Generalized Büchi Automata versus Testing Automata for Model Checking

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 have shown that a kind of ω-automaton known as *testing automata* can outperform the Büchi automata traditionally used in the automata-theoretic approach to model checking [8]. This work completes their experiments by including a comparison with generalized Büchi automata; by using larger state spaces derived from Petri nets; and by distinguishing violated formulæ (for which testing automata fare better) from verified formulæ (where testing automata are hindered by their two-pass emptiness check).

1 Introduction

Context The automata-theoretic approach to model checking linear-time properties [23] 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, $\mathscr{L}(A_M)$, represent all possible executions of M.
2. Translation of the temporal property φ into a ω-automaton $A_{\neg\varphi}$ whose language, $\mathscr{L}(A_{\neg\varphi})$, is the set of all 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 our tools, 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 [3, 10].

Unfortunately, having a smaller property automaton $A_{\neg\varphi}$ does not always imply that the product with the model ($A_M \otimes A_{\neg\varphi}$) will be smaller, and it is the size of this product that really affects the efficiency of the model checking. Instead of targeting smaller property automata, some people have attempted to build automata that are *more deterministic* [21]; however even this does not guarantee the product to be smaller.

Hansen et al. [11] introduced a new kind of ω-automaton called *Testing Automaton*. 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). Also they are often a lot larger than their equivalent Büchi automaton, but surprisingly their good determinism often lead to a smaller product. The reasons why and the conditions under which testing automata perform better are still mysterious [8].

Objectives The objective of this paper is to evaluate efficiency of LTL model checking with these three kinds of ω-automata: classical Büchi Automata (BA), Transition-based Generalized Büchi automata (TGBA), and Testing Automata (TA). Our main motivation is to try to establish some rough rules to choose automatically and *a priori* the technique that seems most suitable to check a given *stuttering-insensitive* property on a given model. This is of interest when a tool offers the choice of several techniques, which is the case for our model checker Spot [16].

Contents Section 2 provides a brief summary of the three ω-automaton and pointers to their associated operations for model checking. Then section 3 reports our experimentation procedure and its results before a discussion in section 4.

2 Presentation of the three Approaches

Let *AP* designate the set of *atomic proposition* of the model that we might want to use to build a linear-time property. Any state of the model can be labeled by a valuation of these atomic propositions. We denote by $K = 2^{AP}$ the set of these valuations. For instance if $AP = \{a, b\}$, then $K = 2^{AP} = \{\bar{a}\bar{b}, \bar{a}b, a\bar{b}, ab\}$. An execution of the model is simply an infinite sequence of such valuations, i.e., an element from K^ω. A property can be seen as a set of sequences, i.e. a subset of K^ω.

This section presents the three kinds of automata we compare in this paper: Transitions-based Generalized Büchi Automata, Büchi Automata and Testing Automata. For all of them, we explain how they recognize subsets of K^ω to show their differences. We do not detail the actual operations that must be performed to model check a system which each approach because this has already been done in other works.

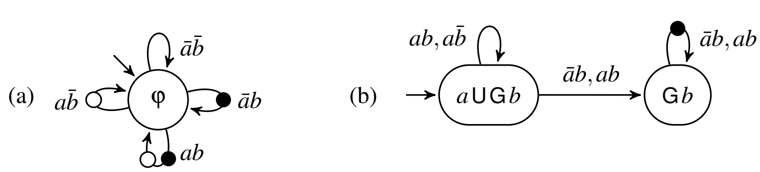


Fig. 1: (a) A TGBA with acceptance conditions $F = \{\bullet, \circ\}$ recognizing the LTL property $\varphi = \mathsf{GF}\,a \wedge \mathsf{GF}\,b$. (b) A TGBA with $F = \{\bullet\}$ recognizing the LTL property $a\,\mathsf{U}\,\mathsf{G}\,b$.

2.1 Transition-based Generalized Büchi Automata

A Transition-based Generalized Büchi Automata (TGBA) [10] over an alphabet $K = 2^{AP}$ is an ω-automaton where transitions are labeled by letters from K and some acceptance conditions. In our context, the TGBA represents the LTL property to verify.

Definition 1 *A TGBA can be formally represented by a tuple $G = \langle S, I, R, F\rangle$ where:*
- *S is finite set of states,*
- *$I \subseteq S$ is the set of initial states,*
- *F is a finite set of acceptance conditions,*
- *$R \subseteq S \times 2^K \times 2^F \times S$ 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 letters K_i, and the set of acceptance conditions F_i.*

An execution $w = k_0k_1k_2\ldots \in K^\omega$ is accepted by 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:
- *$s_0 \in I$, and $\forall i \in \mathbb{N}$, $k_i \in K_i \subseteq K$ (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).*

Fig. 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.

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 $A_M \otimes A_{\neg\varphi}$. We know of at least four algorithms that purposedly translate LTL formulæ into TGBA [10, 3, 4, 22]. The one we use is based on Couvreur's LTL translation algorithm [3].

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 way: either with a variation of Tarjan or Dijkstra algorithm [3] or using several nested depth-first searches to save some memory [22]. The latter proved to be slower [5], so we are using Couvreur's SCC-based emptiness check algorithm [3]. Another advantage of the SCC-based algorithm is that their complexity does not depend on the number of acceptance conditions.

2.2 Büchi Automata

A Büchi Automaton (BA) has only one acceptance condition that is state-based.

