         2^AP, acc ⊆ F, rem ⊆ S⟩
         H: map of S ↦ ℕ
         max ← 0, Gseen ← false
4  begin
5    if ¬ first-pass() then return false
6    if Gseen then return second-pass()
7    return true
8  first-pass()
9    foreach s^0 ∈ I do
10     DFSpush1(∅, s^0)
11     while ¬todo.empty() do
12       if todo.top().succ = ∅ then
13         DFSpop()
14       else
15         pick one ⟨s, k, d⟩ off todo.top().succ
16         if d ∉ H then
17           DFSpush1(k, d)
18         else if H[d] > 0 then
19           merge1(k, H[d])
20           if (SCC.top().acc ≠ ∅) ∧
             (SCC.top().k ≠ ∅) then return
             false
21           if (d ∈ G) ∧ (SCC.top().k = ∅) then
             return false
22   return true

23 DFSpush1(lk ∈ 2^AP, s ∈ S)
24   max ← max + 1
25   H[s] ← max
26   if s ∈ F then
27     SCC.push(⟨max, lk, ∅, {s}, ∅⟩)
28   else
29     SCC.push(⟨max, lk, ∅, ∅, ∅⟩)
30   todo.push(⟨s, {⟨q, k, d⟩ ∈ R | q = s}⟩)
31   if s ∈ G then
32     Gseen ← true

33 merge1(lk ∈ 2^AP, t ∈ ℕ)
34   acc ← ∅
35   r ← ∅
36   k ← lk
37   while t < SCC.top().root do
38     acc ← acc ∪ SCC.top().acc
39     k ← k ∪ SCC.top().k ∪ SCC.top().lk
40     r ← r ∪ SCC.top().rem
41     SCC.pop()
42   SCC.top().acc ← SCC.top().acc ∪ acc
43   SCC.top().k ← SCC.top().k ∪ k
44   SCC.top().rem ← SCC.top().rem ∪ r
```

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



(a) A Kripke structure $\mathcal{K}$ **(b)** A TA $\mathcal{T}$ for $\mathsf{FG}\,p$ **(c)** $\mathcal{K} \otimes \mathcal{T}$

Fig. 10. Example product between a Kripke structure and a TA. The bold cycle is livelock-accepting.

With line~21 included in Algorithm.~2, the first-pass detects both Büchi and some livelock-acceptance cycles. Since in certain cases it may fail to report some livelock-acceptance cycles, a second pass is required to look for possible livelock-acceptance cycles.

This `first-pass` is based on the TGBA emptiness check algorithm presented in Algorithm.~1 with the following changes:

- In each item *scc* of the *SCC* stack: the field *scc.acc* contains the Büchi-accepting states detected in *scc*, *scc.lk* is analogous to *la* in Fig.~8 but it 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~39 and~43).
- After each merge, *SCC*.top() is checked for Büchi-acceptance (line~20) or livelock-acceptance (line~21) depending on the emptiness of *SCC*.top().*k*.

Figure~10 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 def.~8. In this example, $G_{\mathcal{T}} = \{1\}$ 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~19), 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~21 still return false because the union $SCC.top().k$ of all change-sets labeling the transitions of S is not empty (it includes for example t_1's label: $\{p\}$). The `first-pass` algorithm then terminates without reporting any accepting cycle, missing C_2.

Had the `first-pass` 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-acceptance cycles, ignoring the non-stuttering transitions that may belong to the same SCC.

```
1  Data: todo: stack of ⟨state ∈ S, succ ⊆ S⟩
         SCC: stack of ⟨root ∈ ℕ, rem ⊆ S⟩
         H: map of S ↦ ℕ
         max ← 0; init ← I
2  second-pass()
3    while ¬init.empty() do
4      pick one s⁰ off init
5      if s⁰ ∉ H then DFSpush2(∅, s⁰)
6      while ¬todo.empty() do
7        if todo.top().succ = ∅ then
8          DFSpop()
9        else
10         pick one d off todo.top().succ
11         if d ∉ H then
12           DFSpush2(d)
13         else if H[d] > 0 then
14           merge2(H[d])
15           if (d ∈ G) then return false
16   return true

17 DFSpush2(s ∈ S)
18   max ← max + 1
19   H[s] ← max
20   SCC.push(⟨max, ∅⟩)
21   todo.push(⟨s, {d ∈ S | (s, ∅, d) ∈ R}⟩)
22   init ← init ∪ {d ∈ S | (s, k, d) ∈ R, k ≠ ∅}

