Is There a Best Büchi Automaton for Explicit Model Checking?

František Blahoudek
Masaryk University
Brno, Czech Republic
xblahoud@fi.muni.cz

Alexandre Duret-Lutz
LRDE, EPITA
Le Kremlin-Bicêtre, France
adl@lrde.epita.fr

Mojmír Křetínský
Masaryk University
Brno, Czech Republic
kretinsky@fi.muni.cz

Jan Strejček
Masaryk University
Brno, Czech Republic
strejcek@fi.muni.cz

ABSTRACT

LTL to Büchi automata (BA) translators are traditionally optimized to produce automata with a small number of states or a small number of non-deterministic states. In this paper, we search for properties of Büchi automata that really influence the performance of explicit model checkers. We do that by manual analysis of several automata and by experiments with common LTL-to-BA translators and realistic verification tasks. As a result of these experiences, we gain a better insight into the characteristics of automata that work well with Spin.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*temporal logic*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

General Terms

Theory, Algorithms, Verification

Keywords

Linear temporal logic, Büchi automata, explicit model checking

1. INTRODUCTION

The automata-theoretic approach to explicit model checking of *Linear-time Temporal Logic* (LTL) [25] can be broken down into four steps: (1) build the state space, i.e., an automaton $\mathcal{S}$ representing all the possible executions of the system to be verified, (2) translate an LTL formula φ representing a desired property of the system into a *Büchi Automaton* (BA) $\mathcal{A}_{\neg\varphi}$ that accepts all words violating φ, (3) build the synchronous product $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ of these two systems, and finally (4) check this product for emptiness. If $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ accepts a word, it is an execution of $\mathcal{S}$ that invalidates φ, i.e., a counterexample.

In a typical explicit model checker, the construction of the state space S and its synchronous product with $\mathcal{A}_{\neg\varphi}$ are done *one-the-fly*, driven by needs of an emptiness check procedure. This ensures that only the part of the state space that is compatible with $\mathcal{A}_{\neg\varphi}$ will be constructed. Further, the whole construction can be stopped as soon as the emptiness check finds a counterexample, i.e., a reachable cycle containing an accepting state.

Here we focus on the influence of a property automaton $\mathcal{A}_{\neg\varphi}$ on the steps (3) and (4) of a model checking procedure. There are many algorithms and tools for translating an LTL formula into a Büchi automaton, yet they produce various language equivalent automata. For instance, Figure 4 shows several Büchi automata for the LTL formula $\mathsf{GF}a \wedge \mathsf{GF}b$. Should one be preferred over the others?

The intuition that a smaller $\mathcal{A}_{\neg\varphi}$ produces a smaller synchronous product $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ is not always correct. More importantly, it is not quite relevant: ultimately, only the part of the product that is explored by the emptiness check does matter. Some authors of automata optimizations or LTL-to-BA translation improvements (e.g., Etessami and Holzmann [10] and Dax et al. [5]) provide also running times of a selected emptiness check executed on the product of obtained automata and either random state spaces or few realistic systems. Etessami and Holzmann [10] even complained that the relation between the size of $\mathcal{A}_{\neg\varphi}$ and the running time of the model checking procedure was difficult to predict, especially in the presence of a counterexample.

In order to select an ideal automaton for expressing a formula, one should be aware of the inner workings of the emptiness check procedure that will be used. Among the various existing emptiness checks, we have decided to focus on the standard emptiness check of Spin, which is a sequential algorithm based on a *Nested Depth-First Search* (NDFS) [17].

We look at concrete examples of how formulae are translated differently by existing tools to gain a better insight into the characteristics of automata that work well with Spin. Our results should stimulate LTL-to-BA translation

SPIN '14, July 21–23, 2014, San Jose, CA, USA

researchers to focus on another aspects of produced automata: not only their size and determinism.

The paper is organized as follows. The next section motivates our research by experimental results quantifying the influence of property automata on the performance of the explicit model checker Spin. Section 3 describes standard approaches to automata optimization motivated by reduction of the product size. In Section 4, we discuss how property automata can affect the performance of the NDFS-based emptiness check of Spin.

We assume familiarity with LTL and Büchi automata [3].

2. MOTIVATION BY EMPIRICAL DATA

First of all, we present experimental results showing how important the impact of Büchi automata on Spin's performance can be. We use the following benchmark, software, and hardware.

Benchmark. The considered benchmark set is based on the set of realistic model checking tasks BEEM [19]. In addition to the original 769 pairs of a model in Promela and a corresponding specification formula we added, to each model describing some mutual exclusion algorithm (altogether 23 instances of parametric models called `anderson`, `peterson`, and `bakery`), three specification formulae:

1. $\mathsf{GF}(P_0@\mathrm{CS}) \rightarrow \mathsf{GF}(P_0@\mathrm{NCS})$ meaning that if a process P_0 spends infinitely many steps in a critical section, then it also spends infinitely many steps in a non-critical section,
2. $\mathsf{GF}(P_0@\mathrm{NCS}) \rightarrow \mathsf{GF}(P_0@\mathrm{CS})$ meaning that if a process P_0 spends infinitely many steps in a non-critical section, then it also spends infinitely many steps in a critical section,
3. $\mathsf{FG}\neg\big((P_0@\mathrm{CS}\wedge P_1@\mathrm{CS})\vee(P_0@\mathrm{CS}\wedge P_2@\mathrm{CS})\vee(P_1@\mathrm{CS}\wedge P_2@\mathrm{CS})\big)$ meaning that after finitely many steps, it never happens that two of the processes P_0, P_1, and P_2 are in a critical section at the same time.