Definition 2 *A BA over the alphabet $K = 2^{AP}$ is a tuple $B = \langle S, I, R, F\rangle$ where:*
- *S is a set of finite set states,*
- *$I \subseteq S$ is the set of initial states,*
- *$F \subseteq S$ is a finite set of acceptance states,*
- *$R \subseteq S \times 2^K \times S$ is the transition relation where each transition is labeled by a set of letters of K.*

An execution $w = k_0k_1k_2\ldots \in K^\omega$ is accepted by 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:
- *$s_0 \in I$, 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).*

Model checking using BA A BA can be obtained from a TGBA by a procedure known as *degeneralization* [3, 10]. In a worst case, a TGBA with s states and n acceptance conditions will be degeneralized into a BA with $s \times (n+1)$ states (and one acceptance condition). This is what we do in our experiments. Alternatives include the translation of the property into a *state-based* generalized automaton which can then also be degeneralized, or the translation of the property into an alternating Büchi automaton that is then converted into a BA using the Miyano-Hayashi construction [15].

The emptiness check algorithms that can deal with TGBA will also work on BA (a BA can be seen as a TGBA by pushing the acceptance conditions on the transition leaving acceptance states). But it can also be done using two nested depth-first searches. The comparison of these different emptiness checks has raised many studies [9, 20, 5].

Fig. 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 recognize the property $\bigwedge_{i=1}^{n} \mathsf{G}\,\mathsf{F}\,p_i$ so this Büchi automaton is optimal [2]. The property $a\,\mathsf{U}\,\mathsf{G}\,b$, on the other hand, is easier to express: the BA has the same size as the TGBA.

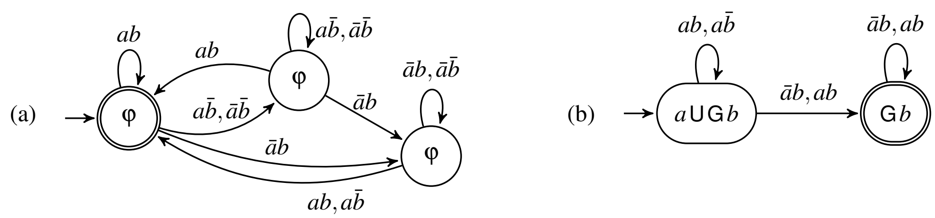


Fig. 2: Two example BA, with acceptance states shown as double circles. (a) A BA for the LTL property $\varphi = \mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$ obtained by degeneralizing the TGBA for Fig. 1(a). (b) A BA for the LTL property $a\,\mathsf{U}\,\mathsf{G}\,b$.

2.3 Testing Automata

A property, i.e., a set of infinite sequences $\mathcal{P} \subseteq K^\omega$, is *stuttering-insensitive* iff any sequence $k_0k_1k_2\ldots \in \mathcal{P}$ remains in $\mathcal{P}$ after repeating any valuation k_i. In other words, $\mathcal{P}$ is stuttering-insensitive iff

$$k_0k_1k_2\ldots \in \mathcal{P} \iff k_0^{i_0}k_1^{i_1}k_2^{i_2}\ldots \in \mathcal{P} \text{ for any } i_0 > 0, i_1 > 0\ldots$$

It is well known that any LTL$\setminus$X formula (i.e. an LTL formula that does not use the X operator) describes a stuttering-insensitive property. (It is possible to build some stuttering-insensitive LTL formulæ using the X operator [6].)

Testing Automata (TA) were introduced by Hansen et al. [11] 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 execution that ends by stuttering in the same TA state, a new kind of acceptance states is introduced: "livelock acceptance states".

If A and B are two valuations, let us note $A \oplus B$ the symmetric set difference, i.e. the set of atomic propositions that changed. E.g. $a\bar{b} \oplus ab = \{b\}$.

Definition 3 *A TA over the alphabet $K = 2^{AP}$ is a tuple $T = \langle S, I, U, R, F, G \rangle$. where:*
- *S is a finite set of states,*
- *$I \subseteq S$ is the set of initial states,*
- *$U : I \to K$ is a function mapping each initial state to a symbol of K interpreted as a valuation (the initial configuration),*
- *$R \subseteq S \times K \times S$ is the transition relation where each transition (s,k,d) is labeled by a* changeset*: $k \in K = 2^{AP}$ is interpreted as a set of atomic propositions that should 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.*

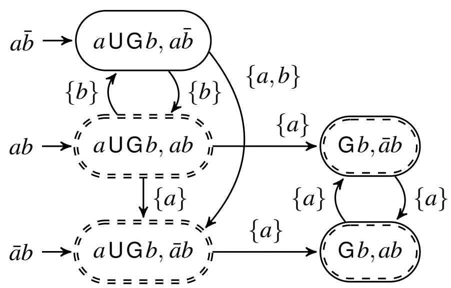
An execution $w = k_0 k_1 k_2 \ldots \in K^\omega$ is accepted by T if there exists an infinite sequence $(s_0, k_0 \oplus k_1, s_1)(s_1, k_1 \oplus k_2, s_2) \ldots (s_i, k_i \oplus k_{i+1}, s_{i+1}) \ldots \in (S \times K \times S)^\omega$ such that:
- *$s_0 \in I$ with $U(s_0) = k_0$,*
- *$\forall i \in \mathbb{N}$, either $(s_i, k_i \oplus k_{i+1}, s_{i+1}) \in R$ (we are progressing in the testing automaton), 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 automaton 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 stay on that state because the execution is stuttering).*

