Compositional Approach to Suspension and Other Improvements to LTL Translation

Tomáš Babiak[2], Thomas Badie[1], Alexandre Duret-Lutz[1], Mojmír Křetínský[2], and Jan Strejček[2]

[1] LRDE, EPITA, Le Kremlin-Bicêtre, France
{badie,adl}@lrde.epita.fr
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbabiak,kretinsky,strejcek}@fi.muni.cz

Abstract. Recently, there was defined a fragment of LTL (containing fairness properties among other interesting formulae) whose validity over a given infinite word depends only on an arbitrary suffix of the word. Building upon an existing translation from LTL to Büchi automata, we introduce a compositional approach where subformulae of this fragment are translated separately from the rest of an input formula and the produced automata are composed in a way that the subformulae are checked only in relevant accepting strongly connected components of the final automaton. Further, we suggest improvements over some procedures commonly applied to generalized Büchi automata, namely over generalized acceptance simplification and over degeneralization. Finally we show how existing simulation-based reductions can be implemented in a signature-based framework in a way that improves the determinism of the automaton.

1 Introduction

Linear Temporal Logic (LTL) is a standard formalism for description of temporal properties of systems. LTL is mainly used as a specification formalism, typically in the context of model checking or control synthesis. Algorithms taking an LTL formula as input usually translate the formula (or its negation) to an equivalent *Büchi automaton* (BA) and subsequently work with that automaton.

Since the publication of the first algorithm translating LTL to Büchi automata [18], 30 years ago, dozens of papers presenting different translation algorithms and their optimizations have been published [e.g., 3, 11, 2, 12, 15, 10]. The quality of automata produced by current translators is much higher than before: automata are substantially smaller and are more often deterministic. In spite of this, we present several ideas to further improve the produced automata.

First, we introduce a *compositional approach to suspension* (or simply *compositional suspension*). It elaborates on the notion of suspension introduced recently [1]. The idea is based on the observation that validity of many interesting formulae (including fairness formulae) over an infinite word depends only on an arbitrary suffix of the word. We say that these formulae are *suspendable*. The original suspension technique, implemented in LTL3BA [1], was closely bound

to the translation of LTL to BA of Gastin and Oddoux [11]. The compositional suspension technique presented in this paper is more effective and more general as it can work on top of an arbitrary translation using *generalized Büchi automata* (GBAs) or *transition-based generalized Büchi automata* (TGBAs) as an intermediate (or a target) formalism. Note that nearly all LTL to BA translation algorithms use either a GBA or a TGBA in some form. (A notable exception is the translation of Fritz [10].) We present our techniques using the TGBA formalism as it encompasses GBAs, and it has been used by translators such as LTL2BA [11], Spot [6, 5], and LTL3BA [1] with a great success.

We also improve some post-processings used in LTL translators:

SCC-based simplifications of acceptance conditions reduce the number of acceptance sets in a TGBA by studying the relation between acceptance sets in each accepting *strongly connected component* (SCC) separately. The implementation of this technique requires careful fine-tuning, as it may greatly affect final Büchi automata produced by the next two procedures.

Transition-based simulation reductions We show how to implement *direct* and *reverse* simulation reductions of TGBAs in a signature-based framework, and show how to adjust these to improve determinism as a side-effect.

SCC-based degeneralization We suggest some improvements to the standard transformation of a TGBA into an equivalent BA.

The rest of the paper is organized as follows. The next section recalls the definition of LTL and several kinds of automata. Section 3 introduces the compositional suspension technique. Section 4 successively describes the other improvements. Experimental results are presented in Section 5.

2 Preliminaries

Let AP be a finite set of (atomic) propositions, and let $\mathbb{B} = \{\mathtt{ff}, \mathtt{tt}\}$ represent Boolean values. An assignment is a function $\ell : AP \to \mathbb{B}$ that valuates each proposition. $\mathbb{B}^{AP}$ is the set of all assignments of AP. X^* (resp. X^ω) denotes the set of finite (resp. infinite) sequences over a set X. In a sequence $\pi = \pi_1\pi_2\pi_3 \ldots \in X^\omega$, π_i denotes the ith element and $\pi_{i..} = \pi_i\pi_{i+1}\pi_{i+2}\ldots$. A *word* $w \in (\mathbb{B}^{AP})^\omega$ is an infinite sequence of assignments. For $\ell \in \mathbb{B}^{AP}$, let $\ell|_{AP'}$ denote the restriction of ℓ to $AP' \subseteq AP$; we extend this notation to words $(w|_{AP'})$ as well.

2.1 Linear temporal logic (LTL)

We define LTL with $\varphi ::= \mathtt{tt} \mid \mathtt{ff} \mid a \mid \bar{a} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{R}\,\varphi$ where $a \in AP$ and $\bar{a}$ denotes negation of a. We omit $\wedge$ in conjunctions of atomic propositions (e.g., $a\bar{b} \equiv a \wedge \bar{b}$). We allow negation only in front of atomic propositions as it is well known that any LTL formula can be rewritten into this form. The *validity* of a formula φ over a word $w \in (\mathbb{B}^{AP})^\omega$, written $w \models \varphi$, is defined by a structural induction on φ in the standard way. For example:

$$w \models a \quad \text{iff} \quad w_1(a) = \mathtt{tt};$$
$$w \models \varphi\,\mathsf{U}\,\psi \quad \text{iff} \quad \exists i \geq 1,\ (w_{i..} \models \psi \ \text{ and } \ \forall j \in \{1, \ldots, i-1\},\ w_{j..} \models \varphi).$$

We say that φ *holds at position* i of w iff $w_{i..} \models \varphi$.

2.2 Automata

A *labeled transition system* (LTS) is a tuple $\mathcal{S} = \langle AP, Q, q_0, \delta \rangle$ where AP is a finite set of atomic propositions, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \mathbb{B}^{AP} \times Q$ is the transition relation, labeling each transition by an assignment. As an implementation optimization, and to simplify illustrations, it is practical to use *edges* labeled by Boolean formulae to group *transitions* with same sources and destinations: for instance two *transitions* $(s_1, a\bar{b}, s_2)$ and (s_1, ab, s_2) will be represented by an *edge* from s_1 to s_2 and labeled by the Boolean formula a. We use the terms *transition* and *edge* to distinguish between these two representations.

An infinite sequence $\pi = (s_1, \ell_1, d_1)(s_2, \ell_2, d_2) \ldots \in \delta^\omega$ is a run of $\mathcal{S}$ if $s_1 = q_0$ and $\forall i \geq 1$, $d_i = s_{i+1}$. $\mathrm{Run}(\mathcal{S})$ denotes the set of all runs of $\mathcal{S}$. Let $\mathrm{Inf}_Q(\pi)$ (resp. $\mathrm{Inf}_\delta(\pi)$) denote the set of states (resp. transitions) that appear infinitely often in π, and let $\mathrm{Labels}(\pi) = \ell_1 \ell_2 \ldots \in (\mathbb{B}^{AP})^\omega$ be the word evaluated by π.

A *Büchi automaton* is a pair $\mathcal{B} = \langle \mathcal{S}, F \rangle$ where $\mathcal{S} = \langle AP, Q, q_0, \delta \rangle$ is an LTS and $F \subseteq Q$ is a set of accepting states. Let $\mathrm{Acc}(\mathcal{B}) = \{\pi \in \mathrm{Run}(\mathcal{S}) \mid \mathrm{Inf}_Q(\pi) \cap F \neq \emptyset\}$ denote the accepting runs of $\mathcal{B}$. The language of $\mathcal{B}$ is the set of words evaluated by accepting runs: $\mathscr{L}(\mathcal{B}) = \{\mathrm{Labels}(\pi) \mid \pi \in \mathrm{Acc}(\mathcal{B})\}$.

A *Transition-based Generalized Büchi automaton* (TGBA) is a pair $\mathcal{T} = \langle \mathcal{S}, F \rangle$ where $\mathcal{S} = \langle AP, Q, q_0, \delta \rangle$ is an LTS and $F \subseteq 2^\delta$ is a set of *acceptance sets* of transitions. Let $\mathrm{Acc}(\mathcal{T}) = \{\pi \in \mathrm{Run}(\mathcal{S}) \mid \forall Z \in F, \mathrm{Inf}_\delta(\pi) \cap Z \neq \emptyset\}$ denote the accepting runs of $\mathcal{B}$, i.e., runs of $\mathcal{S}$ whose transitions visit each acceptance set infinitely often. The language of $\mathcal{T}$ is the set of words evaluated by accepting runs: $\mathscr{L}(\mathcal{T}) = \{\mathrm{Labels}(\pi) \mid \pi \in \mathrm{Acc}(\mathcal{T})\}$. On figures, membership of transitions to acceptance sets is indicated using one colored marker (●, ○, ■, ...) per set.