23 merge2(t ∈ ℕ)
24   r ← ∅
25   while t < SCC.top().root do
26     r ← r ∪ SCC.top().rem
27     SCC.pop()
28   SCC.top().rem ← SCC.top().rem ∪ r
```

Algorithm 3: The second-pass of the TA emptiness check algorithm.

The `second-pass` (Algorithm.~3) is a DFS exploring only stuttering transitions (line~21). To report a livelock-accepting cycle, it detects "stuttering-SCCs" and tests if they contain a livelock-accepting state (line~15).

Ignoring the non-stuttering transitions during the DFS, may lead to miss some parts of the product so any destination of a non stuttering transition is stored in *init* for later exploration (line~22).

In the algorithm proposed by Geldenhuys and Hansen~[10], the first pass uses a heuristic to detect livelock-acceptance cycles when possible. This heuristic detects more livelock-acceptance cycles than Algorithm.~2. In certain cases this first pass may still fail to report some livelock-acceptance cycles. Yet, this heuristic is very efficient: when counterexamples exist, they are usually caught by the first pass, and the second is rarely needed. However, when properties are verified, the second pass is always required.

Optimizations In our experimentation, we implement the algorithm proposed by Geldenhuys and Hansen~[10] including the heuristic and we have added some improvements to the first pass:

1. If no livelock-acceptance state is visited during the first pass, then the second pass can be disabled: this is the purpose of variable *Gseen*. In our experiments, this optimization greatly improves the performance of the TA approach in the cases where the formula is verified.
2. A cycle detected during the first pass is also accepted if it contains a livelock-acceptance state $(s_{\mathcal{K}}, s_{\mathcal{T}})$ such that $s_{\mathcal{T}}$ has no successor. Indeed, from this state, a run can only executes stuttering transitions. Therefore, a cycle containing this state, is composed only of stuttering transitions: it is a livelock accepting cycle.

E Proofs for TGTA Construction

Proof of property 4. ~
($\subseteq$) Let $w = k_0k_1k_2\ldots \in \mathscr{L}(\mathcal{G})$ be an execution accepted by $\mathcal{G}$. By Def.~3, this execution is recognized by a path $(s_0,K_0,F_0,s_2)(s_2,K_1,F_1,s_2)\ldots \in R_{\mathcal{G}}^{\omega}$ of $\mathcal{G}$, such that $s_0 \in I, \forall i \in \mathbb{N}, (k_i \in K_i)$, and $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i,\ f \in F_j$. By applying (ii) and (i), we can see that there exists a corresponding path $((s_0,k_0),k_0 \oplus k_1,F_0,(s_1,k_1))((s_1,k_1),k_1 \oplus k_2,F_1,(s_2,k_2))\ldots \in R_{\mathcal{T}}^{\omega}$ of $\mathcal{T}$ such that $(s_0,k_0) \in I_{\mathcal{T}}$, $k_0 \in U_{\mathcal{T}}((s_0,k_0))$, and still $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i,\ f \in F_j$. By Def.~6 we therefore have $w \in \mathscr{L}(\mathcal{T})$.
($\supseteq$) Let $w = w_0w_1w_2\ldots \in \mathscr{L}(\mathcal{T})$ be an execution accepted by $\mathcal{T}$. By Def.~6, this execution is recognized by a path $((s_0,k_0),w_0 \oplus w_1,F_0,(s_1,k_1))((s_1,k_1),w_1 \oplus w_2,F_1,(s_2,k_2))\ldots \in R_{\mathcal{T}}^{\omega}$ of $\mathcal{T}$ such that $(s_0,k_0) \in I_{\mathcal{T}}$, $w_0 \in U_{\mathcal{T}}((s_0,k_0))$, and $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i,\ f \in F_j$. Of course we have $w_i \oplus w_{i+1} = k_i \oplus k_{i+1}$ but this does not suffice to imply that $k_i = w_i$. However (i) tells us that $w_0 \in U_{\mathcal{T}}((s_0,k_0)) = \{k_0\}$ so $w_0 = k_0$, and since $w_i \oplus w_{i+1} = k_i \oplus k_{i+1}$ it follows that $w_i = k_i$. By applying (ii) can now find a corresponding path $(s_0,K_0,F_0,s_2)(s_2,K_1,F_1,s_2)\ldots \in R_{\mathcal{G}}^{\omega}$ of $\mathcal{G}$, such that $s_0 \in I, \forall i \in \mathbb{N}, (w_i = k_i \in K_i)$, and $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i,\ f \in F_j$. By Def.~3 we therefore have $w \in \mathscr{L}(\mathcal{G})$. □

Proof of property 5. ~