Construction of a Testing Automaton from a Büchi Automaton From a BA $B = (S_B, I_B, R_B, F_B)$ over the alphabet $K = 2^{AP}$, we obtain a TA $T = (S_T, I_T, U_T, R_T, F_T, G_T)$ representing the same property in two steps [8]:

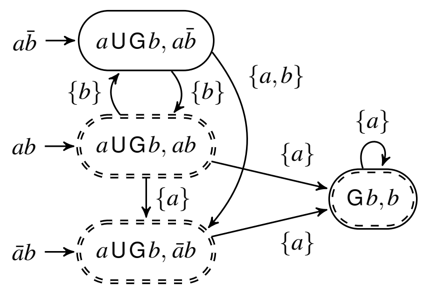
1. Converting B into an intermediate form of T with $G_T = \emptyset$:
   - $S_T = S_B \times K$, $I_T = I_B \times K$, $F_T = F_B \times K$, and $G_T = \emptyset$
   - $\forall (s,k) \in I_T$, $U_T((s,k)) = k$
   - $\forall (s_1,k_1) \in S_T, \forall (s_2,k_2) \in S_T$,
     $((s_1,k_1), k_1 \oplus k_2, (s_2,k_2)) \in R_T \iff \exists k \in 2^K, ((s_1,k,s_2) \in R_B) \wedge (k_1 \in k)$
2. Filling G_T to simplify T. For that, compute all strongly connected components using only stuttering transitions (i.e., transitions labeled by $\emptyset$). If such a SCC is not trivial (i.e., it contains a cycle) and contains a Büchi acceptance state, then add all its states to G_T. Add to I_T or G_T any state that can respectively reach I_T or G_T using only stuttering transitions. Finally remove all stuttering transitions from R_T.

Additionally, the TA can be minimized by merging bisimilar states.

Fig. 3 shows the automaton constructed for $a \,\mathsf{U}\, \mathsf{G}\, b$ by applying the above construction on the automaton from Fig. 2(b). The TA for $\mathsf{G}\,\mathsf{F}\, a \wedge \mathsf{G}\,\mathsf{F}\, b$ is too big to be shown: it has 11 states and 64 transitions.



(a) Before reduction.



(b) After reduction by bisimulation.

Fig. 3: Two TA for the LTL formula $a\,\mathsf{U}\,\mathsf{G}\,b$. States with a double enclosure belong to either F or G: states in $F \setminus G$ (none here) have a double plain line, states in $G \setminus F$ have a double dashed line, and state in $F \cap G$ use a mixed dashed/plain style.

Emptiness check using TA A first difference between the BA and TA approaches appears in the product computation. Indeed, a testing automaton remains in the same state when the Kripke structure executes a stuttering step.

The emptiness check also requires a dedicated algorithm because there are two ways to accept an execution: Büchi acceptance or livelock acceptance. In the algorithm sketched by Geldenhuys and Hansen [8], a first pass is used with an heuristic to detect both Büchi and livelock acceptance cycles. Unfortunately, in certain cases this first pass fails to report existent livelock acceptance cycles. This implies that when no counterexample is found by the first pass, a second one is required to double-check for possible livelock acceptance cycles. These two passes are annoying when the property is satisfied (no counterexample) since the entire state-space has to be explored twice.

Optimizations Looking at Fig. 3 inspires two optimizations. The first one is based on the fact that the construction of testing automata described in previous section will generate a lot of bisimilar states such as $(\mathsf{G}\,b,\bar{a}b)$ and $(\mathsf{G}\,b,ab)$. This is because the construction considers all the elements of K that are compatible with $\mathsf{G}\,b$. Had the LTL formula been over $AP = \{a,b,c\}$, e.g., $(a \vee c)\,\mathsf{U}\,\mathsf{G}\,b$, then we would have had four bisimilar states: $(\mathsf{G}\,b,\bar{a}b\bar{c})$, $(\mathsf{G}\,b,\bar{a}bc)$, $(\mathsf{G}\,b,ab\bar{c})$, and $(\mathsf{G}\,b,abc)$. These state are *necessarily* isomorphic, because they only differ in a and c, some propositions that the formula $\mathsf{G}\,b$ does not *observe*.

A more efficient way to construct the testing automaton (and to construct the automaton from Fig. 3b directly) would be to consider only the subset of atomic propositions that are observed by the corresponding state of the Büchi automaton or its descendants (if the state is labeled by an LTL formula, the atomic propositions occurring in this formula give an over-approximation of that set).

A second optimization relies on the fact any state that no part of a SCC (also called *trivial* SCC) can be added to F without changing the language of the automaton. This is true for the three kinds of automata. For instance on Fig. 3 the state $(a\,\mathsf{U}\,\mathsf{G}\,b,\bar{a}b)$ can be added to F. Since this state is not part of any cycle, it cannot occur infinitely often and therefore cannot change the accepted language of the automaton.

This change allows further simplifications by bisimulation: the state $(a\,\mathsf{U}\,\mathsf{G}\,b, a\bar{b})$ is now obviously equivalent to the $(\mathsf{G}\,b, b)$ state. Fig. 4 shows the resulting automaton. Note that putting any trivial SCC x in F before preforming bisimulation could hinder the reduction if x was isomorphic to some state not in F. However if x has only successors in F, as in our example, then it can be put safely in F: indeed, it can only be isomorphic to an F-state, or to another trivial SCC that will be added to F. This condition is similar to the one used by Löding before minimizing deterministic weak ω-automata [14].