A Büchi automaton $\mathcal{B} = \langle \mathcal{S}, F_\mathcal{B} \rangle$ can easily be converted into a TGBA $\mathcal{T} = \langle \mathcal{S}, F_\mathcal{T} \rangle$ such that $\mathscr{L}(\mathcal{B}) = \mathscr{L}(\mathcal{T})$ by setting $F_\mathcal{T} = \{\{(s, \ell, d) \in \delta \mid s \in F_\mathcal{B}\}\}$. A similar view can be used to interpret state-based generalized Büchi automata (which we do not define) as TGBAs. Although we describe our improvements on TGBAs, they adapt easily to these classes of Büchi automata with such views.

The reverse operation, *degeneralizing* a TGBA with multiple acceptance sets into a Büchi automaton, is discussed in Sec. 4.3.

A *promise automaton* is again a pair $\mathcal{P} = \langle \mathcal{S}, F \rangle$ where $F \subseteq 2^\delta$ is a set of *promise sets* of transitions. The runs accepted by a promise automaton are those which have no suffix that stays continuously in any promise set: $\mathrm{Acc}(\mathcal{P}) = \{\pi \in \mathrm{Run}(\mathcal{S}) \mid \forall Z \in F, \forall i \geq 1, \pi_{i..} \notin Z^\omega\}$. As expected, the language of $\mathcal{P}$ is $\mathscr{L}(\mathcal{P}) = \{\mathrm{Labels}(\pi) \mid \pi \in \mathrm{Acc}(\mathcal{P})\}$.

Because a run that does not visit infinitely often a set of transitions Z will have a suffix that stays continuously in the set $\delta \setminus Z$, $\mathcal{T} = \langle \mathcal{S}, F_\mathcal{T} \rangle$ can be converted into a promise automaton $\mathcal{P} = \langle \mathcal{S}, F_\mathcal{P} \rangle$ such that $\mathscr{L}(\mathcal{T}) = \mathscr{L}(\mathcal{P})$ by complementing the acceptance sets: $F_\mathcal{P} = \{\delta \setminus Z \mid Z \in F_\mathcal{T}\}$. The converse holds as well. The name of *promise automaton* comes from an interpretation of the elements of $F_\mathcal{P} = \{Z_1, \ldots, Z_n\}$ as promises: a transition in the set Z_i can be seen as *making* the promise Z_i. A promise Z_i is *fulfilled* by a run that does not stay in Z_i continuously, and a run is accepting if it fulfills all promises.

A strongly connected component (SCC) $C \subseteq Q$ is a non-empty set of states such that any ordered pair of states of C can be connected by a sequence of transitions. Let $C_\delta = \{(s, \ell, d) \in \delta \mid s \in C, d \in C\}$ denote the set of transitions induced by C. An SCC C is said to be *accepting* if: $C \cap F \neq \emptyset$ on a Büchi automaton, $\forall Z \in F$, $C_\delta \cap Z \neq \emptyset$ on a TGBA, $\forall Z \in F$, $C_\delta \cap Z \neq C_\delta$ on a promise automaton. With these definitions, any accepting run π is necessarily ultimately contained by some accepting SCC C, i.e., $\mathrm{Inf}_\delta(\pi) \subseteq C_\delta$.

3 Compositional Approach to Suspension

3.1 Suspendable formulae

A *suspendable* formula, originally called *alternating* formula [1][3], has at least one F and at least one G operator on each branch of its syntax tree. The formal definition is given by the following abstract syntax equations, where φ ranges over general LTL formulae. Besides suspendable formulae ξ, these equations also define *pure eventuality* formulae μ and *pure universality* formulae ν introduced by Etessami and Holzmann [8].

$$\begin{array}{l}\mu ::= \mathsf{F}\varphi \mid \mu \vee \mu \mid \mu \wedge \mu \mid \mathsf{X}\mu \mid \varphi\,\mathsf{U}\,\mu \mid \mu\,\mathsf{R}\,\mu \mid \mathsf{G}\mu \\ \nu ::= \mathsf{G}\varphi \mid \nu \vee \nu \mid \nu \wedge \nu \mid \mathsf{X}\nu \mid \nu\,\mathsf{U}\,\nu \mid \varphi\,\mathsf{R}\,\nu \mid \mathsf{F}\nu \\ \xi ::= \mathsf{G}\mu \mid \mathsf{F}\nu \mid \xi \vee \xi \mid \xi \wedge \xi \mid \mathsf{X}\xi \mid \varphi\,\mathsf{U}\,\xi \mid \varphi\,\mathsf{R}\,\xi \mid \mathsf{F}\xi \mid \mathsf{G}\xi\end{array}$$

The class of suspendable formulae contains many specification patterns frequently used in practical applications of LTL like model checking. For example, unconditional fairness $\mathsf{GF}\varphi$, weak fairness $\mathsf{FG}\varphi \to \mathsf{GF}\rho$ ($\equiv \mathsf{GF}(\varphi \to \rho)$), strong fairness $\mathsf{GF}\varphi \to \mathsf{GF}\rho$, and their negation can be easily transformed into suspendable formulae (our definition of LTL does not allow $\to$).

The following lemma states that a suspendable formula either holds at each position of a word or at none of them.

Lemma 1 ([1]). *Let ξ be a suspendable formula. For all $u \in (\mathbb{B}^{AP})^*$, $w \in (\mathbb{B}^{AP})^\omega$, we have $uw \models \xi \iff w \models \xi$.*

Consequently, every suspendable formula ξ satisfies $\xi \equiv \mathsf{X}\xi$. This property provides a theoretical base for the *suspension* technique [1] that was used to improve the translation of Gastin and Oddoux [11]. This translation uses a *very weak alternating automaton* (VWAA) and a TGBA as intermediate formalisms. States of the VWAA are identified with subformulae of the input formula. States of the TGBA are sets of VWAA states. Transitions leaving from a TGBA state M are computed as combinations of transitions leaving from the VWAA states in M. If M contains a suspendable subformula ξ, the corresponding VWAA state can be temporarily suspended: during the computation step, ξ is treated as $\mathsf{X}\xi$ and hence it has only one transition leading back to ξ. As a result, the number of transition combinations is reduced and a smaller automaton is produced. For

[3] We change the terminology here as the original name seems to be ambiguous.

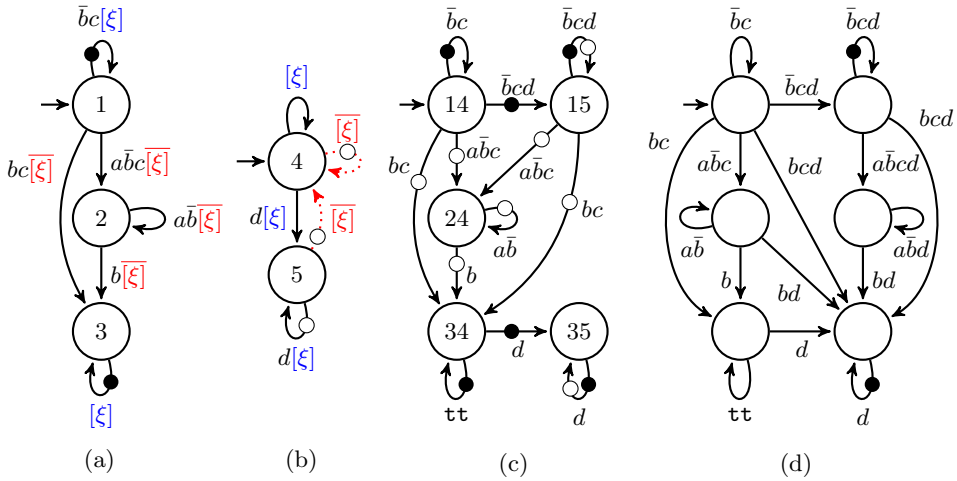


Fig. 1: (a) Skeleton TGBA for $((a \,\mathsf{U}\, b) \,\mathsf{R}\, c) \wedge \xi$. (b) Suspendable TGBA for $\xi = \mathsf{FG}d$ (equipped with suspending arcs). (c) Composition of the previous two automata. (d) Traditional translation of $((a \,\mathsf{U}\, b) \,\mathsf{R}\, c) \wedge \mathsf{FG}d$.

correctness, ξ should not be suspended forever during any accepting run of the TGBA, we therefore enable suspension only in TGBA states that are not on any accepting cycle. Because the detection of such a cycle is complicated as the TGBA is only under construction, a heuristic was used to detect some of the TGBA states not lying on any accepting cycle [1].

