LTL Translation Improvements in Spot 1.0

Alexandre Duret-Lutz

EPITA's Research and Development Laboratory (LRDE)
14-16 rue Voltaire, 94270 Le Kremlin-Bicêtre, France
E-mail: adl@lrde.epita.fr

Abstract: Spot is a library of model-checking algorithms started in 2003. This paper focuses on its module for translating linear-time temporal logic (LTL) formulas into Büchi automata: one of the steps required in the automata-theoretic approach to LTL model-checking.

We detail the different algorithms involved in this translation: the core translation itself, which performs many simplifications thanks to its use of binary decision diagrams; the pre-processing of the LTL formulas with rewriting rules chosen to help their translation; and various post-processing algorithms whose use depends on the intent of the translation: do we favor deterministic automata, or small automata?

Using different benchmarks, we show how Spot competes with other LTL translators, and how it has improved over the years.

Keywords: formal methods, model checking, Büchi automata, LTL, temporal logic, translation, simplifications, implementation, software, verification

Reference to this paper should be made as follows: A. Duret-Lutz. (2014) 'LTL Translation Improvements in Spot 1.0', *Int. J. Critical Computer-Based Systems*, Vol. 5, Nos. 1/2, pp. 31–54.

1 Introduction

One of the first steps of the automata-theoretic approach to model checking of linear-time properties (Vardi 1996, 2007) is to translate the property to verify into an ω-automaton. This automaton is then synchronized with a model of the system in order to find executions that invalidate the property. By constructing a smaller or more deterministic property automaton, we can hope (this is generally the case) to obtain a smaller synchronized product to explore, resulting in faster model checking.

The Spot library (Duret-Lutz & Poitrenaud 2004) offers algorithms to realize the above automata-theoretic approach. A salient feature of Spot is its preference for using Transition-based Generalized Büchi Automata (TGBA) instead of the more commonly used Büchi Automata (BA). Section 2 explains the difference.

This paper attempts to give a global view of the different algorithms involved into the LTL-to-TGBA or LTL-to-BA translation of Spot, to explain why it often produces smaller automata than other available translators, and why it does not always produce them as fast. Along the way, we point some steps (like the degeneralization) that could probably be improved. We believe the insight we provide into the implementation of Spot should be helpful to anyone devising a new translator.

Spot actually offers four translation procedures, and we shall only discuss the most efficient one, derived from an algorithm by Couvreur (1999).

A previous version of this paper was presented at VECOS'11 (Duret-Lutz 2011). The text has been augmented to discuss new optimizations implemented between Spot 0.7 and Spot 1.0, and presents new benchmarks featuring more LTL translators.

We assume the reader is familiar with LTL (Clarke et al. 2000) and Binary Decision Diagrams (Bryant 1986), abbreviated as BDDs in the sequel.

This paper is organized as follows. Section 2 defines Transition-based Generalized Büchi Automata as opposed to Büchi Automata. Section 3 presents the core of the translation algorithm, with an emphasis on the optimizations that are enabled by the use of BDDs, and discusses some improvements to this translation. In sections 4 and 5 we discuss pre-processing and post-processings. Finally, Section 6 compares Spot with other translators on various benchmarks.

Throughout the paper, the reader is invited to play with an on-line version of the translator at `http://spot.lip6.fr/ltl2tgba.html`. This page has options for many optimizations discussed herein.

2 Two kinds of Büchi automata

Let *AP* be a set of *atomic propositions*, i.e., propositional variables that may be true or false in the system. 2^{AP} denotes the set of minterms (or assignments) over *AP*, and $2^{2^{AP}}$, interpreted as the set of sums of minterms, denotes the Boolean formulas over *AP*.

Definition 1 *A Büchi automaton is a tuple* $\mathcal{B} = \langle AP, \mathcal{Q}, q^0, \mathcal{F}, \delta\rangle$ *where AP is a set of* atomic propositions*,* $\mathcal{Q}$ *is a finite set of states,* $q^0 \in \mathcal{Q}$ *is the initial state,* $\mathcal{F} \subseteq \mathcal{Q}$ *is a set of acceptance states, and* $\delta \subseteq \mathcal{Q} \times 2^{AP} \times \mathcal{Q}$ *is a transition relation in which each transition is labeled by a Boolean assignment.*
An infinite word $c_0c_1c_2\ldots \in (2^{AP})^\omega$ *of assignments is* accepted *by* $\mathcal{B}$ *if there exists a run of* $\mathcal{A}$*, say* $(q^0, l_0, q_1)(q_1, l_1, q_2)(q_2, l_2, q_3)\ldots \in \delta^\omega$*, that recognizes the word* $(\forall i, c_i = l_i)$ *and that visits infinitely many acceptance states* $(\forall i \geq 0, \exists j \geq i, q_j \in \mathcal{F})$*.*

A common implementation technique is to group transitions with common source and destination into edges labeled by Boolean formulas. E.g., the three *transitions* $(q_1, a\bar{b}, q_2)$, (q_1, ab, q_2), and $(q_1, \bar{a}b, q_2)$ can be represented by one *edge* $(q_1, a \vee b, q_2)$.