To sum up, we consider $769 + 3 \cdot 23 = 838$ verification tasks. All the benchmarks and measurements presented in this section are available at `http://fi.muni.cz/~xstrejc/publications/spin2014.tar.gz`.

Software. We use five LTL-to-BA translators presented in Table 1: Spin and LTL2BA are well established and popular translators, MoDeLLa was the first translator focusing on determinism of produced automata, and LTL3BA and Spot represent contemporary translators. The last two translators are used in several settings: the settings denoted by *LTL3BA (det)* and *Spot (det)* aim to produce more deterministic automata, while the setting called *Spot (no jump)* is explained in Section 4. The same version of Spin (with its default settings and the maximal search depth set to 100 000 000) is also used in all our experiments to perform all model checking steps except the LTL-to-BA translation. In particular, the partial-order reduction, which severely limits the exploration of the state-space, is enabled.

Hardware. All computations are performed on an HP DL980 G7 server with 8 eight-core 64-bit processors Intel Xeon X7560 2.26GHz and 448 GiB DDR3 RAM. Each execution of Spin has been restricted by 30 minutes timeout and a memory limit of 20GiB.

Table 1: Considered LTL-to-BA translators, for reference.

tool	version	command
Spin [10, 16]	6.2.5	`spin -f`
LTL2BA [12]	1.1	`ltl2ba -f`
MoDeLLa [21]	1.5.9	`mod2spin -f`
LTL3BA [1]	1.0.2	`ltl3ba -S -f`
LTL3BA (det)		`ltl3ba -S -M -f`
Spot [7]	1.2.4	`ltl2tgba -s`
Spot (det)		`ltl2tgba -s -D`
Spot (no jump)		`ltl2tgba -s -x degen-lskip=0`

Originally, we have measured the impact of Büchi automata on Spin by its running time. Unfortunately, our computation server is shared with other users and its variable workload has led to enormous dispersion of measured running times. We have observed a running time difference of over 300% on the same input. Hence, instead on running times, we focus on the count of *visited transitions*, which is a stable statistic produced directly by Spin. The number of visited transitions accumulates the numbers of product transitions explored in depth-first searches executed during a run of the NDFS algorithm (see Section 4 for a brief description of NDFS). Hence, the number of visited transitions should be proportional to the running time on a dedicated machine.

For each of the 838 considered verification tasks, we translate the negation of the formula by all the mentioned translators and we run Spin on the model with each of the obtained automata. Translation of the negated formula to an automaton is instantaneous (it takes less than 0.1s) in nearly all cases: there is only one formula for which the translator built in Spin needs a couple of seconds to finish. For 823 tasks, Spin successfully finishes the computation within the given limits for at least two automata obtained by different translation tools. For each such verification task, we find the maximal and the minimal numbers of visited transitions and we compute their ratio. Intuitively, the ratio represents how many times slower Spin can be if we choose the worst of the produced automata compared to the best of those.

Out of the 823 tasks, the ratio is exactly 1 only in 35 cases. In other words, in more than 95% of the considered verification tasks, the choice of an LTL-to-BA translator has an influence on running time of Spin.

In fact, the ratios significantly differ for verification tasks where the model satisfies a given formula and for those with a counterexample. Out of the 823 tasks, 731 tasks contain counterexamples while 92 tasks do not. The ratios for these two sets are presented by box-plots in Figure 1. One can clearly see that the selection of a Büchi automaton has a bigger impact on the verification tasks with counterexamples (median ratio is over 5.6) than on the tasks without counterexamples (median ratio is 1.4). Both sets contain extreme cases where the ratios exceed 10^6.

Spin also provides statistics for *stored states*, which is the total count of constructed and stored product states and should be proportional to the memory consumed by Spin. If we compute ratios of maximal and minimal numbers of stored states, we get the ratio 1 in 68 out of the 823 tasks.

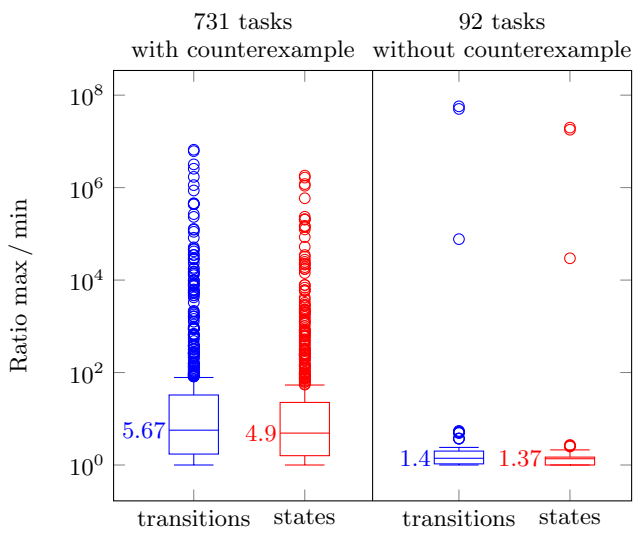

Figure 1: Impact of the Büchi automata on model checking. For each verification task, we compute ratios between the maximum and minimum number of transitions (or unique states) visited by Spin using all available Büchi automata. In each column, a box spans between the first and third quartiles, and is split by the median (whose value is given). The whiskers show the range of ratios below the first and above the third quartile that are not further away from the quartiles than 1.5 times the interquartile range. Other values are shown as outliers using circles.