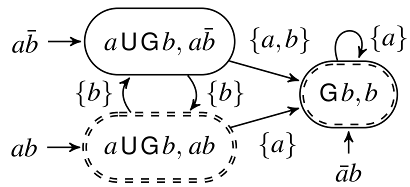


Fig. 4: Reduced TA for $a\,\mathsf{U}\,\mathsf{G}\,b$.

3 Experimentation

This section presents our experimentation of the various types of automata within our tool Spot [16]. 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.

3.1 Implementation on top of Spot

Spot is a model-checking library offering several algorithms that can be combined to build a model checker [7]. Fig. 5 shows the building blocks we used to implement the three approaches. The TGBA and BA approaches share the same synchronized product and emptiness check, while a dedicated algorithms is required by the TA approach.

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, or TA) 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

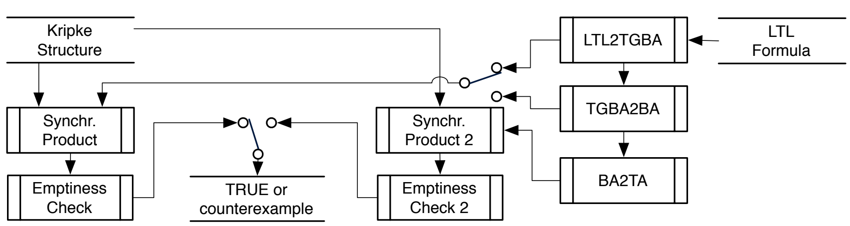


Fig. 5: The experiment's architecture. Two command-line switches controls which one of the three approaches is used to verify an LTL formula on a Kripke structure.

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 [11] 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.

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.

3.2 Benchmark Inputs

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

Toy Examples A first class of four models were selected from the Petri net literature [1]: the flexible manufacturing system (FMS), the Kanban system, the dining philosophers, and the slotted-ring system. All these models have a parameter n. For the dining philosophers, and the slotted-ring, the model are composed of n identical 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. The objective is to have comparable state spaces to be synchronized.

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

MAPK models a biochemical reaction: Mitogen-activated protein kinase cascade [12]. 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:

$$\Phi_1 = \neg((\neg \mathit{RafP}) \mathsf{U} \mathit{MEKP}) \wedge \neg((\neg \mathit{MEKP}) \mathsf{U} \mathit{MEKPP}) \wedge \\ \neg((\neg \mathit{MEKPP}) \mathsf{U} \mathit{ERKP}) \wedge \neg((\neg \mathit{ERKP}) \mathsf{U} \mathit{ERKPP})$$

PolyORB models the core of the μbroker component of a middleware [13] in an implementation using a Leader/Followers policy [18]. 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:

$$\Phi_2 = \mathsf{G}(\mathit{MSrc}_1 \rightarrow \mathsf{F}(\mathit{DOSrc}_1)) \wedge \mathsf{G}(\mathit{MSrc}_2 \rightarrow \mathsf{F}(\mathit{DOSrc}_2)) \wedge \mathsf{G}(\mathit{MSrc}_3 \rightarrow \mathsf{F}(\mathit{DOSrc}_3))$$

Types of Formulæ As suggested by Geldenhuys and Hansen [8], 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 TGBA approach.
- *WFair*: properties of the form $(\bigwedge_{i=1}^{n} \mathsf{G}\,\mathsf{F}\,p_i) \rightarrow \varphi$, where φ is a randomly generated LTL formula. This represents the verification of φ under the weak-fairness hypothesis $\bigwedge_{i=1}^{n} \mathsf{G}\,\mathsf{F}\,p_i$. 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 $n \approx 3.19$ on the average.

All formulæ were translated into automata using Spot, which was shown experimentally to be very good at this job [19].

3.3 Results

Table 1 and 2 show how the three approaches deal with toy models and random formulæ (Table 1) and with toy models against WFair formulæ (Table 2). Table 3 shows the results of the two cases studies against random, weak-fairness, and dedicated formulæ.

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.

The numbers displayed in parentheses on both sides of the tables are the number of formulæ involved in the experiment. For instance (reading Table 2) we checked Kanban5 against 98 weak-fairness formulæ that had no counterexample, and against 102 weak-fairness 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, including 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). These tests were performed on a 64bit Linux system running on an Intel Core i7 CPU 960 at 3.20GHz, with 24GB of RAM. Running this entire benchmark with four tasks in parallel took us two days.