A Transition-based Generalized Büchi Automaton (TGBA) is a Büchi automaton in which multiple acceptance marks are carried by the transitions.

Definition 2 *A TGBA is a tuple* $\mathcal{T} = \langle AP, \mathcal{Q}, q^0, \mathcal{F}, \delta\rangle$ *where AP is a set of* atomic propositions*,* $\mathcal{Q}$ *is a finite set of states,* $q^0 \in \mathcal{Q}$ *is the initial state,* $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ *is a finite set of* acceptance marks*,* $\delta \subseteq \mathcal{Q} \times 2^{AP} \times 2^{\mathcal{F}} \times \mathcal{Q}$ *is a transition relation in which each transition is labeled by a Boolean assignment and a set of acceptance marks.*
An infinite word $c_0c_1c_2\ldots \in (2^{AP})^\omega$ *of assignments is* accepted *by* $\mathcal{T}$ *if there exists a run of* $\mathcal{A}$*, say* $(q^0, l_0, F_0, q_1)(q_1, l_1, F_1, q_2)(q_2, l_2, F_2, q_3)\ldots \in \delta^\omega$*, that recognizes the word* $(\forall i, c_i = l_i)$ *and that visits each acceptance mark infinitely often* $(\forall f \in \mathcal{F}, \forall i \geq 0, \exists j \geq i, f \in F_j)$*.*

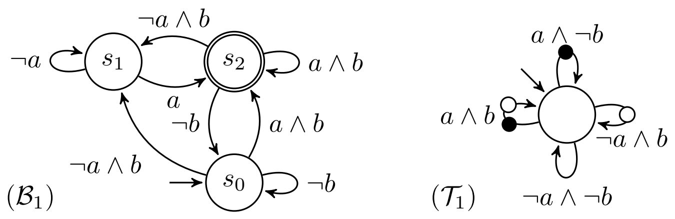


Figure 1 Two automata recognizing the LTL formula $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$. $\mathcal{B}_1$: Büchi automaton with a single acceptance state (double circle). $\mathcal{T}_1$: TGBA with $\mathcal{F} = \{\bigcirc, \bullet\}$.

Similarly, transitions that share the same source, destination and acceptance mark may be implemented by a single edge labeled by a Boolean formula. For simplicity, we only display these edges on the figures.

Figure 1 illustrates these definitions with two automata that recognize the LTL property: $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$. The infinite sequence $\begin{smallmatrix} a\!: & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \dots \\ b\!: & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \dots \end{smallmatrix}$ will be accepted by $\mathcal{T}_1$ because it visits the top and right loops infinitely often, therefore all acceptance marks are seen infinitely often. Similarly this sequence visits the only acceptance state of $\mathcal{B}_1$ infinitely often.

Spot is built around TGBAs and can perform the entire model-checking approach with these automata. However most other model-checking tools use Büchi automata. Fortunately, TGBAs can be *degeneralized* into Büchi automata by an operation discussed in Section 5.4. Automaton $\mathcal{B}_1$ in Fig. 1 was obtained by degeneralizing $\mathcal{T}_1$.

We will often name the states of automata with the LTL formula they accept. These extra annotations have no influence on the behavior of the automata.

In a Büchi automaton, we say that a strongly connected component (SCC) is accepting if it contains some accepting state. In a TGBA an SCC is accepting if for each acceptance mark it contains at least one marked transition.

3 From LTL to TGBA

The algorithm of Couvreur (1999) for the translation of LTL automata into TGBA is based on a tableau method. Although the following explanations are self-contained, we refer the reader to Duret-Lutz & Poitrenaud (2004) for an illustration of this algorithm as a tableau that can be used to build generalized Büchi automata with state-based or transition-based acceptance conditions. Here we shall present the algorithm at a lower level to explain how the use of BDDs helps the translation.