3.2 Translation with Compositional Suspension

We now present a new version of the suspension technique that has two advantages over the original one: it can be combined with all LTL translation algorithms producing a TGBA or GBA and it is more effective as it uses a more precise detection of TGBA states not lying on any accepting cycle.

To explain the general idea, consider an LTL formulae of the form $\varphi \wedge \xi$ where ξ is a suspendable formula. If $\mathcal{T}_\varphi$ and $\mathcal{T}_\xi$ are TGBAs for φ and ξ, we can construct an automaton for $\varphi \wedge \xi$ by composing $\mathcal{T}_\varphi$ and $\mathcal{T}_\xi$ using a synchronous product. However, ξ is suspendable, so by Lemma 1 we can suspend its verification by any finite prefix. In our composition we could delay the verification of ξ until $\mathcal{T}_\varphi$ has entered an accepting SCC. This remark calls for the implementation of a new synchronous product, that synchronizes $\mathcal{T}_\xi$ only in the accepting SCCs of $\mathcal{T}_\varphi$.

One way to describe this product is to introduce a new atomic proposition $[\xi]$ and its negation $\overline{[\xi]}$ to mark where the two automata should be synchronized. Figure 1(a) shows a TGBA for $\varphi = (a \,\mathsf{U}\, b) \,\mathsf{R}\, c$ equipped with these new properties and ready to be composed: transitions induced by accepting SCCs $\{1\}$ and $\{3\}$ carry the additional label $[\xi]$, while all other transitions have $\overline{[\xi]}$. We call such

an automaton a *skeleton automaton* for $\varphi \wedge \xi$ because it indicates the places where the suspended ξ should be composed. Figure 1(b) shows a TGBA $\mathcal{T}_\xi$ for $\xi = \mathsf{FG}d$ also equipped with the same labels: transitions from the original translation of ξ carry the $[\xi]$ label, and additional "suspending transitions" (the dotted arcs) have been added to reset the automaton to its initial state when $\mathcal{T}_\xi$ leaves an accepting SCC (and thus suspend checking ξ by another step). We call this a *suspendable automaton* for ξ. The synchronous product of both automata can then be stripped of all occurrences of the auxiliary proposition $[\xi]$ and its negation, and $[\xi]$ is removed from its set of atomic propositions. The resulting Fig. 1(c), should be compared to the automaton of Fig 1(d) that we would get by a traditional translation. The superfluous acceptance set we obtain can be easily removed, as explained in Sec. 4.1.

We now focus on constructing a skeleton automaton for an arbitrary formula φ that contains suspendable subformulae (not necessarily at the top level). We first replace every maximal suspendable subformula ξ of φ by the subformula $\mathsf{G}[\xi]$ with fresh auxiliary propositions $[\xi]$. The resulting formula, denoted φ' is translated into a TGBA $\mathcal{T}_{\varphi'}$. This automaton can directly be used as a skeleton for φ: whenever $\mathsf{G}[\xi]$ holds at some positions of a word accepted by this automaton, the product with a suspendable TGBA for ξ will check the validity of ξ on this word. Note that we do not say that validity of ξ will be checked exactly at the positions where $\mathsf{G}[\xi]$ holds. Indeed, this is not needed as ξ is a suspendable formula and thus it either holds at each position of a word or at none of them.

Even if $\mathcal{T}_{\varphi'}$ is a correct skeleton for φ, it is not what we typically use in the synchronous product with a suspendable TGBA for ξ. To avoid checking ξ whenever possible, we want to reduce the set of words w' accepted by the skeleton and such that $\mathsf{G}[\xi]$ holds at some positions of w'. We use two reductions:

- We replace $[\xi]$ with $\overline{[\xi]}$ on transitions that are not induced by any accepting SCC. (This is what we did in Fig 1(a).) The reduction is correct as for every word w' accepted by the original skeleton, there is a word w'' accepted by the reduced skeleton such that $w'|_{AP} = w''|_{AP}$ and $\mathsf{G}[\xi]$ holds at some positions of w' if and only if it holds at some position of w''. The last equivalence holds because we do not change transition labels in accepting SCCs.
- We remove transitions labeled with $[\xi]$ from the skeleton if they are not needed, i.e. there are analogous transitions that differ only in validity of $[\xi]$. Formally, we remove each transition (s, ℓ, d) such that $\ell([\xi]) = \mathtt{tt}$ if there exists a transition (s, ℓ', d) where $\ell'|_{AP} = \ell|_{AP}$, $\ell'([\xi]) = \mathtt{ff}$, and the two transitions belong to the same acceptance sets. This reduction is correct as for each w' accepted before this reduction and such that $\mathsf{G}[\xi]$ holds at some position of w', there is another word w'' accepted by the reduced skeleton and satisfying $w'|_{AP} = w''|_{AP}$. (Note that either $\mathsf{G}[\xi]$ holds at some positions of w'' too and then the product with suspendable automaton for ξ checks validity of ξ on $w'|_{AP}$ anyway, or $\mathsf{G}[\xi]$ does not hold at any positions of w'' and $w'|_{AP}$ satisfies φ regardless validity of ξ.)

We call `susp(`φ`)` the function that transforms φ into φ', `make_suspendable(`$\mathcal{T}, [\xi]$`)` the function that transforms a TGBA $\mathcal{T}$ for a suspendable subformula ξ into a suspendable automaton for ξ, and

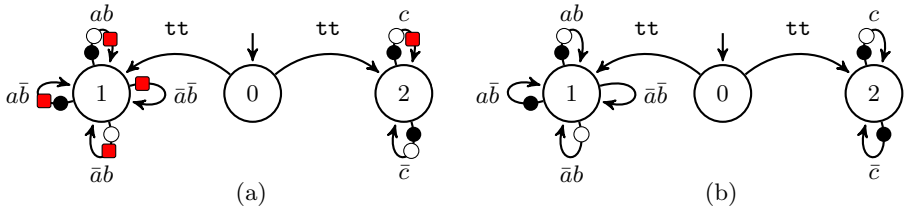


Fig. 2: (a) TGBA for $(\mathsf{GF}(a) \wedge \mathsf{GF}(b)) \vee \mathsf{GF}(c)$, with three acceptance sets denoted by ●, ○, and ■. (b) Same automaton after SCC-based acceptance simplification.

`reduce_skel`$(\mathcal{T}, \{\xi_1, ..., \xi_n\})$ the function that reduces a skeleton automaton for a set of suspendable subformulae. They will be used in Fig. 6.

4 Other Improvements

4.1 SCC-based simplifications of acceptance conditions

While simplifying the acceptance sets of an automaton does not immediately change the size of the underlying LTS, it can lead to smaller automata as it eases the job of simulation-based reductions (Sec. 4.2) and degeneralization (Sec. 4.3).

Let $\mathcal{T} = \langle \mathcal{S}, F \rangle$ be a TGBA with n acceptance sets: $F = \{Z_1, Z_2, \ldots, Z_n\}$. Let $\{A_1, \ldots, A_m\}$ denote the set of all accepting SCCs of $\mathcal{T}$ and let $A_\delta = A_{1\delta} \cup \ldots \cup A_{m\delta}$ be the set of all transitions induced by these accepting SCCs.

Because any accepting run will ultimately be contained in some accepting SCC, any transition outside of A_δ can be removed from the acceptance sets without changing the language. A typical simplification is therefore to restrict all Z_i to A_δ: we have $\mathscr{L}(\langle \mathcal{S}, F \rangle) = \mathscr{L}(\langle \mathcal{S}, \{Z_1 \cap A_\delta, \ldots, Z_n \cap A_\delta\} \rangle)$.

If there exists $i \neq j$ such that $Z_i \subseteq Z_j$, then any run that visits Z_i infinitely often will necessarily visit Z_j infinitely often. In other words, Z_j can be removed from F without changing the language: $\mathscr{L}(\langle \mathcal{S}, F \rangle) = \mathscr{L}(\langle \mathcal{S}, F \setminus \{Z_j\} \rangle)$.