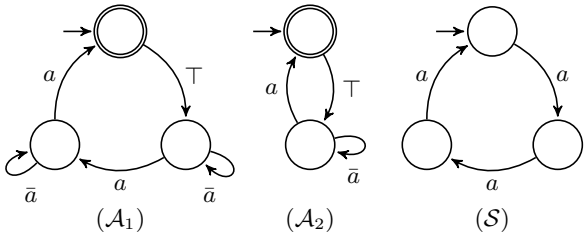

Figure 2: Two BA for $\mathtt{GF}a$ and a state space. $\mathcal{A}_1 \otimes \mathcal{S}$ has 3 states whereas $\mathcal{A}_2 \otimes \mathcal{S}$ has 6. Note that edges in the automata are labelled by Boolean formulae over atomic propositions, where $\bar{a}$ means $\neg a$, $\top$ stands for *true*, and $a\bar{b}$ used later means $a \wedge \neg b$. Formally, an edge labelled with a formula ρ represents all the transitions that are labelled with a subset M of atomic propositions such that $M \models \rho$.

On Figure 1 one can see that the situation is analogous to ratios of visited transitions, but the ratios of stored states are slightly lower.

To sum up, the choice of a Büchi automaton is an important issue substantially affecting both running time and memory needed for the explicit model checking process implemented in Spin.

3. STANDARD APPROACH TO OPTIMIZATION: HELPING THE PRODUCT

Most of the work on optimizing the translation of LTL formulae to Büchi automata has focused on building Büchi automata with the smallest possible number of states [e.g. 4, 12, 22, 15, 24]. This is motivated by the observation that the synchronous product of a Büchi automaton $\mathcal{A}$ with a state space $\mathcal{S}$ can have the same number of states as their Cartesian product in the worst case: $|\mathcal{S} \otimes \mathcal{A}| \leq |\mathcal{S}| \times |\mathcal{A}|$. Therefore, decreasing $|\mathcal{A}|$ lowers the upper bound on $|\mathcal{S} \otimes \mathcal{A}|$.

However it is possible to build contrived examples where a smaller $|\mathcal{A}|$ yield a larger product. For instance, removing one state in the automaton $\mathcal{A}_1$ of Figure 2 doubles the size of its product with the state space $\mathcal{S}$ of the same figure from 3 to 6 states. Of course, if $\mathcal{S}$ was a similar cycle of 2 states, the smaller automaton $\mathcal{A}_2$ would give a smaller product.

Hence, one cannot hope to build an optimal property automaton $\mathcal{A}$ without a priori knowledge of the system $\mathcal{S}$.

With the introduction of LBTT [23], a tool that checks the output of different LTL-to-BA translators by doing many cross-comparisons, including some products with random state spaces, tool designers started to evaluate not only the size of the produced automata, but also the size of their products with random state spaces [e.g. 21, 8]. A recent clone of LBTT called `ltlcross` [6] computes multiple products with random state spaces to lessen the *luck factor*. Sebastiani and Tonetta [21] used this "product with a random state space" measurement to benchmark their translator MoDeLLa against other available translators to support the claim that producing "more deterministic" Büchi automata might be more important than producing small Büchi automata.

Benchmarks based on the size of products may look like Table 2. The table shows that MoDeLLa generates automata that are slightly bigger than LTL2BA (its competitor in 2003) but when looking at the product, MoDeLLa causes fewer transitions to be built. If the number of transitions is proportional to the running time of a model checker and the number of states is proportional to its memory consumption, MoDeLLa has effectively traded memory for speed.

MoDeLLa's results do not appear to hold today: more recent translators such as LTL3BA or the translator of Spot can produce automata that are significantly smaller and yield smaller products with random state spaces. These translators also have options to produce more deterministic automata, but the resulting products are not always better.

The right part of Table 2 compares the translators by the sizes of products of produced automata with a fixed set of random systems. For instance, one can observe that even though Spot (6) produces the lowest accumulated number of product transitions in this benchmark, there are 30 formulae where the generated products have more transitions than those obtained by LTL3BA (det) (5). Conversely, automata from LTL3BA (det) produce products with more transitions than those of Spot for 76 formulae.

It should be noted that optimizing $\mathcal{A}$ to minimize $|\mathcal{S} \otimes \mathcal{A}|$ is not equivalent to optimizing $\mathcal{A}$ for the model checking procedure, because the product $\mathcal{S} \otimes \mathcal{A}$ is constructed on-the-fly by most emptiness check algorithms. An emptiness check may explore a part of the product, and may explore it several times. Ultimately, any change to $\mathcal{A}$ should really be measured only by its effect on the model checker used. Such an evaluation was done for instance by Dax et al. [5]: in addition to explaining how to build minimal weak deterministic Büchi automata (WDBA) for a subclass of LTL,

Table 2: Translation of 178 formulae from the literature [9, 22, 11] using different LTL-to-BA translators, with a timeout of 60 seconds. Column *n* indicates how many translations are successful within the allocated time. The *automata* columns show accumulated values of standard automata characteristics for all successful translations. Column *ndst* gives the number of non-deterministic states in the automata. All produced automata are synchronized with the same 100 random systems, and the median number of states and transitions of these products is kept. The *products* columns represent the medians accumulated over all successful translations. The right-most part of the table counts the number of formulae for which the translator on the row produces an automaton with higher median number of transitions in the products that the translator of the column.