			Property verified (no counterexample)							Property violated (a counterexample exists)							
			Automaton		Full product		Emptiness check			Automaton		Full product		Emptiness check			
			st.	tr.	st.	tr.	st.	tr.	T	st.	tr.	st.	tr.	st.	tr.	T	
FMS5 (70)	TGBA	Avg	5.9	67.1	698 449	4 750 201	**698 449**	**4 750 201**	**740**	6.3	75.8	8 190 410	73 457 965	118 742	681 874	109	**FMS5** (230)
		Max	24	310	5 961 942	54 621 333	5 961 942	54 621 333	7 685	30	493	35 692 168	462 702 111	4 554 970	28 262 831	4 127	
	BA	Avg	7.3	79.6	790 859	5 389 591	790 859	5 389 591	830	7.6	89.9	8 848 201	79 645 055	89 948	451 848	77	
		Max	28	338	8 310 792	72 673 494	8 310 792	72 673 494	9 582	63	1 037	37 211 496	473 322 666	3 085 939	23 927 298	3 565	
	TA	Avg	27.1	365.5	**521 260**	**4 023 469**	1 042 519	8 046 939	1 865	26.8	389.6	8 235 551	67 897 061	**61 095**	**338 607**	**91**	
		Max	82	2 256	4 078 242	32 815 605	8 156 484	65 631 210	14 490	123	3 255	34 897 110	295 594 539	1 860 929	14 720 770	3 819	
Kanban5 (100)	TGBA	Avg	5.2	48.5	852 364	7 279 249	**852 364**	**7 279 249**	**909**	7.0	71.6	7 126 650	77 809 374	47 984	237 295	33	**Kanban5** (100)
		Max	27	264	6 694 184	70 465 136	6 694 184	70 465 136	8 373	22	292	21 715 730	241 387 835	1 604 560	11 177 672	1 510	
	BA	Avg	6.1	56.3	852 493	7 279 889	**852 493**	**7 279 889**	**910**	8.6	87.6	8 041 841	87 518 994	36 085	194 392	25	
		Max	29	296	6 694 184	70 465 136	6 694 184	70 465 136	8 335	38	472	23 997 065	270 130 066	1 628 283	11 232 778	1 513	
	TA	Avg	20.2	227.2	**651 299**	**6 074 858**	1 302 598	12 149 717	2 451	29.7	368.3	7 162 575	70 438 470	**17 766**	**141 630**	**29**	
		Max	114	1 858	6 409 984	62 033 608	12 819 968	124 067 216	25 344	134	2 221	17 551 016	175 769 251	1 163 547	10 736 232	2 217	
Philo8 (100)	TGBA	Avg	6.1	87.0	219 303	1 232 080	**219 303**	**1 232 080**	**257**	7.3	99.2	637 670	4 950 129	36 161	168 189	37	**Philo8** (100)
		Max	20	338	830 533	6 366 282	830 533	6 366 282	1 172	27	360	1 489 852	16 311 100	634 183	5 245 872	963	
	BA	Avg	7.1	98.5	220 049	1 234 944	**220 049**	**1 234 944**	**258**	9.1	122.0	737 638	5 767 111	29 216	105 082	25	
		Max	21	367	830 533	6 366 282	830 533	6 366 282	1 174	38	604	3 005 819	32 843 222	344 134	1 308 577	317	
	TA	Avg	30.9	541.9	**148 562**	**1 029 393**	297 124	2 058 786	662	36.6	619.0	636 866	4 677 877	**18 925**	**89 670**	**33**	
		Max	110	3 123	554 335	3 980 981	1 108 670	7 961 962	2 472	160	3 225	2 491 222	20 365 681	217 114	1 549 281	497	
Ring6 (100)	TGBA	Avg	5.4	58	476 612	2 940 953	**476 612**	**2 940 953**	**564**	7.0	90.4	1 702 969	11 452 375	144 848	694 019	136	**Ring6** (100)
		Max	18	236	4 162 012	45 176 784	4 162 012	45 176 784	7 181	20	385	5 172 800	35 474 194	1 172 951	7 407 167	1 401	
	BA	Avg	6.3	65.7	494 077	3 012 946	494 077	3 012 946	582	8.5	109.2	1 865 260	12 543 141	117 181	576 625	110	
		Max	22	326	4 378 216	46 903 064	4 378 216	46 903 064	7 683	25	401	5 211 769	43 250 640	1 323 327	8 460 521	1 584	
	TA	Avg	22.6	310.0	**379 088**	**2 163 360**	758 175	4 326 721	1 329	33.8	540.6	1 697 686	10 029 775	**68 807**	**366 600**	**113**	
		Max	122	2 382	2 232 820	14 106 432	4 465 640	28 212 864	8 130	141	3 531	4 891 128	28 812 656	946 951	5 415 785	1 726	