If we define $U = \{Z_j \in F \mid \exists Z_i \in F, (Z_i \subsetneq Z_j) \vee (Z_i = Z_j \wedge j > i)\}$ to be the set of useless acceptance sets, we have $\mathscr{L}(\langle \mathcal{S}, F \rangle) = \mathscr{L}(\langle \mathcal{S}, F \setminus U \rangle)$. Note that the definition of U carefully keeps one copy when two sets are equal. We view this simplification as the standard way to diminish the number of acceptance sets in an automaton [16]. For instance after restricting the acceptance sets of Fig. 1c to the accepting SCCs $\{15\}_\delta \cup \{35\}_\delta$, one of ○ or ● can be removed (not both).

Detecting inclusion between acceptance sets at the automaton level fails to simplify the TGBA from Fig. 2(a): in this automaton there is no inclusion between acceptance sets. However, by considering such inclusions in each accepting SCC, we can notice that ■ is useless in SCC $A_1 = \{1\}$ (because ■ includes either ● or ○), while ● and ○ are both useless in SCC $A_2 = \{2\}$. We can therefore reorganize the acceptance sets of the automaton to use only two acceptance sets: Fig. 2(b) shows one possibility.

More formally, for an accepting SCC A_k, let $U_k = \{j \in \{1,\dots,n\} \mid \exists i \in \{1,\dots,n\}, (Z_i \cap A_{k\delta} \subsetneq Z_j \cap A_{k\delta}) \vee (Z_i \cap A_{k\delta} = Z_j \cap A_{k\delta} \wedge j > i)\}$ be the set of *indices* of useless acceptance sets in the sub-automaton induced by A_k, and let $N_k = \{1,\dots,n\} \setminus U_k$ be the set of *needed* acceptance sets. Because acceptance sets are defined for the whole automaton, we may not use a different number of acceptance sets for each SCC: $n' = \max_{k\in\{1,\dots,n\}} |N_k|$ acceptance sets are required to hold all *needed* acceptance sets. Let N'_k be a copy of N_k in which we have added $n' - |N_k|$ items from U_k. Then for each accepting SCC A_k, $|N'_k| = n'$ and let $\alpha_k : \{1,\dots,n'\} \to N'_k$ be any bijection. We can define the new acceptance sets $F' = \{Z'_1,\dots,Z'_{n'}\}$ as:

$$Z'_i = \bigcup_{k\in\{1,\dots,m\}} (Z_{\alpha_k(i)} \cap A_{k\delta}) \tag{1}$$

Then we have $\mathscr{L}(\langle \mathcal{S}, F\rangle) = \mathscr{L}(\langle \mathcal{S}, F'\rangle)$. In the example of Fig. 2(a), with $A_1 = \{1\}$ and $A_2 = \{2\}$, let us assume that ●, ○, and ■ respectively denote the acceptance sets Z_1, Z_2, and Z_3. We have $U_1 = \{3\}$, $N_1 = \{1,2\}$, $U_2 = \{1,2\}$, $N_2 = \{3\}$, $n' = 2$, and we define $N'_1 = N_1$, $N'_2 = N_2 \cup \{1\}$, $\alpha_1(1) = 1$, $\alpha_1(2) = 2$, $\alpha_2(1) = 1$, $\alpha_2(2) = 3$ to get the TGBA of Fig. 2(b).

Note that there is a lot of freedom in the definition of the bijective function α_k for each accepting SCC. In our implementation we make sure α_k is monotonic so that the order of the acceptance sets are preserved: we have found that this usually helps the degeneralization algorithm that is run afterwards. Furthermore, in accepting SCCs that require less than n' acceptance sets, the $n' - |N_k|$ extra sets that are added could be defined in many different ways: instead of reusing some of the *useless* acceptance sets, we could duplicate some of the *needed* ones (making $\alpha_k : \{1,\dots,n'\} \to N_k$ a surjection), or adding all transitions of $A_{k\delta}$ into the extra sets (at the price of more complex definitions). Our attempts at implementing these alternative definitions had a negative effect on the simulation-based reductions described in the next section.

Note that Somenzi and Bloem [16, Theorem 4] proposed another SCC-based acceptance simplification, that simplifies Fig. 2(a) differently. If we have $Z_i \cap A_{k\delta} \subseteq Z_j \cap A_{k\delta}$ for some i, j, k, they remove the transitions $A_{k\delta} \setminus Z_i$ from Z_j. While this reduces the size of Z_i, it does not yet change the number of acceptance sets. On Fig. 2(a), this would remove the bottom right loop from the sets ● and ○, after which it would be possible to detect that ■ includes all sets, and remove it.

4.2 Transition-based simulation reductions

Spot has an implementation of simulation-based reductions described by Somenzi and Bloem [16], but adapted to work on promise automata (easily converted to and from TGBAs, see Sec. 2.2) instead of BAs. Intuitively, *direct* simulation can merge states or remove transitions based on the inclusion of the sets of infinite runs *starting from* these states, while *reverse* simulation is based on the inclusion between sets of (finite or infinite) runs *leading to* these states.

Our implementation is a signature-based implementation of Moore's classic partition refinement algorithm: initially all states belong to the same class, and the partition is iteratively refined until fixpoint. Depending on the definition of the signature, we compute a bisimulation or simulation relation, direct or reverse.

Direct bisimulation: We first explain how to perform a signature-based, direct bisimulation of a promise automaton $\langle S, F\rangle$, using a setup inspired from Wimmer et al. [17]. The signature $\mathrm{sig}^i(q)$ of a state q is a Boolean function that describes the outgoing transitions of q, their membership to acceptance sets, and the class of their destination at iteration i.

If the acceptance sets are $F = \{Z_1, \ldots, Z_n\}$, and the partition of Q at iteration i is $P^i = \{C_1^i, \ldots, C_m^i\}$, we use Boolean variables $\hat{Z}_k$ and $\hat{C}_k$ to denote membership to the sets Z_k and C_k^i, and we define $\mathrm{sig}^i(s)$ as:

$$\mathrm{sig}^i(q) = \bigvee_{\substack{(s,\ell,d)\in\delta \\ q=s}} \ell \wedge \mathrm{Acc}(s,\ell,d) \wedge \mathrm{Class}^i(d);$$

$$\text{where} \quad \mathrm{Acc}(s,\ell,d) = \bigwedge_{\substack{Z_k\in F \\ (s,\ell,d)\in Z_k}} \hat{Z}_k \quad \text{and} \quad \mathrm{Class}^i(d) = \hat{C}_k \iff d \in C_k^i.$$

With this encoding, two states that have the same outgoing transitions (same labels, membership to acceptance sets, and destination class) will have the same signature. Also if a state q has two outgoing transitions $t_1 = (q, a, d_1)$ and $t_2 = (q, a, d_2)$ such that $t_1 \in Z_1$ is in a promise set but $t_2 \notin Z_1$ we have $\mathrm{sig}^i(q) = a \wedge ((\hat{Z}_1 \wedge \mathrm{Class}^i(d_1)) \vee \mathrm{Class}^i(d_2))$. If the two classes are the same, the signature simplifies to $\mathrm{sig}^i(q) = a \wedge \mathrm{Class}^i(d_2)$, as if t_1 had been merged into t_2. This simplification, correct on promise automata, would be incorrect on TGBAs.

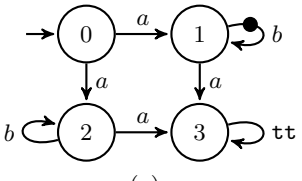
To compute a direct bisimulation relation we start with the partition $P^0 = \{Q\}$ that considers all states as equivalent, and then split the partition according to the signatures of the states: $P^{i+1} = \{\{s \in Q \mid \mathrm{sig}^i(s) \equiv \mathrm{sig}^i(q)\} \mid q \in Q\}$. Once a fixpoint has been reached (i.e., $P^j = P^{j+1}$ for some j), the partition provides the set of states that are (direct) bisimilar and can therefore be merged. It should be noted that the signature associated to each class can also be used to reconstruct the quotient automaton. By extension, let $\mathrm{sig}^{i-1}(C)$ denote the signature common to all states of the class $C \in P^i$.