		automata				products		cases with *product trans* bigger than...							
	n	states	ndst	edges	trans	states	trans	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
(1) Spin	161	1739	1474	9318	46252	260934	8892105	0	102	143	107	150	150	150	146
(2) LTL2BA	178	1003	802	3360	30159	191668	5556159	5	0	137	49	161	157	156	142
(3) MoDeLLa	178	1297	647	4311	23874	216938	4193567	15	33	0	41	110	116	114	91
(4) LTL3BA	178	795	595	2209	21240	151373	4273646	0	23	126	0	149	153	152	140
(5) LTL3BA (det)	178	830	326	2405	14414	155716	2901474	0	0	10	5	0	76	75	63
(6) Spot	178	657	94	1615	10304	127792	2326271	1	6	15	5	30	0	1	1
(7) Spot (det)	178	662	88	1639	10414	128178	2328422	1	7	17	6	33	4	0	0
(8) Spot (no jump)	178	785	104	1874	12273	152592	2719360	12	28	40	27	70	61	57	0

they showed that their minimal WDBA are smaller than the non-deterministic BA produced by other translators. They also show that they improved the running times of Spin on a few verification tasks.[1]

We study how Spin's emptiness check can be helped by changing $\mathcal{A}$ in the next section. Improving the size of the product is one way to improve the performance of Spin (as the example of Section 4.5 illustrates), but there are also other aspects. For example, the location of accepting states have an influence too.

4. ANOTHER VIEW TO OPTIMIZATION: HELPING THE EMPTINESS CHECK

4.1 Emptiness Checks with Nested DFS

To check the emptiness of $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$, one should search for a cycle that is reachable from the initial state and that contains at least one accepting state. The emptiness check procedure used in Spin by default is based on two nested depth-first searches [17]: the main DFS, which we shall call *blue*, explores the product (on-the-fly) and every time it would backtrack from an accepting state s (i.e., all successors of s have been explored by the blue DFS) it starts a second, *red* DFS from s. If the red DFS reaches any state on the blue DFS search stack then a reachable and accepting cycle is found (since s is reachable from all states on the blue DFS search stack) and the algorithm reports it as a counterexample. Otherwise, the red DFS terminates and the blue DFS can continue. The two DFS always ignore states that have been completely explored by an instance of the red DFS, so a state is never visited more than twice.

As an extra optimization, if the blue DFS hits its own search stack by following a transition that is either going to or coming from an accepting state [13, 20], then an accepting cycle can be reported without even starting any red DFS. This can be effectively applied only on products with an accepting cycle.

When a counterexample exists in the product, the emptiness check may report it more or less rapidly depending on the order in which it has explored the transitions of the product. With any luck, the first transition selected at each step of the DFS will lead to an accepting cycle. Conversely, the first transitions followed might lead to a huge component of the product that just turns out to be a dead-end, and from which the emptiness check has to backtrack before finding the counterexample. As the selected transition order in $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ depends on the order of the transitions in the property automaton $\mathcal{A}_{\neg\varphi}$, this explains some of the huge differences noticed in Figure 1. Note that previous attempts to explore reordering of the transitions of $\mathcal{A}$ to help the emptiness check have been inconclusive [14], so we did not pursue this direction. (Furthermore the swarming techniques [18] used nowadays makes this topic even less attractive: in these approaches several threads compete to find a counterexample in $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ using a different, random transition order for $\mathcal{A}_{\neg\varphi}$.)

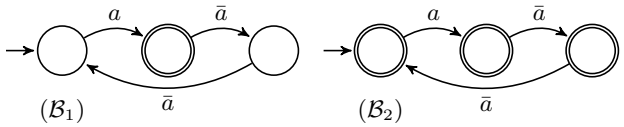

Figure 3: Automata for $a \wedge \mathsf{G}(a \rightarrow \mathsf{X}(\bar{a} \wedge \mathsf{X}(\bar{a} \wedge \mathsf{X}a)))$. $\mathcal{B}_1$ is inherently weak, $\mathcal{B}_2$ is weak.

4.2 Weak Automata

The optimization we just described, where the blue DFS can detect an accepting cycle without running a red DFS if it hits its own stack on (or from) an accepting state, suggests that of the two automata of Figure 3, $\mathcal{B}_2$ should be preferred. Indeed when the blue DFS reaches a state of its search stack in the product $\mathcal{S} \otimes \mathcal{B}_2$, it is guaranteed to come from (and go to) an accepting state, detecting the accepting cycle without

[1] We omitted their tool from our benchmark because (1) it only supports a subset of LTL, and (2) their optimization is implemented in Spot and both tools would therefore return the same automata. Besides, the subset of LTL does not include the formulae studied in Sections 4.3 and 4.5.

starting a red DFS. In the product $\mathcal{S} \otimes \mathcal{B}_1$ we might be less lucky if we close the cycle with the transition at the bottom of $\mathcal{B}_1$: in that case the product has to be explored a second time by the red DFS.

This example actually illustrates the distinction between weak automata and inherently weak automata. An *inherently weak automaton* is an automaton in which strongly connected components (SCCs) cannot mix accepting cycles with non-accepting cycles. A *weak automaton* is an inherently weak automaton in which the states of each SCC are either all accepting or all non-accepting. Any inherently weak automaton can evidently be transformed into an equivalent weak automaton [2].

Having more accepting states is not necessarily good from the point of view of the NDFS since a red DFS is started every time the blue DFS backtracks from an accepting state. However if an entire SCC is non-accepting, the first red DFS will cover it fully, and each successive red DFS will immediately return because it attempts to process a state that has already been seen by a previous red DFS.

4.3 Automata for $\mathsf{GF}a \wedge \mathsf{GF}b$

Figure 4 shows six different Büchi automata for the formula $\mathsf{GF}a \wedge \mathsf{GF}b$ produced by the considered tools. Note that if you ignore the exchange of a and b (which have symmetric purpose in the original formula), automata $\mathcal{C}_4$ and $\mathcal{C}_5$ differ only in the initial state and thus cannot be distinguished by any determinism-based or size-based metrics.