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

			Property verified (no counterexample)							Property violated (a counterexample exists)							
			Automaton		Full product		Emptiness check			Automaton		Full product		Emptiness check			
			st.	tr.	st.	tr.	st.	tr.	T	st.	tr.	st.	tr.	st.	tr.	T	
FMS5 (37)	TGBA	Avg	3.1	26	5 197 375	43 078 717	**5 197 375**	**43 078 717**	**6 191**	5.4	49.5	**9 935 828**	**89 550 059**	627 618	3 517 626	559	**FMS5** (163)
		Max	7	104	9 866 094	91 499 667	9 866 094	91 499 667	13 282	12	212	21 413 973	319 212 813	5 865 891	51 379 790	7 313	
	BA	Avg	7.3	58.4	7 325 010	53 471 546	7 325 010	53 471 546	7 708	11.8	112.6	17 297 219	154 876 145	651 799	3 894 388	593	
		Max	35	526	11 338 161	103 816 053	11 338 161	103 816 053	13 394	49	578	64 477 308	784 721 607	17 345 804	148 875 504	21 435	
	TA	Avg	36.0	361.1	**3 967 433**	**31 419 765**	7 934 866	62 839 531	14 231	60.6	656.7	15 339 186	126 259 786	**216 321**	**1 526 860**	**364**	
		Max	215	3 460	9 002 196	70 152 851	18 004 392	140 305 702	31 515	205	2 985	47 074 692	415 672 995	3 732 706	30 145 223	7 165	
Kanban5 (98)	TGBA	Avg	2.7	14.9	2 730 709	23 071 387	**2 730 709**	**23 071 387**	**2 788**	3.5	25	5 484 209	55 893 401	526 015	3 049 738	410	**Kanban5** (102)
		Max	7	56	8 092 182	78 624 126	8 092 182	78 624 126	10 214	10	140	13 900 320	166 038 726	2 895 449	24 460 029	3 005	
	BA	Avg	5.9	31	3 382 871	26 705 745	3 382 871	26 705 745	3 183	7.1	53.2	8 408 110	82 426 568	531 367	3 035 376	415	
		Max	20	150	12 307 085	113 079 575	12 307 085	113 079 575	11 962	30	354	23 144 848	300 434 051	6 104 368	43 693 336	6 384	
	TA	Avg	21.0	123.6	**1 923 597**	**17 403 907**	3 847 194	34 807 815	6 891	32.8	281.9	6 365 280	61 028 298	**146 619**	**1 200 463**	**240**	
		Max	108	1 364	6 677 524	63 784 672	13 355 048	127 569 344	26 651	187	2 554	18 114 712	190 516 984	1 163 652	10 736 394	2 146	
Philo8 (100)	TGBA	Avg	3.0	19.1	191 233	1 072 039	**191 233**	**1 072 039**	**225**	4.1	40.9	**388 356**	**2 836 796**	**11 526**	**22 540**	**8**	**Philo8** (100)
		Max	10	72	961 946	8 584 333	961 946	8 584 333	1 581	11	110	1 106 279	10 139 160	148 028	667 632	153	
	BA	Avg	7.2	47.7	226 231	1 219 657	226 231	1 219 657	254	9.5	107.3	925 540	6 664 879	13 374	32 724	10	
		Max	24	213	961 946	8 584 333	961 946	8 584 333	1 577	29	459	3 369 900	24 286 322	290 681	1 107 465	265	
	TA	Avg	32.3	245.9	**141 303**	**969 063**	282 607	1 938 127	615	68.0	839.7	898 752	6 458 513	**11 212**	**24 675**	**13**	
		Max	128	1 746	665 509	5 048 600	1 331 018	10 097 200	3 026	205	3 027	2 280 459	16 828 197	99 824	619 861	200	
Ring6 (100)	TGBA	Avg	3.5	21.3	362 296	2 072 837	**362 296**	**2 072 837**	**413**	3.7	37.3	**903 909**	**5 518 052**	**27 114**	**105 130**	**23**	**Ring6** (100)
		Max	10	98	2 116 458	13 877 156	2 116 458	13 877 156	2 531	12	109	2 573 186	16 268 868	831 566	4 479 900	929	
	BA	Avg	7.2	44.9	436 729	2 370 915	436 729	2 370 915	476	8.6	92.8	2 112 826	12 623 603	39 004	168 105	35	
		Max	22	240	2 868 218	17 192 038	2 868 218	17 192 038	3 168	37	528	6 641 645	42 624 886	1 123 128	5 300 114	1 145	
	TA	Avg	30.3	220.1	**329 599**	**1 831 831**	659 198	3 663 661	1 121	61.6	732.3	2 166 241	12 573 562	**27 645**	**141 549**	**44**	
		Max	154	2 020	1 658 112	9 402 736	3 316 224	18 805 472	5 629	237	3 456	5 113 422	30 167 566	793 363	4 498 438	1 408	

Table 2: Comparison of the three approaches on toy examples with weak-fairness formulæ, when counterexamples do not exist (left) or when they do (right).