Direct simulation: To perform the (direct) simulation of a promise automaton, we alter sig to include all the classes implied by the destination class:

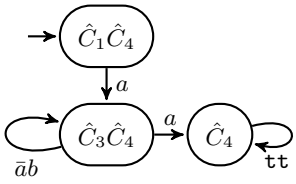
$$\mathrm{sig}^i(q) = \bigvee_{\substack{(s,\ell,d)\in\delta \\ q=s}} \ell \wedge \mathrm{Acc}(s,\ell,d) \wedge \mathrm{Implied}^i(d) \quad \text{where} \quad \mathrm{Implied}^i(d) = \bigwedge_{\substack{C_k^i\in P^i \\ \mathrm{sig}^{i-1}(d)\to\mathrm{sig}^{i-1}(C_k^i)}} \hat{C}_k$$

We fix $\mathrm{Implied}^0(q) = \hat{C}_1$ for all $q \in Q$ initially, as there is only one class. Then partition refinement can be iterated until both $P^k = P^{k+1}$ and $\mathrm{Implied}^k = \mathrm{Implied}^{k+1}$. The implication $\mathrm{sig}^{i-1}(d) \to \mathrm{sig}^{i-1}(C_k^i)$ can be tested easily since signatures are encoded as BDDs.

Figure 3 illustrates this reduction on an example. Although all states start in the same class, computing sig^0 is enough to separate all states into four



(a)



(b)

i	q	$\text{Class}^i(q)$	$\text{Implied}^i(q)$	$\text{sig}^i(q)$
0	0	$\hat{C}_1$	$\hat{C}_1$	$a\hat{C}_1 \vee a\hat{C}_1 = a\hat{C}_1$
	1	$\hat{C}_1$	$\hat{C}_1$	$b\hat{\bullet}\hat{C}_1 \vee a\hat{C}_1$
	2	$\hat{C}_1$	$\hat{C}_1$	$b\hat{C}_1 \vee a\hat{C}_1$
	3	$\hat{C}_1$	$\hat{C}_1$	$\hat{C}_1$
1	0	$\hat{C}_1$	$\hat{C}_1\hat{C}_4$	$a\hat{C}_2\hat{C}_3\hat{C}_4 \vee a\hat{C}_3\hat{C}_4 = a\hat{C}_3\hat{C}_4$
	1	$\hat{C}_2$	$\hat{C}_2\hat{C}_3\hat{C}_4$	$b\hat{\bullet}\hat{C}_2\hat{C}_3\hat{C}_4 \vee a\hat{C}_4$
	2	$\hat{C}_3$	$\hat{C}_3\hat{C}_4$	$b\hat{C}_3\hat{C}_4 \vee a\hat{C}_4$
	3	$\hat{C}_4$	$\hat{C}_4$	$\hat{C}_4$
2	0	$\hat{C}_1$	$\hat{C}_1\hat{C}_4$	$a\hat{C}_2\hat{C}_3\hat{C}_4 \vee a\hat{C}_3\hat{C}_4 = a\hat{C}_3\hat{C}_4$
	1	$\hat{C}_2$	$\hat{C}_2\hat{C}_3\hat{C}_4$	$b\hat{\bullet}\hat{C}_2\hat{C}_3\hat{C}_4 \vee a\hat{C}_4$
	2	$\hat{C}_3$	$\hat{C}_3\hat{C}_4$	$b\hat{C}_3\hat{C}_4 \vee a\hat{C}_4$
	3	$\hat{C}_4$	$\hat{C}_4$	$\hat{C}_4$

(c)

Fig. 3: (a) A promise automaton to simplify. (b) The result of direct simulation reduction. (c) Detailed steps for the signature-based direct simulation.

classes. The refinement stops at iteration $i = 2$ because $\text{Class}^2 = \text{Class}^1$ and $\text{Implied}^2 = \text{Implied}^1$. The signatures computed at the last iteration can be used to reconstruct the automaton. Especially, the edge $(0, a, 1)$ in the original automaton is dominated by edge $(0, a, 2)$ in the computation of $\text{sig}^2(0)$ (intuitively, the suffixes accepted via $(0, a, 1)$ are included in those accepted via $(0, a, 2)$), so only the latter edge appears in the resulting signature and in the final automaton.

An additional trick can be used to improve the determinism of the constructed automaton. Because $\text{sig}^2(2) = b\hat{C}_3\hat{C}_4 \vee a\hat{C}_4$ is equivalent to $\bar{a}b\hat{C}_3\hat{C}_4 \vee a\hat{C}_4$, the self-loop to state 2 (represented by $\hat{C}_3\hat{C}_4$ in the signature) can be labeled by $\bar{a}b$ instead of just b. In practice, for each state q we iterate over all assignments $f \in \mathbb{B}^{AP}$, and compute the possible destinations by rewriting $\text{sig}^i(q) \wedge f$ as an irredundant sum of products.

Reverse simulation: A reverse simulation can be built and used similarly by computing a signature using the incoming transitions:

$$\text{sig}^i(q) = \text{Init}(q) \vee \bigvee_{\substack{(s,\ell,d)\in\delta \\ q=d}} \ell \wedge \text{Acc}(s,\ell,d) \wedge \text{Implied}^i(d) \quad \text{with} \quad \text{Init}(q) = \begin{cases} \hat{I} \text{ if } q = q_0; \\ \texttt{ff} \text{ else.} \end{cases}$$

Because the reverse simulation has to distinguish finite prefixes from infinite prefixes we use an extra Boolean variable $\hat{I}$ to distinguish the initial state.

In practice we alternate direct and reverse simulations until the automaton is no longer reduced. Most of the time only one iteration is needed (meaning that we do the second iteration just to discover that the produced automaton has the same size). As an optimization, we abort this loop when the automaton produced

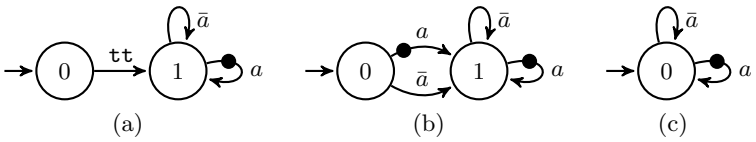


Fig. 4: Three equivalent TGBAs. (a) is obtained from (b) with the acceptance simplifications of Sec. 4.1. (c) is obtained from (b) using the simulation reductions of Sec. 4.2, however the latter reductions are unable to reduce (a) into (c).

by the direct simulation is deterministic: the reverse simulation cannot improve a deterministic automaton since all prefixes leading to a state are unique.

Other simulations have been suggested, such as *fair* or *delayed* simulation [9], both relaxing the handling of acceptance conditions, and these have also been extended to (state-based) generalized Büchi automata [13]. All of these are presented in a game-theoretic framework that is not straightforward to implement, especially in the generalized Streett game version required for generalized Büchi automata. Conversely, our implementation of direct and reverse simulation easily deals with TGBAs (when first converted as promise automata), augments the determinism as a side-effect, and was simple to implement because it uses the same BDD framework that Spot is already using for the LTL translation.

Although the operations described in Section 4.1 simplify the SCC-based acceptance conditions of a TGBA, there are situations where it worsens the results of the simulation-based reductions. A typical example is given by Fig. 4.

Since Couvreur's translation can produce automata with a configuration similar to Fig. 4(b), we use an alternative acceptance simplification that preserves the acceptance of all transitions entering an accepting SCC. This corresponds to replacing $A_{k\delta}$ by $\{(s,\ell,d) \in \delta \mid d \in A_k\}$ in equation (1). In our tests, this is always favorable to the simulation.

Unfortunately, the situation depicted by Fig. 4(a) also occurs in the output of some translations, even before acceptance simplification. This is even more frequent with the compositional approach to suspension presented in Sec. 3.

4.3 SCC-based degeneralization

While any Büchi automaton can be converted into a TGBA without altering the underlying LTS (see Sec. 2.2), the reverse is not generally true.