Table 3 captures data about Spin's runs on a model of the bakery mutual exclusion protocol taken from BEEM and the property automata of Figure 4. The propositions a and b describe situations that (different) pairs of processes are in the critical section at the same time. The protocol prevents such situation so neither a nor b is ever true in the model. We observe that in case of products with automata $\mathcal{C}_5$ and $\mathcal{C}_6$ (both produced by Spot), Spin explores each product twice because it triggers the red DFS from the initial state of the product. This is not the case for the other automata. This yields the following hypothesis: *When we suppose that there is no accepting cycle in the product, the automaton should keep its accepting states as far as possible from the initial state.* The further they are, the more chance we have that the product will never reach the state, and therefore no red DFS will be triggered.

For instance, if we ignore the renaming of atomic propositions, the automaton $\mathcal{C}_3$ could be obtained from $\mathcal{C}_6$ by *unrolling* the accepting cycle by one step, so that the cycle is entered on a non-accepting state, and the accepting state is actually the last one visited on the cycle.[2] This superfluous initial state only makes a negligible difference on the product, and does not incur any noticeable difference for Spin compared to $\mathcal{C}_1$, $\mathcal{C}_2$, or $\mathcal{C}_4$.

Similarly, if we do not expect an accepting cycle in the product, the inherently weak automaton $\mathcal{B}_1$ of Figure 3 could be changed by letting the right-most state be accepting instead of the middle one.

[2]This is not actually the reason why MoDeLLa produces $\mathcal{C}_3$. Internally, MoDeLLa translates the formula into a Büchi automaton with labels on states and has to deal with possibly multiple initial states. When it outputs an automaton, it *always* adds an extra initial state with copies of the outgoing transitions of all the original initial states, even if the original automaton had only one initial state. See also $\mathcal{D}_3$ of Figure 6 where s_0 and s_2 were the original initial states.

4.4 Translation Differences

Most LTL-to-BA translators follow a multi-steps procedure where they first translate a given LTL formula into a generalized Büchi automaton, often with transition-based acceptance (TGBA), such as those of Figure 5. Translators then *degeneralize* these automata to obtain a BA. Other simplification procedures may be applied to these automata, but it turns out that the last three automata of Figure 4 were all obtained by degeneralizing $\mathcal{G}_1$ in Figure 5, and their differences are due to choices made in the degeneralization procedure.

When degeneralizing a TGBA $\mathcal{G}$ with m acceptance sets $F_1, \ldots, F_m$ (the ● and ○ on the Figure 5), the structure of $\mathcal{G}$ is cloned $m+1$ times. Let us call each of these clones a level. For each state of level $i \leq m$, all transitions that were originally in F_i have their destination redirected to the next level, the destination of all transitions in level $m+1$ are redirected to level 1. Finally, all the states of the level $m+1$ are made accepting. The initial state can be put on any level.

This procedure ensures that words accepted by the degeneralized automaton correspond to words recognized by runs of $\mathcal{G}$ that visit all acceptance sets infinitely often. Accepting cycles in products involving these degeneralized automata will always involve at least $m+1$ states.

The degeneralization applied to $\mathcal{G}_1$ with the initial state on the last level and the acceptance sets ordered as ○, then ●, produces the automaton $\mathcal{C}_6$ of Figure 4. Recall that the edge labelled with $\top$ corresponds to the four edges labelled by $\bar{a}\bar{b}$, $\bar{a}b$, $a\bar{b}$, and ab in the original automaton $\mathcal{G}_1$.

An optimization introduced by Gastin and Oddoux [12] consists in jumping levels. If a transition of a level $i \leq m$ belongs to $F_i \cap \ldots \cap F_j$, its destination can be redirected directly to the level $j+1$. Similarly, if a transition from the level $m+1$ is in $F_1 \cap \ldots \cap F_j$, it can be redirected to the level $j+1$. Implementing this optimization gives automaton $\mathcal{C}_5$.

Changing the degeneralization order to ●, then ○, and putting the initial states on the first level would give automaton $\mathcal{C}_4$.

Often (but not in this example), jumping levels is a way to effectively avoid creating useless copies of some states. Another side effect of this optimization is that some accepting cycles may be shorter than $m+1$: the change effectively keeps the automaton as close to the accepting level as possible. If we are looking for counterexamples, $\mathcal{C}_5$ appear better than $\mathcal{C}_6$ because its accepting cycles are shorter on the average.

We recall that the initial state of a degeneralized automaton can be put on any level. For example, Giannakopoulou and Lerda [15] noticed that by changing the initial level, they could sometimes save some states, so they try to use both the first and the last level and keep the smallest automaton. In our example, $\mathcal{C}_4$ and $\mathcal{C}_5$ differ only by the choice of the initial level (and degeneralization order but this is negligible as a and b are symmetric in our problem), there is no size difference, and yet it makes a huge difference in the running time of Spin, as discussed in the previous section.