1. $(\mathcal{T}' \supseteq \mathcal{T})$ Obvious because we are only adding transitions. $(\mathcal{T}' \subseteq \mathcal{T})$ Let $R' = R \cup \{(q,\emptyset,F,q) \mid q \in Q\}$. Consider an accepting execution $w = k_0k_1k_2\ldots\mathscr{L}(\mathcal{T}')$ recognized by an accepting path π' on $\mathcal{T}'$. Any transition of π' that is not in R is a self-loop $(q,\emptyset,F,q)$ that has been added to R' because an accepting stuttering-SCC exists in R around q: so any $(q,\emptyset,F,q) \in R'$ can be replaced by a sequence of stuttering transitions $(q,\emptyset,G_0,q_1)(q_1,\emptyset,G_1,q_2)\ldots(q_n,\emptyset,G_n,q) \in R^*$ such that $G_0 \cup G_1 \cup \ldots G_n = F$. The path $\pi \in R^{\omega}$ obtained by replacing all such transitions is an accepting path of $\mathcal{T}$ that recognizes a word that is stuttering equivalent to w. Since $\mathscr{L}(\mathcal{T})$ is stuttering-insensitive, it must also contain w. □
2. $(\mathcal{T}'' \supseteq \mathcal{T})$ Obvious for the same reason. $(\mathcal{T}'' \subseteq \mathcal{T})$ We consider the case where s_0 is non initial (the initial case is similar). Let $R'' = R \cup \{(s,k,f,s_n)\}$. Consider an accepting execution $w = k_0k_1k_2\ldots\mathscr{L}(\mathcal{T}'')$ recognized by a path π'' on $\mathcal{T}''$. Let π be the path on $\mathcal{T}$ obtained by replacing in π'' any occurrence of $(s,k,f,s_n) \in (R'' \setminus R)$ by the sequence $(s,k,f,s_0)(s_0,\emptyset,F_1,s_1)(s_1,\emptyset,F_2,s_2)\cdots(s_{n-1},\emptyset,F_n,s_n) \in R^*$. The path $\pi \in R^{\omega}$ is also an accepting path of $\mathcal{T}$ that recognizes a word that is stuttering equivalent to w. Since $\mathscr{L}(\mathcal{T})$ is stuttering-insensitive, it must also contain w. □
3. $\mathscr{L}(\mathcal{T}^{\dagger}) = \mathscr{L}(\mathcal{T})$ by application of the previous two properties, therefore $\mathscr{L}(\mathcal{T}^{\dagger})$ is a stuttering-insensitive language. $\mathscr{L}(\mathcal{T}''')$ is also a stuttering-insensitive language because $\mathcal{T}'''$ is obtained from $\mathcal{T}^{\dagger}$ that recognizes a stuttering-insensitive language, by adding stuttering self-loops on all its states before removing all stuttering-transitions that are not self-loops.
   To prove that two stuttering-insensitive languages are equal, it is sufficient to verify that they contain the same words of the following two forms:
   - $w = k_0k_1k_2\ldots$ with $\forall i \in \mathbb{N}, k_i \oplus k_{i+1} \neq \emptyset$ (non-stuttering words), or
   - $w = k_0k_1k_2\ldots(k_n)^{\omega}$ with $\forall i < n, k_i \oplus k_{i+1} \neq \emptyset$ (terminal stuttering words)

All other accepted words can be generated by duplicating letters in the above words. Since we have only touched stuttering transitions, it is clear that the non-stuttering words of $\mathscr{L}(\mathcal{T})$ are the non-stuttering words of $\mathscr{L}(\mathcal{T}''')$.

We now consider the case of a terminal stuttering word $w = k_0 k_1 k_2 \dots (k_n)^\omega$ with $\forall i < n, k_i \oplus k_{i+1} \neq \emptyset$.

$(\mathcal{T}''' \subseteq \mathcal{T}^\dagger)$ The path π''' that recognizes w in $\mathcal{T}'''$ has the form $(s_0, k_0 \oplus k_1, F_0, s_1)$ $(s_1, k_1 \oplus k_2, F_1, s_2) \dots (s_n, \emptyset, F, s_n)^\omega$ where all transitions are necessarily from $\mathcal{T}^\dagger$ because we have only added in $\mathcal{T}'''$ transitions of the form $(s, \emptyset, \emptyset, s)$. π''' is thus also an accepting path of $\mathcal{T}^\dagger$ and $w \in \mathscr{L}(\mathcal{T}^\dagger)$.

$(\mathcal{T}''' \supseteq \mathcal{T}^\dagger)$ The path $\pi^\dagger$ that recognizes w in $\mathcal{T}^\dagger$ does only stutter after k_n. Because this is an accepting path, it has a lasso-shape, where the cyclic part is only stuttering and accepting. Let us denote it $\pi^\dagger = (s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2) \dots (s_{n-1}, k_{n-1} \oplus k_n, F_{n-1}, s_n)(s_n, \emptyset, F_n, s_{n+1}) \dots [(s_m, \emptyset, F_m, s_{m+1}) \dots (s_l, \emptyset, F_l, s_m)]^\omega$, with $\forall i < n, k_i \oplus k_{i+1} \neq \emptyset$.

Thanks to property 5.1, the accepting cycle $[(s_m, \emptyset, F_m, s_{m+1}) \dots (s_l, \emptyset, F_l, s_m)]$ of $\pi^\dagger$ can be replaced by an accepting self-loop $(s_m, \emptyset, F, s_m)$. And thanks to property 5.2, the transitions from s_{n-1} to s_m can be replaced by a single transition $(s_{n-1}, k_{n-1} \oplus k_n, F_{n-1}, s_m)$. The resulting path $\pi''' = (s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2) \dots (s_{n-1}, k_{n-1} \oplus k_n, F_{n-1}, s_m)(s_m, \emptyset, F, s_m)^\omega$ is an accepting path of $\mathcal{T}'''$ that accepts w, so $w \in \mathscr{L}(\mathcal{T}''')$. □

4. This is a classical optimization on Büchi automata. □