A TGBA $\mathcal{T} = \langle \mathcal{S}, F\rangle$ with $\mathcal{S} = \langle AP, Q, q_0, \delta\rangle$ and $F = \{Z_1, \ldots, Z_n\}$ can be degeneralized into a BA $\mathcal{B} = \langle \mathcal{S}', F'\rangle$ with $\mathcal{S}' = \langle AP, Q', q_0', \delta'\rangle$ as follows [11, 12]:

- $Q' = Q \times \{0, \ldots, n\}$, i.e., the original automaton is cloned in $n+1$ levels,
- $F' = Q \times \{n\}$, i.e., states from the last level are accepting,
- $\delta' = \{((s,j),\ell,(d, L_j((s,\ell,d)))) \mid (s,\ell,d) \in \delta, j \in \{0,\ldots,n\}\}$ where

$$L_j(t) = \begin{cases} 0 & \text{if } j = n; \\ j+1 & \text{if } t \in Z_{j+1}; \\ j & \text{otherwise.} \end{cases}$$

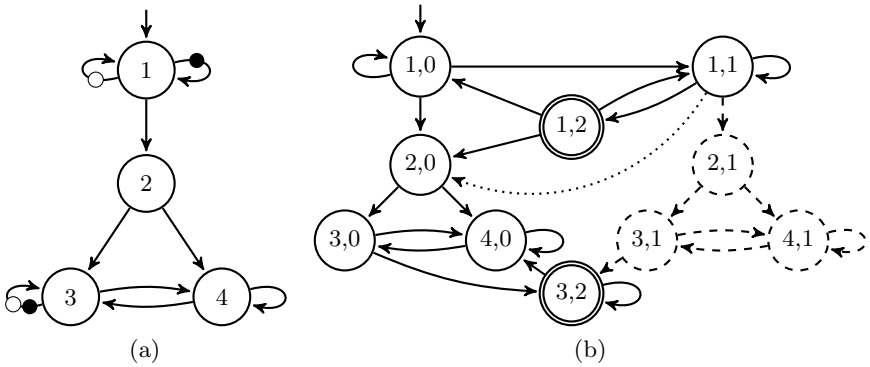


Fig. 5: Example of degeneralization of the TGBA (a) with $F = \{\bullet, \circ\}$ taken in this order. Transition labels are omitted for clarity. The automaton (b) with the dashed part is obtained by the classical degeneralization (with jumping of levels). Our redefinition replaces the dashed part by the dotted transition.

Note: the definition of q'_0 and the example of Fig. 5 are bogus in the proceedings of SPIN'13. This version of the paper is fixed.

i.e., for each level $j < n$ the outgoing transitions that belong to Z_{j+1} are redirected to the next level and all outgoing transitions from the last level are redirected to the first one.

– $q'_0 = (q_0, 0)$, i.e., the initial state is on the first level (any level works).

This leveled setup guarantees that any accepting path in $\mathcal{B}$ correspond to an infinite path that sees all acceptance conditions infinitely often in $\mathcal{T}$. If $\mathcal{T}[q]$ denotes the automaton $\mathcal{T}$ in which the initial state has been changed to q, we have $\mathscr{L}(\mathcal{T}[q]) = \mathscr{L}(\mathcal{B}[(q, j)])$ for all states $q \in Q$ and all levels $j \in \{0, \ldots, n\}$.

The classical optimization is to "jump levels", i.e., when a transition from level $j < n$ belongs to acceptance sets Z_{j+1}, Z_{j+2}, and Z_{j+3}, it can be redirected to the level $j + 3$. This corresponds to the following redefinition of L_j:

$$L_j(t) = \begin{cases} \max\{k \in \{j, \ldots, n\} \mid t \in Z_{j+1} \cap \ldots \cap Z_k\} & \text{if } j < n; \\ \max\{k \in \{0, \ldots, n\} \mid t \in Z_1 \cap \ldots \cap Z_k\} & \text{if } j = n. \end{cases}$$

Of course only the reachable part of $\mathcal{B}$ is constructed, so it is not frequent to construct the maximum number of $(n + 1) \times |Q|$ states. Fig. 5 applies the above definition to an example. The transition from $(3, 0)$ to $(3, 2)$ jumps level 1.

This construction offers several degrees of freedom: for instance there are $n!$ possible orderings of the acceptance sets, and $n + 1$ possible levels for the initial state. Giannakopoulou and Lerda [12] perform two degeneralizations starting respectively at level 0 and n, then they keep the best. We are not aware on any work on the selection of a suitable ordering. Empirical evidence shows that the order in which these sets are created during translation is often favorable to the degeneralization (the reverse order, at least, is catastrophic), this is why Sec. 4.1 defines α in a way that preserves the ordering.

Staying away from these combinatorial possibilities, we suggest two ideas to improve degeneralization procedures: *level reset* and *level caching*. Both require knowledge of the set $\{A_1, \ldots, A_m\}$ of accepting SCCs of $\mathcal{T}$.

Level reset: Since non-accepting SCCs of $\mathcal{T}$ do not contain accepting cycles (by definition), they do not need to be cloned on different levels. Therefore all transitions that are not induced by any accepting SCC, are directed to level 0.

Level caching: Consider a transition (s, ℓ, d) that enters an accepting SCC ($\exists i,\ s \notin A_i \wedge d \in A_i$): as with the initial state, the level associated to d can be set to any arbitrary value. If a copy of d already exists on some level, we should start on that level to avoid creating a new one. This optimization is of course affected by the order chosen to construct the degeneralized automaton (we do a simple DFS but there could be room for improvement).

These two optimizations can be implemented with $L_j((s, \ell, d)) =$

$$\begin{cases} \max\{k \in \{j, \ldots, n\} \mid (s, \ell, d) \in Z_{j+1} \cap \ldots \cap Z_k\} & \text{if } j < n \wedge \exists i,\ (s, d) \in A_i^2; \\ \max\{k \in \{0, \ldots, n\} \mid (s, \ell, d) \in Z_1 \cap \ldots \cap Z_k\} & \text{if } j = n \wedge \exists i,\ (s, d) \in A_i^2; \\ 0 & \text{if } \nexists i,\ d \in A_i; \\ x & \text{if } \exists i,\ s \notin A_i \wedge d \in A_i. \end{cases}$$

Where x is any level such that the state (d, x) already exists, or 0 otherwise.

On the example of Fig. 5, the *level reset* alone is enough to replace transition $((1,1), \ell, (2,1))$ by transition $((1,1), \ell, (2,0))$, therefore avoiding state $(2,1)$ and all its descendants. Using *level caching* without *level reset*, and assuming the descendants of $(2,0)$ have been built before those of $(2,1)$, then state $(2,1)$ would be connected to states $(3,0)$ and $(4,0)$ instead of states $(3,1)$ and $(4,1)$. It is hard to find a small example to illustrate that both optimizations are useful together: the smallest such occurrence in our benchmarks has 20 states.

5 Experimental Results

5.1 Translation Scenarios

All of the above improvements are implemented in Spot 1.1[4] on top of Couvreur's LTL to BA translation algorithm [2] (denoted by `Cou99`(φ) in the sequel although it has been regularly improved over the past years [5]). Besides the techniques discussed in this paper, Spot implements the WDBA-minimization algorithm of Dax et al. [4] that converts any TGBA representing an *obligation property* [14] into a minimal Weak Deterministic Büchi Automaton. The corresponding function `WDBA_minimize`$(\mathcal{T}, \varphi)$ requires the formula φ represented by automaton $\mathcal{T}$ to check the validity of the minimized automaton.

There are cases where the deterministic BA produced by WDBA-minimization is bigger than the nondeterministic automaton obtained via simulation and degeneralization. In the context of model checking, it is not clear when a deterministic

[4] `http://spot.lip6.fr/`. After download and installation, see the man pages of `ltl2tgba(1)` and `spot-x(7)` for the options to enable the algorithms discussed here, and see also `bench/spin13/README`.

	"small size" scenario	"determinism" scenario

composition	$\varphi', \{\xi_1, ..., \xi_n\} \leftarrow \texttt{susp}(\varphi)$
	$\mathcal{T} \leftarrow \texttt{Cou99}(\varphi')$
	(S) $\mathcal{T} \leftarrow \texttt{iter\_simulations}(\mathcal{T})$
	$\mathcal{T} \leftarrow \texttt{reduce\_skel}(\mathcal{T}, \{\xi_1, ..., \xi_n\})$
	for $\xi \in \{\xi_1, ..., \xi_n\}$ do:
	  $\mathcal{T}_\xi \leftarrow \texttt{Cou99}(\xi)$
	(S) $\mathcal{T}_\xi \leftarrow \texttt{iter\_simulations}(\mathcal{T}_\xi)$
	  $\mathcal{T}_\xi \leftarrow \texttt{make\_suspendable}(\mathcal{T}_\xi, [\xi])$
	  $\mathcal{T} \leftarrow \texttt{product}(\mathcal{T}, \mathcal{T}_\xi)$
		$\varphi', \{\xi_1, ..., \xi_n\} \leftarrow \texttt{susp}(\varphi)$
		$\mathcal{T}' \leftarrow \texttt{Cou99}(\varphi')$
		$\mathcal{T} \leftarrow \texttt{WDBA\_minimize}(\mathcal{T}', \varphi')$
		if $\mathcal{T}$ could not be built:
		(S) $\mathcal{T} \leftarrow \texttt{iter\_simulations}(\mathcal{T}')$
		$\mathcal{T} \leftarrow \texttt{reduce\_skel}(\mathcal{T}, \{\xi_1, ..., \xi_n\})$
		for $\xi \in \{\xi_1, ..., \xi_n\}$ do:
		  $\mathcal{T}_\xi \leftarrow \texttt{Cou99}(\xi)$
		(S) $\mathcal{T}_\xi \leftarrow \texttt{iter\_simulations}(\mathcal{T}_\xi)$
		  $\mathcal{T}_\xi \leftarrow \texttt{make\_suspendable}(\mathcal{T}_\xi, [\xi])$
		  $\mathcal{T} \leftarrow \texttt{product}(\mathcal{T}, \mathcal{T}_\xi)$