Another translation difference evidently comes from the difference between the generalized automata obtained from the LTL formula. In our case $\mathcal{C}_4$, $\mathcal{C}_5$, and $\mathcal{C}_6$ were obtained from $\mathcal{G}_1$ while $\mathcal{C}_1$ and $\mathcal{C}_2$ were obtained from $\mathcal{G}_2$. (The difference with Spin ($\mathcal{C}_1$) is that it does no level jumping from the accepting state.) The difference between $\mathcal{G}_1$ and $\mathcal{G}_2$ is caused

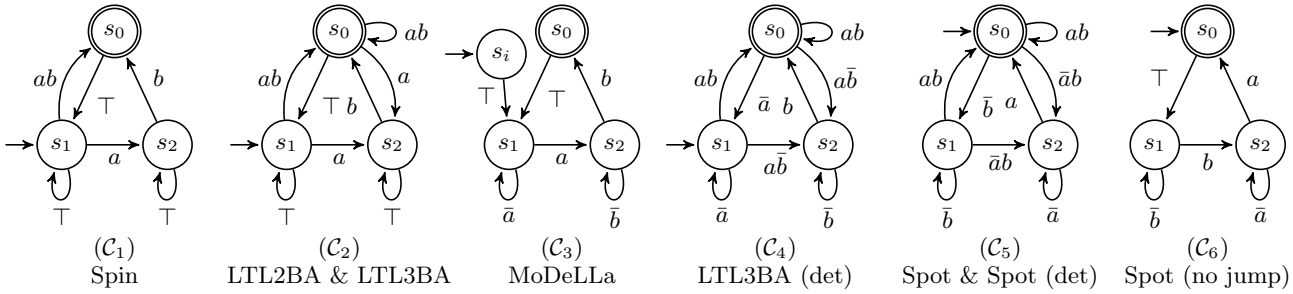

Figure 4: Automata for $\mathsf{GF}a \wedge \mathsf{GF}b$ generated by different tools and options.

Table 3: Statistics about generated automata and Spin's run on model `bakery.7.pm` and formula $\mathsf{GF}a \wedge \mathsf{GF}b$ where neither a nor b ever occurs in the model. The corresponding automata are shown in Fig. 4.

	automaton size				statistics from Spin's execution		
	states	ndst	edges	trans	stored states	visited trans	time
$\mathcal{C}_1$ Spin	3	2	6	17	27531713	95071k	88s
$\mathcal{C}_2$ LTL2BA & LTL3BA	3	3	8	20	27531713	95071k	99s
$\mathcal{C}_3$ MoDeLLa	4	0	6	16	27531714	95071k	109s
$\mathcal{C}_4$ LTL3BA (det)	3	0	8	12	27531713	95071k	101s
$\mathcal{C}_5$ Spot & Spot (det)	3	0	8	12	27531714	190143k	211s
$\mathcal{C}_6$ Spot (no jump)	3	0	5	12	27531714	190143k	191s

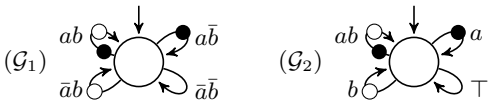

Figure 5: Two TGBA for $\mathsf{GF}a \wedge \mathsf{GF}b$. Accepting runs must visit ● and ○ infinitely often.

by choices made during the translation to favor deterministic states in the case of $\mathcal{G}_1$. In our example of Table 3, this improved determinism makes no difference since a and b are never true in the model.

4.5 Automata for $\neg(\mathsf{GF}a \rightarrow \mathsf{GF}b)$

We now focus on another concrete case: $\neg(\mathsf{GF}a \rightarrow \mathsf{GF}b)$ on mutex protocols. The formula without negation describes that if some process visits infinitely often the critical section, it infinitely often leaves it—this property holds in model `peterson.4.pm` and therefore Spin has to build the whole product to find that it contains no accepting cycle.

Table 4 shows a series of experiments of verification of the model `peterson.4.pm` against this formula, using different tools to obtain a Büchi automaton.

In this case, each tool produces a different automaton, as shown in the first part of Figure 6. Note again that automata $\mathcal{D}_2$ and $\mathcal{D}_4$ cannot be distinguished only by determinism and size metrics (see Table 4). They differ only in the target of the outgoing edge of s_0, yet we observe a significant difference in Spin's behaviors.

We actually use 12 different automata for this formula. The first seven of the table are generated by the considered tools. The other are handwritten by modifying the previous automata to explore which aspects of the automata make a significant difference in Spin's behavior as described further.

$\mathcal{D}_8$ is adapted from $\mathcal{D}_6$ by changing the degeneralization level on which we enter the SCC. $\mathcal{D}_9$ keeps the strong initial guard of $\mathcal{D}_6$ but then uses the accepting SCC of $\mathcal{D}_2$. $\mathcal{D}_{10}$ is a mix of $\mathcal{D}_6$ and $\mathcal{D}_2$ to observe the influence of the guards $\bar{a}\bar{b}$ compared to $\bar{b}$. $\mathcal{D}_{11}$ is a version of $\mathcal{D}_2$ in which the SCC is made deterministic as in $\mathcal{D}_6$. Finally, $\mathcal{D}_{12}$ fixes $\mathcal{D}_5$ by removing the spurious s_i.

Based on Table 4 we can group these automata in three categories, listed from the best to the worst with respect to Spin's performance. Before we discuss these categories, it is important to notice that in a model where a means *the process is in the critical section* and b means *the process leaves the critical section*, we can expect most of the state space to be labelled by $\bar{a}\bar{b}$.

$\mathcal{D}_6, \mathcal{D}_7, \mathcal{D}_8, \mathcal{D}_9$ Automata with the smallest number of transitions. Note that the *no jump* version ($\mathcal{D}_7$) and the one with a non-deterministic SCC ($\mathcal{D}_9$) both yields a few more states and transitions in the product, but the difference is not significant. The key property of these automata is that they can leave state s_0 only by reading $a\bar{b}$, whereas other automata are more permissive.

$\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_{10}, \mathcal{D}_{11}$ All these automata exhibit more nondeterminism on state s_0 and will enter the accepting SCC even after reading $\bar{a}\bar{b}$. However when this happens, they do not reach the accepting state before $a\bar{b}$ is read, so this limits the number of red DFS.