				Automaton (left)		Full product (left)		Emptiness check (left)			Automaton (right)		Full product (right)		Emptiness check (right)		
				st.	tr.	st.	tr.	st.	tr.	T	st.	tr.	st.	tr.	st.	tr.	T
MAPK 8	RND (100)	TGBA	Avg	4.4	40.7	539 552	6 674 103	**539 552**	**6 674 103**	**887**	5.2	52.5	13 077 352	168 677 550	41 840	182 026	33
			Max	14	191	15 567 779	261 545 658	15 567 779	261 545 658	31 855	16	256	28 096 430	392 571 703	2 245 468	13 298 596	2 249
		BA	Avg	5.2	47.4	539 557	6 674 123	**539 557**	**6 674 123**	**885**	6.0	59.2	14 328 287	186 237 286	49 451	258 493	43
			Max	19	227	15 567 780	261 545 660	15 567 780	261 545 660	31 558	19	304	50 416 848	723 868 664	1 880 880	13 168 794	2 059
		TA	Avg	16.8	192.1	**471 923**	**5 943 950**	943 846	11 887 899	2 623	19.4	263.6	13 631 076	176 047 549	**26 698**	**158 686**	**41**
			Max	90	2 148	12 969 362	172 035 602	25 938 724	344 071 204	73 136	61	1 606	37 259 478	491 488 765	1 126 525	12 146 182	2 855
	WFair (100)	TGBA	Avg	2.7	20.3	1 536 626	16 368 553	**1 536 626**	**16 368 553**	**2 330**	3.6	29.4	**8 187 969**	**105 698 151**	79 943	663 821	96
			Max	9	116	11 888 331	160 777 864	11 888 331	160 777 864	21 133	10	102	29 833 996	474 629 285	5 845 125	56 616 219	7 964
		BA	Avg	6.8	55.8	1 948 686	18 950 258	1 948 686	18 950 258	2 731	7.8	67.7	20 075 109	258 176 409	20 787	92 900	15
			Max	29	234	18 595 927	201 692 352	18 595 927	201 692 352	29 129	29	379	63 544 808	777 667 365	435 162	2 407 814	391
		TA	Avg	37.9	360.8	**1 193 177**	**14 474 879**	2 386 354	28 949 758	6 473	47.5	462.0	18 994 457	243 894 317	**12 906**	**83 099**	**19**
			Max	151	2 068	10 842 174	134 517 672	21 684 348	269 035 344	60 556	205	2 765	49 024 627	623 237 293	607 938	6 462 486	1 350
	Φ_1	TGBA	–	6	165	46 494	302 350	**46 494**	**302 350**	**40**							
		BA	–	6	165	46 494	302 350	**46 494**	**302 350**	**37**							
		TA	–	38	1 245	**33 376**	**289 235**	66 752	578 470	121							
PolyORB 3/3/2	RND (100)	TGBA	Avg	7.0	98.1	63 442	163 279	**63 442**	**163 279**	**303**	6.2	97.2	95 338	234 746	**49 026**	**113 889**	**232**
			Max	22	378	185 103	528 174	185 103	528 174	888	36	832	400 890	1 518 043	114 641	339 747	540
		BA	Avg	8.4	114.6	64 662	165 861	**64 662**	**165 861**	**309**	7.6	120.3	101 418	251 469	**49 650**	**115 621**	**235**
			Max	38	550	218 541	608 274	218 541	608 274	1 045	54	1 240	558 927	1 950 778	114 641	339 810	540
		TA	Avg	28.7	492	**59 497**	**127 607**	118 994	255 214	598	33.6	646.2	105 717	228 225	52 554	116 077	264
			Max	71	2 264	184 974	396 105	369 948	792 210	1 863	164	3 600	319 171	725 255	193 931	436 562	990
	WFair (100)	TGBA	Avg	4.1	40.6	**58 539**	**132 985**	**58 539**	**132 985**	**278**	3.4	34.4	**67 654**	**145 394**	**55 160**	**120 796**	**262**
			Max	9	128	122 817	373 584	122 817	373 584	582	8	136	152 049	351 735	116 340	274 221	552
		BA	Avg	9.6	103.9	88 845	197 798	88 845	197 798	420	7.9	84.3	146 686	316 641	100 382	221 237	474
			Max	38	612	243 637	522 549	243 637	522 549	1 145	29	379	283 274	615 449	191 284	443 762	909
		TA	Avg	65.1	771.1	92 749	198 283	185 498	396 567	933	57.9	681	168 325	361 321	105 868	232 165	533
			Max	244	4 132	288 852	618 696	577 704	1 237 392	2 927	205	2 765	348 705	752 969	199 777	442 369	1 017
	Φ_2	TGBA	–	7	576	**345 241**	**760 491**	**345 241**	**760 491**	**1 642**							
		BA	–	7	576	**345 241**	**760 491**	**345 241**	**760 491**	**1 646**							
		TA	–	79	14 526	**342 613**	**742 815**	685 226	1 485 630	3 532							

Table 3: Comparison of the three approaches for the case studies when counterexamples do not exist (left) or when they do (right).

4 Discussion

Although the state space of cases studies can be very different from random state spaces [17], a first look at our results confirms two facts already observed by Geldenhuys and Hansen using random state spaces [8]: (1) although the TA constructed from properties are usually a lot larger than BA, 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 the BA.

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

On verified properties, the results are very straightforward to interpret: the BA are slightly worse than the TGBA because they have to be degeneralized. In fact, the average number of acceptance conditions needed in random formulæ (Table 1 and 3) is so close to 1 that the degeneralization barely changes the sizes of the automata. With weak-fairness formulæ (Table 2 and 3), the number of acceptance conditions is greater, so TGBA are favored over BA. Surprisingly, both TGBA and BA, although they are not tailored to *stuttering-insensitive* properties like TA, appear more effective to prove that a *stuttering-insensitive* property is verified. In the three tables, although the full product of the TA approach is smaller than the other approaches, it has to be explored twice (as explained in section 2.3): the emptiness-check consequently explores more states and transitions. This double exploration is not enough to explain the big runtime differences. Two other subtler implementation details contribute to the time difference:

– To synchronize a transition of a Kripke structure with a transition (or a state in case of stuttering) of a TA, we must compute the symmetric difference $l(s) \oplus l(d)$ between the labels of the source and destination states. The same synchronization in the TGBA and BA approaches requires to know only the source label.
  Computing these labels is a costly operation in CheckPN because Petri net marking are compressed in memory to save space. Although we implemented some (limited) caching to alleviate the number of such label computation, profiling measures revealed the TA approach was 3 times slower than the TGBA and BA approaches, but that labels where computed 9 times more.
– A second implementation difference, this time in favor of the TA approach, is that transitions of testing automata are labeled by elements of K, while transitions of TGBA and BA are labeled by elements of 2^K. That means that once $l(s) \oplus l(d) \in K$ has been computed, we can use a hash table to immediately find matching transitions of the testing automaton. In the TGBA and BA implementations, we linearly scan the list of transitions of the property automaton until we find one compatible with $l(s)$. The BA and TGBA approaches could be improved by replacing each transition labeled by an element of 2^K by many transitions labeled by an elements of K, and then using a hash table, but we have not implemented it yet.