post-processings	$\mathcal{T} \leftarrow \texttt{prune\_dead\_SCCs}(\mathcal{T})$
	(A) $\mathcal{T} \leftarrow \texttt{acc\_simplify}(\mathcal{T})$
	$\mathcal{T}_1 \leftarrow \texttt{WDBA\_minimize}(\mathcal{T}, \varphi)$
	(S) $\mathcal{T} \leftarrow \texttt{iter\_simulations}(\mathcal{T})$
	(D) $\mathcal{T} \leftarrow \texttt{degeneralize}(\mathcal{T})$
	(B) $\mathcal{T} \leftarrow \texttt{iter\_simulations}(\mathcal{T})$
	if $\mathcal{T}_1$ could be built:
	  return $\texttt{smallest}(\mathcal{T}, \mathcal{T}_1)$
	return $\mathcal{T}$
		$\mathcal{T} \leftarrow \texttt{prune\_dead\_SCCs}(\mathcal{T})$
		(A) $\mathcal{T} \leftarrow \texttt{acc\_simplify}(\mathcal{T})$
		$\mathcal{T}_1 \leftarrow \texttt{WDBA\_minimize}(\mathcal{T}, \varphi)$
		if $\mathcal{T}_1$ could be built:
		  return $\mathcal{T}_1$
		(S) $\mathcal{T} \leftarrow \texttt{iter\_simulations}(\mathcal{T})$
		(D) $\mathcal{T} \leftarrow \texttt{degeneralize}(\mathcal{T})$
		(B) $\mathcal{T} \leftarrow \texttt{iter\_simulations}(\mathcal{T})$
		return $\mathcal{T}$

Fig. 6: Two translation scenarios that use compositional suspension (denoted "Comp" in the sequel) to produce a BA. Automata are stored as TGBAs even when they represent BAs. The scenarios without compositional suspension (denoted as "Cou99") arise by replacing all the composition lines by $\mathcal{T} \leftarrow \texttt{Cou99}(\varphi)$. To produce a TGBA instead of a BA, we omit lines (D) and (B).

automaton should be favored over a small one. For instance, Sebastiani and Tonetta [15] have shown that their larger and more deterministic automata yield smaller synchronized products with a model than the smaller automata produced by Gastin and Oddoux [11]. Spot implements two translation scenarios: the "small size" scenario tries to reduces the size of the automaton, while the "determinism" scenario tries to reduces the number of nondeterministic states. According to our experience with Spot, automata produced by our "small size" scenario tends to give smaller synchronized products.

Fig. 6 shows how the different techniques we have presented are chained in these two scenarios. The function `prune_dead_SCCs` is a classical optimization that removes states that may not reach an accepting SCC. `susp`, `reduce_skel`, `make_suspendable` correspond to operations defined in Sec. 3.2. We will use keys (S),(A),(D),(B) to denote lines that are enabled or disabled in our experiments. For acceptance simplification (A) and degeneralization (D), we write (a) and (d) to indicate that old definitions are used. For instance "Cou99 (a)" means that Spot's implementation of Couvreur's translation was used to translate the formulae, and that the only post-processings performed were `prune_dead_SCCs`,

				"small size" scenario						"determinism" scenario					
				$\|Q\|$	$\|\delta\|$	$\|F\|$	ns	nA	time	$\|Q\|$	$\|\delta\|$	$\|F\|$	ns	nA	time
known.ltl: 184 formulae	TGBA	1	Cou99 (a)	672	10921	198	113	49	7.09	676	10805	198	105	45	7.13
		2	Cou99 (A)	672	10921	195	113	49	7.24	676	10805	195	105	45	7.38
		3	Cou99 (AS)	636	9848	195	88	50	7.50	641	9958	195	82	45	7.34
		4	**Comp** (AS)	636	9838	195	85	50	7.63	644	9968	195	79	45	7.53
	BA	5	Cou99 (ad)	717	11653	184	124	49	6.93	721	11537	184	116	45	6.84
		6	Cou99 (Ad)	717	11653	184	124	49	6.94	721	11537	184	116	45	6.97
		7	Cou99 (ASd)	678	10511	184	97	50	7.28	683	10621	184	91	45	7.02
		8	Cou99 (ASD)	675	10463	184	97	50	7.35	680	10573	184	91	45	7.08
		9	Cou99 (ASDB)	673	10362	184	95	49	7.50	678	10472	184	89	44	7.14
		10	**Comp** (ASDB)	673	10352	184	92	49	7.56	687	10530	184	86	44	7.32
weak3.ltl: 100 formulae	TGBA	1	Cou99 (a)	749	116312	361	324	92	6.81	749	116312	361	324	92	6.83
		2	Cou99 (A)	743	115104	357	319	90	8.89	743	115104	357	319	90	8.94
		3	Cou99 (AS)	618	84603	355	237	86	10.42	618	84603	355	237	86	10.36
		4	**Comp** (AS)	617	83875	355	225	87	7.51	647	90771	355	195	69	7.84
	BA	5	Cou99 (ad)	2030	299376	100	776	92	7.75	2030	299376	100	776	92	7.76
		6	Cou99 (Ad)	2018	296904	100	765	90	8.50	2018	296904	100	765	90	8.55
		7	Cou99 (ASd)	1700	212984	100	549	86	10.04	1700	212984	100	549	86	9.92
		8	Cou99 (ASD)	1565	193157	100	493	86	9.99	1565	193157	100	493	86	9.90
		9	Cou99 (ASDB)	1525	188873	100	435	86	11.32	1525	188873	100	435	86	11.41
		10	**Comp** (ASDB)	1530	188939	100	424	87	8.48	1588	201611	100	387	69	8.51
strong2.ltl: 100 formulae	TGBA	1	Cou99 (a)	6237	3524004	261	4633	100	82.32	6237	3524004	261	4633	100	82.65
		2	Cou99 (A)	6183	3485348	257	4583	100	151.17	6183	3485348	257	4583	100	151.49
		3	Cou99 (AS)	1900	508972	255	879	100	178.43	1900	508972	255	879	100	178.68
		4	**Comp** (AS)	1731	434812	255	703	100	50.41	1801	464412	255	675	100	46.80
	BA	5	Cou99 (ad)	8207	3928868	100	5379	100	114.24	8207	3928868	100	5379	100	114.38
		6	Cou99 (Ad)	8083	3876308	100	5290	100	151.76	8083	3876308	100	5290	100	151.57
		7	Cou99 (ASd)	3488	782324	100	1368	100	178.83	3488	782324	100	1368	100	178.73
		8	Cou99 (ASD)	3330	745280	100	1292	100	177.44	3330	745280	100	1292	100	178.14
		9	Cou99 (ASDB)	3259	727416	100	1211	100	181.43	3259	727416	100	1211	100	182.34
		10	**Comp** (ASDB)	3091	668768	100	1039	100	53.92	3201	713152	100	991	100	49.98
		11	`ltl3ba`	5389	2473408	100	5041	100	2.38	8660	2281988	100	4515	100	4.77
		12	`ltl3ba` susp.	5298	2458372	100	4950	100	2.38	5418	1424964	100	2409	100	2.56

Table 1: Results for selected combinations of the presented techniques. Numbers are accumulated over all translated formulae. Smaller numbers are better everywhere. Keys A/a,S,D/d,B indicate when the corresponding lines of Fig. 6 have been enabled a/d respectively denote the original acceptance simplification and degeneralization while A/D apply the definitions from this paper. Compositional suspension is only enabled on "**Comp**" lines.

the old version of `acc_simplify`, and WDBA-minimization when applicable; especially, no simulation-based reduction or degeneralization was performed.