$\mathcal{D}_4, \mathcal{D}_5, \mathcal{D}_{12}$ These automata go from s_0 to the accepting state s_1 each time they read $\bar{a}\bar{b}$. This both makes the product unnecessarily large, but it also forces many calls to the red DFS every time a product state with property automaton state s_1 is backtracked. The nondeterminism in accepting SCC of $\mathcal{D}_4$ causes it to visits only slightly more states than the other two automata.

A comparison of automata $\mathcal{D}_6$ and $\mathcal{D}_{11}$ and their impact on Spin's performance show that the hypothesis of Section 4.3 cannot be used alone to select the best automaton.

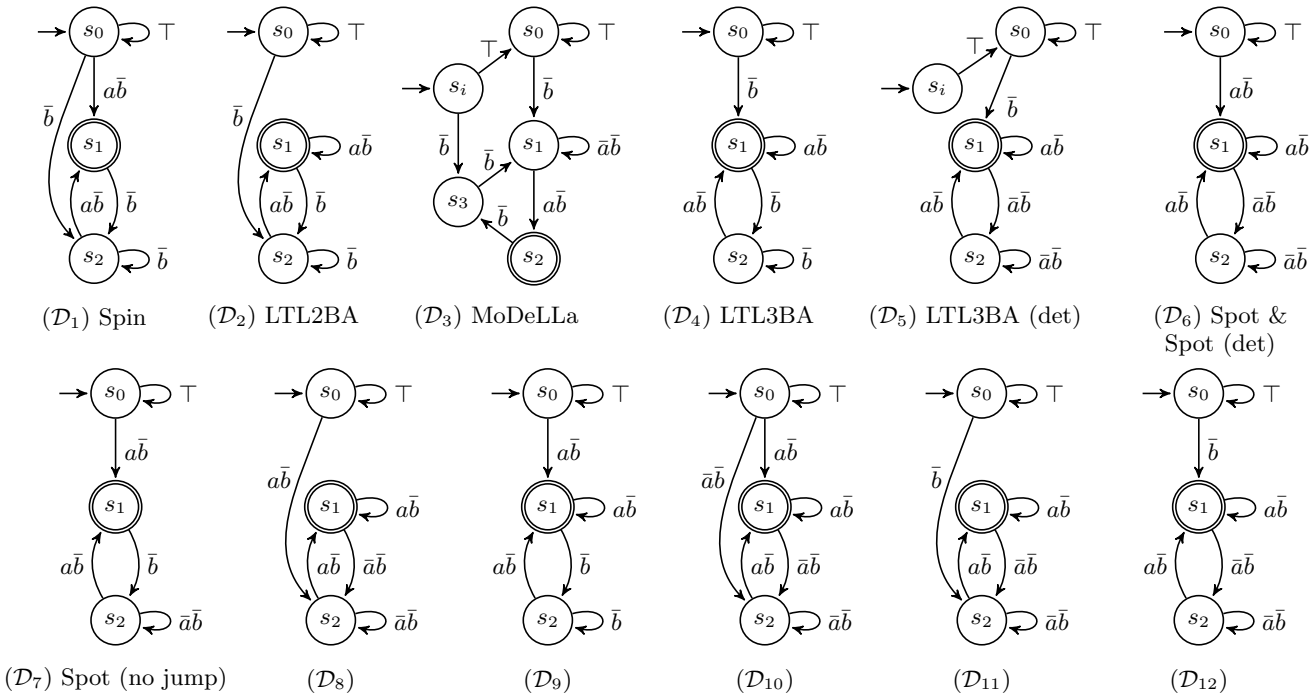

Figure 6: Automata for the formula $\neg(\mathsf{GF}a \to \mathsf{GF}b)$.

Table 4: Statistics about generated automata and Spin's run on the empty product between model `peterson.4.pm` and formula $\neg(\mathsf{GF}a \to \mathsf{GF}b)$. The corresponding automata are shown in Fig. 6.

	automaton size				statistics from Spin's execution		
	states	ndst	edges	trans	stored states	visited trans	time
$\mathcal{D}_1$ Spin	3	2	6	12	1577846	7680k	6.04s
$\mathcal{D}_2$ LTL2BA	3	3	6	12	1577440	7684k	5.95s
$\mathcal{D}_3$ MoDeLLa	5	2	8	18	1580893	7670k	6.13s
$\mathcal{D}_4$ LTL3BA	3	3	6	12	2299250	15583k	12.10s
$\mathcal{D}_5$ LTL3BA (det)	4	1	7	14	2297625	15561k	12.00s
$\mathcal{D}_6$ Spot	3	1	6	9	848641	2853k	2.26s
$\mathcal{D}_7$ Spot (no jump)	3	1	5	9	852094	2863k	2.34s
$\mathcal{D}_8$	3	1	6	9	848641	2853k	2.43s
$\mathcal{D}_9$	3	3	6	11	852094	2878k	2.43s
$\mathcal{D}_{10}$	3	1	7	10	1575844	7658k	7.38s
$\mathcal{D}_{11}$	3	1	6	10	1577440	7657k	7.07s
$\mathcal{D}_{12}$	3	1	6	10	2297625	15561k	12.30s

Indeed, $\mathcal{D}_6$ outperforms $\mathcal{D}_{11}$ even if the distance from the initial to the accepting state is shorter in $\mathcal{D}_6$. Here the more restrictive label of transition (s_0, s_1) in $\mathcal{D}_6$ plays an important role as well.

To sum up, if we suppose that there is no accepting cycle in the product, the automaton should

1. keep accepting states as far as possible from the initial state (compare $\mathcal{D}_{11}$ to $\mathcal{D}_{12}$) and

2. use more restrictive labels (compare $\mathcal{D}_6$ to $\mathcal{D}_{12}$)

in order to make the accepting states as hard to reach as possible. Moreover, making use of more restrictive labels can also help to reduce the product.