In an implementation where computing labels is cheap, the run time should be proportional to the number of transitions explored by the emptiness check, so it is important not to consider only the run time provided by our experiments.

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.

This ordering luckiness explains why the BA approach sometimes outperforms the TGBA approach: one very bad case is enough to bias the average case. For instance this occurred on the Philo8 model with random formulæ: the worst TGBA case explored 4 times more transitions than the BA case, although the full product was twice smaller.

We believe that the TA, since they are more deterministic, are less sensible to this ordering. They also explore a smaller state space on the average. This smaller exploration is not always tied a good runtime because of the extra computation of labels discussed previously. Again, looking at the average number of transition explored by the emptiness check indicates that the TA approach would outperform the others if the computation of labels was cheap.

Finally 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 debugging (less than 0.005% of the cases in their experiments [8]).

5 Conclusion

Geldenhuys and Hansen have evaluated the performance of the BA and TA approaches with small random Kripke structures checked against LTL formulæ taken from the literature [8]. In this work, we have completed their experiments by using actual models and different kinds of formulæ (random formulæ not trivially verifiable, random formulæ expressing weak-fairness formulæ, and a couple of real formulæ), by evaluating the TGBA approach, and by distinguishing violated formulæ and verified formulæ in the benchmark.

For verified formulæ, we found that the state space reduction achieved by the TA approach was not enough to compensate for the two-pass emptiness check this approach requires. It is therefore better to use the TGBA approach to prove that a *stuttering-insensitive* formula is verified and TA approach in an earlier "debugging phase".

When the formulæ are violated, the TA approach usually processes less transitions than the BA approach and TGBA to find a counterexample. This approach should therefore be a valuable help to debug models (i.e. when counterexamples are *expected*). This is especially true on random formulæ. With weak-fairness formulæ, generalized automata are advantaged and are able to beat the TA on the average in 3 of our 6 examples (Philo8, Ring6, PolyORB 3/2/2).

Future work We plan to combine the ideas of TA and TGBA approaches. We believe it would be interesting to have testing automata with transition-based generalized acceptance conditions. We think the LTL translation algorithm we use to produce TGBAs could be adjusted to product such automata directly.

References

1. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. of ICATPN'00*, vol. 1825 of *LNCS*, pp. 103–122. Springer.
2. J. Cichoń, A. Czubak, and A. Jasiński. Minimal Büchi automata for certain classes of LTL formulas. In *Proc. of DEPCOS'09*, pp. 17–24. IEEE Computer Society.
3. J.-M. Couvreur. On-the-fly verification of temporal logic. In *Proc. of FM'99*, vol. 1708 of *LNCS*, pp. 253–271. Springer.
4. J.-M. Couvreur. Un point de vue symbolique sur la logique temporelle linéaire. In *Actes du Colloque LaCIM 2000*, vol. 27 of *Publications du LaCIM*, pp. 131–140. Université du Québec à Montréal, Aug. 2000.
5. J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. of SPIN'05*, vol. 3639 of *LNCS*, pp. 143–158. Springer.
6. J. Dallien and W. MacCaull. Automated recognition of stutter-invariant LTL formulas. *Atlantic Electronic Journal of Mathematics*, (1):56–74, 2006.
7. A. Duret-Lutz and D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In *Proc. of MASCOTS'04*, pp. 76–83. IEEE Computer Society Press.
8. J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *Proc. of SPIN'06*, vol. 3925 of *LNCS*, pp. 53–70. Springer.
9. J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proc. of TACAS'04*, vol. 2988 of *LNCS*, pp. 205–219. Springer.
10. D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In *Proc. of FORTE'02*, vol. 2529 of *LNCS*, pp. 308–326.
11. H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In *Proc. of FMICS'02*, vol. 66(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier.
12. M. Heiner, D. Gilbert, and R. Donaldson. Petri nets for systems and synthetic biology. In *Proc. of SFM'08*, vol. 5016 of *LNCS*, pp. 215–264. Springer.
13. J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Barrir, and T. Vergnaud. On the formal verification of middleware behavioral properties. In *Proc. of FMICS'04*, vol. 133 of *Electronic Notes in Theoretical Computer Science*, pp. 139–157. Elsevier.
14. C. Löding. Efficient minimization of deterministic weak ω-automata. *Information Processing Letters*, 79(3):105–109, 2001.
15. S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.
16. MoVe/LRDE. The Spot home page: `http://spot.lip6.fr`, 2011.
17. R. Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(5):443–454, 2008.
18. I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt. Evaluating and optimizing thread pool strategies for RT-CORBA. In *Proc. of LCTES'00*, pp. 214–222. ACM.
19. K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Proc. of SPIN'07*, vol. 4595 of *LNCS*, pp. 149–167. Springer.
20. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *Proc. of TACAS'05*, vol. 3440 of *LNCS*. Springer.
21. R. Sebastiani and S. Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *Proc. of CHARME'03*, vol. 2860 of *LNCS*, pp. 126–140. Springer.
22. H. Tauriainen. *Automata and Linear Temporal Logic: Translation with Transition-based Acceptance*. PhD thesis, Helsinki University of Technology, Espoo, Finland, Sept. 2006.
23. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of Banff'94*, vol. 1043 of *LNCS*, pp. 238–266. Springer.