We note that suspendable formulae are not obligation properties, so the presence of a suspendable subformulae prevents the application of WDBA-minimization except in pathological cases.

5.2 Experiments

Table 1 presents results of selected combinations of the presented techniques applied according to the two scenarios to three different sets of formulae.[5] For

[5] More measures and details at `http://www.lrde.epita.fr/~adl/spin13/`.

each configuration, scenario, and set of formulae we show the cumulative size of the automata produced for formulae in the set, namely numbers of states, transitions, acceptance sets, nondeterministic states (ns), and nondeterministic automata (nA). We also provide total translation time. Grey rectangles mark the best results: smallest automata for the "small size" scenario and automata with the least nondeterminism for the "determinism" scenario.

`known.ltl` contains 92 formulae and their negation, collected from the literature [7, 16, 8]. 122 of these 184 formulae describe obligation properties, for which WDBA minimization computes a minimal deterministic automaton during the post-processing. Only 14 formulae of the set require more than one acceptance set for the translation. The potential for improvement on this set is very thin.

`weak3.ltl` contains 100 formulae combined with a weak fairness hypothesis. Formulae have the form $\varphi_i \wedge \mathsf{GF}a \wedge \mathsf{GF}b \wedge \mathsf{GF}c$ where φ_i is a random LTL formula with a syntax tree of 15..20 nodes, using up to 6 atomic propositions. The fairness hypothesis $\mathsf{GF}a \wedge \mathsf{GF}b \wedge \mathsf{GF}c$ is a single suspendable subformula which can be translated to a one-state deterministic TGBA.

`strong2.ltl` contains 100 formulae combined with a strong fairness hypothesis. Formulae have the form $\varphi_i \wedge (\mathsf{GF}a \rightarrow \mathsf{GF}b) \wedge (\mathsf{GF}c \rightarrow \mathsf{GF}d)$ where φ_i are the same as in the previous set.

For each formula set, the table can be read vertically to see the incremental effect of improvements presented in Sec. 4 on translations "Cou99". The difference between lines 1 and 2 shows that our acceptance simplification improvement is rather small: situations such as the one depicted by Fig. 2 are rare. Applying simulations to move from line 2 to 3 shows a much greater improvement, both in term of states and determinism. Analogous conclusions can be made by comparing lines 5, 6, and 7 where the original degeneralization is additionally applied to get BAs. The effect of the new degeneralization (line 8) defined in Sec. 4.3 is very limited on `known.ltl` because most BAs come directly out of the WDBA minimization function. It is much clearer in the other two sets of formulae. Application of a final simulation on the BA (line 9) saves a few more states.

The table also includes a compositional suspension with all other improvements (line 10). Its results on `known.ltl` are not very relevant as only 18 formulae of this set contain at least some suspendable subformula. The results are more interesting on the other two sets. As suspendable subformula $\mathsf{GF}a \wedge \mathsf{GF}b \wedge \mathsf{GF}c$ of each formula in `weak3.ltl` translates only to a one-state TGBA, one cannot expect improvements in automata size. The improvement here comes from the fact that the one-state TGBA is deterministic and the compositional approach allows to apply WDBA minimization to skeletons (note that it cannot be applied to the full formulae as fairness breaks obligation property). In many cases, we get a deterministic skeleton and composition with a deterministic TGBA results into a deterministic TGBA. To sum up, compositional suspension used in "deterministic" scenario produces substantially more deterministic automata (both TGBAs and BAs) than any other translation. The situation regarding automata size is different for `strong2.ltl` as $(\mathsf{GF}a \rightarrow \mathsf{GF}b) \wedge (\mathsf{GF}c \rightarrow \mathsf{GF}d)$ is a suspendable formulae that translates into a nondeterministic TGBA with 5

states. As the suspended TGBA is relatively big, compositional suspension brings a nice reduction of automata size and also an interesting speedup (again, for both TGBAs and BAs).

The table finally presents the results of `ltl3ba` [1] on `strong2.ltl`. `ltl3ba` improves `ltl2ba` [11] in several ways including the original suspension technique (see Sec. 3.1) and application of direct simulation on the final BA (but not before). We run `ltl3ba` with options `-S -A` to enable the direct simulation and disable the suspension and with option `-S` to enable both. Moreover, in the "deterministic" scenario we add the option `-M` leading to more deterministic automata. The lines 11 and 12 illustrate the gain that could be expected from the on-the-fly suspension [1] implemented in `ltl3ba`. It can be compared to the gain of compositional suspension from Sec. 3: the reduction between lines 11 and 12 should be compared to the reduction between lines 9 and 10.

6 Conclusion

We have presented four techniques to improve LTL-to-Büchi translators.

The compositional suspension improves the translations of *suspendable* subformulae (such as fairness constraints) and is especially effective in the case where the suspendable subformulae are expressed with automata of more than one state: in that case we avoid synchronizing the suspendable subformulae in non-accepting SCCs of the resulting automaton. The technique can accommodate any translator, by replacing the suspendable subformulae by fresh atomic propositions.

The other three contributions are improvements to the post-processings performed on the translated automaton. The SCC-based acceptance simplifications is an improvement over the transitional acceptance simplifications used in GBA. Its effect is limited as the forms of automata it attempts to simplify are not frequent in our benchmarks. Our simulation-based reductions build upon the existing *direct* and *reverse* simulations, but have been adapted to generalized acceptance sets, and implemented in a way that can be used to improve the determinism of the reduced automaton. Finally, we have shown that the degeneralization procedure could also benefit from the knowledge of the accepting SCCs.

In our experiments, we managed to reduce automata by a few states even on set of simple formulae (`known.ltl`) where years of developments have left only a little room for improvement. The bigger reduction were clearly achieved on formulae using strong fairness hypotheses (`strong2.ltl`).

Along the way, we pointed a couple of opportunities for further improvements. For instance in the degeneralization, and as far as we know, nobody has ever studied the selection of a suitable ordering (maybe SCC-based), or the selection of the best initial level. Our simulation currently suffers from the fact that Fig. 4(a) cannot be reduced to Fig. 4(c). Since suspendable subformulae are best translated separately, maybe we could consider other class of subformulae to translate separately (e.g., obligation properties are appealing since we already know how to construct a minimal WDBA from them).

Acknowledgments. T. Babiak, M. Křetínský, and J. Strejček have been supported by The Czech Science Foundation, grant No. P202/12/G061.

References

1. T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS'12*, volume 7214 of *LNCS*, pages 95–109. Springer, 2012.
2. J.-M. Couvreur. On-the-fly verification of temporal logic. In *FM'99*, volume 1708 of *LNCS*, pages 253–271. Springer, 1999.
3. M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *CAV'99*, volume 1633 of *LNCS*, pages 249–260. Springer, 1999.
4. C. Dax, J. Eisinger, and F. Klaedtke. Mechanizing the powerset construction for restricted classes of ω-automata. In *ATVA'07*, volume 4762 of *LNCS*, pages 223–236. Springer, 2007.
5. A. Duret-Lutz. LTL translation improvements in Spot. In *VECoS'11*, Electronic Workshops in Computing. British Computer Society, 2011.
6. A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. In *MASCOTS'04*, pages 76–83. IEEE, 2004.
7. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *FMSP'98*, pages 7–15, New York, Mar. 1998. ACM Press.
8. K. Etessami and G. J. Holzmann. Optimizing Büchi Automata. In *CONCUR'00*, volume 1877 of *LNCS*, pages 153–167. Springer, 2000.
9. K. Etessami, T. Wilke, and R. A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In *ICALP'01*, volume 2076 of *LNCS*, pages 694–707. Springer, 2001.
10. C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In *CIAA'03*, volume 2759 of *LNCS*, pages 35–48. Springer, 2003.
11. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *CAV'01*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.
12. D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE'02*, volume 2529 of *LNCS*, pages 308–326. Springer, 2002.
13. S. Juvekar and N. Piterman. Minimizing generalized Büchi automata. In *CAV'06*, volume 4144 of *LNCS*, pages 45–58. Springer, 2006.
14. Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *PODC'90*, pages 377–410. ACM press, 1990.
15. R. Sebastiani and S. Tonetta. "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking. In *CHARME'03*, volume 2860 of *LNCS*, pages 126–140. Springer, 2003.
16. F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *CAV'00*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.
17. R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. Sigref — a symbolic bisimulation tool box. In *ATVA'06*, volume 4218 of *LNCS*, pages 477–492. Springer, 2006.
18. P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *FOCS'83*, pages 185–194. IEEE, 1983.