An appropriate metric taking these two factors into account, as well as an LTL-to-BA translation reflecting these hypotheses, are topics for our future research.

5. CONCLUSION

LTL-to-BA translators have several degrees of freedom when producing automata. Some of these choices have effects on the product with a system to be verified and also to the emptiness check of the product. However, these effects are difficult to predict. So far, most authors of LTL-to-BA translation tools have measured the performance of their tools by looking at the size of the output, sometimes also by looking at the size of products with random state spaces.

While building a small product generally helps the emptiness check, we have provided evidence that the size of $\mathcal{A}_{\neg\varphi}$ and even the size of $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ does not always correlate to the performance of the emptiness check of $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$. For instance, as Spin uses a Nested DFS, the locations of accepting states of $\mathcal{A}_{\neg\varphi}$ can have a dramatic impact to Spin's running time.

When a system $\mathcal{S}$ satisfies φ, i.e., $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ contains no accepting cycle, the best automaton for Spin to verify it should have accepting states that are hard to reach from the initial state, as it will lessen the chance that a red DFS is started. We observed that such a choice can be made during the degeneralization procedure, or by unrolling some accepting cycles.

On the contrary, if $\mathcal{S} \otimes \mathcal{A}_{\neg\varphi}$ contains an accepting cycle, Spin can find it faster if the accepting states of $\mathcal{A}_{\neg\varphi}$ are easy to reach from the initial state and the accepting cycles are short. Furthermore, the emptiness check can use an optimization if the automaton is weak.

We plan to examine these suggestions and potentially integrate them in future versions of our translators. Furthermore, we plan to devise a set of heuristics to select the best automaton of a given set of candidates. Clearly, LTL-to-BA translators should tune their output according to the purposed use of the BA: a BA used for bug finding need not to be the same as a BA used to prove correctness. Here we focused on the Nested DFS implementation of Spin, but many other emptiness checks exist. For instance, some emptiness checks based on the enumeration of SCCs are insensible to the location of accepting states on a cycle, so our suggestions should not be generalized blindly.

Another point that can be influenced by the property automaton is the size of the counterexample generated. The question of finding an automaton that is optimal from this point of view is left open by Gastin et al. [13].

6. ACKNOWLEDGMENTS

Authors would like to thank Vojtěch Rujbr for the initial inspiration and three anonynous referees for their suggestions. Fr. Blahoudek, M. Křetínský, and J. Strejček have been supported by The Czech Science Foundation, grant GBP202/12/G061.

7. REFERENCES

[1] T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS'12*, vol. 7214 of *LNCS*, pp. 95–109. Springer, 2012.

[2] B. Boigelot, S. Jodogne, and P. Wolper. On the use of weak automata for deciding linear arithmetic with integer and real variables. In *IJCAR'01*, vol. 2083 of *LNCS*, pp. 611–625. Springer, 2001.

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

[4] J.-M. Couvreur. On-the-fly verification of temporal logic. In *FM'99*, vol. 1708 of *LNCS*, pp. 253–271, Sept. 1999. Springer.

[5] C. Dax, J. Eisinger, and F. Klaedtke. Mechanizing the powerset construction for restricted classes of ω-automata. In *ATVA'07*, vol. 4762 of *LNCS*. Springer, Oct. 2007.

[6] A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In *ATVA'13*, vol. 8172 of *LNCS*, pp. 442–445, Oct. 2013. Springer.

[7] A. Duret-Lutz. LTL translation improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems*, 5(1/2):31–54, Mar. 2014.

[8] A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In *MASCOTS'04*, pp. 76–83, Oct. 2004. IEEE Computer Society Press.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *FMSP'98*, pp. 7–15, Mar. 1998. ACM Press.

[10] K. Etessami and G. J. Holzmann. Optimizing Büchi Automata. In *CONCUR'00*, vol. 1877 of *LNCS*, pp. 153–167. Springer, 2000.

[11] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Concur'00*, vol. 1877 of *LNCS*, pp. 153–167, 2000. Springer.

[12] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV'01*, vol. 2102 of *LNCS*, pp. 53–65, 2001. Springer.

[13] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *SPIN'04*, vol. 2989 of *LNCS*, pp. 92–108, Apr. 2004.

[14] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, Nov. 2005.

[15] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In *FORTE'02*, vol. 2529 of *LNCS*, pp. 308–326, Nov. 2002. Springer.

[16] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[17] G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In *SPIN'96*, vol. 32 of *DIMACS*. American Mathematical Society, May 1996.

[18] G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. *IEEE Transaction on Software Engineering*, 37(6):845–857, 2011.

[19] R. Pelánek. BEEM: benchmarks for explicit model checkers. In *SPIN'07*, vol. 4595 of *LNCS*, pp. 263–267. Springer, 2007.

[20] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *TACAS'05*, vol. 3440 of *LNCS*, Apr. 2005. Springer.

[21] R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *CHARME'03*, vol. 2860 of *LNCS*, pp. 126–140, Oct. 2003. Springer.

[22] F. Somenzi and R. Bloem. Efficient Büchi automata for LTL formulæ. In *CAV'00*, vol. 1855 of *LNCS*, pp. 247–263, 2000. Springer.

[23] H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer*, 4 (1):57–70, 2002.

[24] X. Thirioux. Simple and efficient translation from LTL formulas to Büchi automata. In *FMICS'02*, vol. 66(2) of *ENTCS*, July 2002. Elsevier.

[25] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff'94*, vol. 1043 of *LNCS*, pp. 238–266, 1996. Springer.