To put this algorithm in context, the complete translation procedure to go from LTL to a Büchi Automaton can be presented as four steps:

1. Simplify the LTL formula syntactically. E.g., rewrite $\mathsf{F}\,\mathsf{F}\,a$ (a 3-state automaton) into $\mathsf{F}\,a$ (2 states). These pre-processings are discussed in Section 4.
2. Translate the simplified formula into a TGBA using the algorithm presented in this section.
3. Post-process the resulting TGBA, e.g., by pruning useless SCCs, or running various simulation-based reductions or minimizations discussed in section 5.
4. If desired (and needed after the previous post-processing) degeneralize the TGBA into a Büchi automaton, as discussed in section 5.4.

$$\begin{array}{rl}
 & r(\mathsf{X}\,f) = \mathrm{Nxt}[f] \\
r(\top) = \top & r(\mathsf{F}\,f) = r(f) \lor (\mathrm{Nxt}[\mathsf{F}\,f] \land \mathrm{P}[f]) \\
r(\bot) = \bot & r(\mathsf{G}\,f) = r(f) \land \mathrm{Nxt}[\mathsf{G}\,f] \\
r(p) = \mathrm{Var}[p] & r(f\,\mathsf{U}\,g) = r(g) \lor (r(f) \land \mathrm{Nxt}[f\,\mathsf{U}\,g] \land \mathrm{P}[g]) \\
r(\neg p) = \neg\mathrm{Var}[p] & r(f\,\mathsf{W}\,g) = r(g) \lor (r(f) \land \mathrm{Nxt}[f\,\mathsf{W}\,g]) \\
r(f \lor g) = r(f) \lor r(g) & r(f\,\mathsf{R}\,g) = r(g) \land (r(f) \lor \mathrm{Nxt}[f\,\mathsf{R}\,g]) \\
r(f \land g) = r(f) \land r(g) & r(f\,\mathsf{M}\,g) = r(g) \land (r(f) \lor (\mathrm{Nxt}[f\,\mathsf{M}\,g] \land \mathrm{P}[f]))
\end{array}$$

Figure 2 Recursive rules to translate an LTL formula into a BDD.

3.1 Basic translation

If we omit BDDs, the procedure is simple enough to be performed by hand on a paper or blackboard. The algorithm generates an automaton whose states corresponds to LTL formulas. The initial state is the formula to translate. This formula is then rewritten as a sum of products where the only temporal operator allowed at the top level is X.

For instance if we were to translate $\Psi = (\mathsf{X}\,a) \land (b\,\mathsf{U}\,\neg a)$ we would use the fact that $\varphi\,\mathsf{U}\,\psi = \psi \lor (\varphi \land \mathsf{X}(\varphi\,\mathsf{U}\,\psi))$ to rewrite Ψ as $(\neg a \land \mathsf{X}\,a) \lor (b \land \mathsf{X}\,a \land \mathsf{X}(b\,\mathsf{U}\,\neg a))$. Reading this formula, it is clear that a state that must recognize Ψ should either accept an assignment compatible with $\neg a$ and verify a at the next step, or accept an assignment compatible with b and then verify $a \land (b\,\mathsf{U}\,\neg a)$ at the next step. The start of the automaton is thus as follows:

$$\begin{tikzcd}
 & & \boxed{a} \\
{} \arrow[r] & \boxed{(\mathsf{X}\,a) \land (b\,\mathsf{U}\,\neg a)} \arrow[ur, "\neg a"] \arrow[dr, "b"'] & \\
 & & \boxed{a \land (b\,\mathsf{U}\,\neg a)}
\end{tikzcd}$$

The procedure should then be applied similarly on the new states. There is little technicality that has to be taken into account when translating the $\varphi\,\mathsf{U}\,\psi$ operator: the formula ψ *must* be satisfied eventually, it cannot be postponed continuously. This is solved in the translation by making a *promise to fulfill* ψ while rewriting the formula. The actual rewriting rule used for U is: $\varphi\,\mathsf{U}\,\psi = \psi \lor (\varphi \land \mathsf{X}(\varphi\,\mathsf{U}\,\psi) \land \mathsf{P}\,\psi)$, with the operator P denoting an explicit promise.

All these formulas can be simplified using classical Boolean rules like $(\alpha \land \beta) \lor \alpha = \alpha$ to kill some terms (even $\mathsf{X}\,\varphi$ or $\mathsf{P}\,\varphi$). This is where using BDD really helps. The core of the translation is the rewriting function $r(f)$ defined recursively as in Fig. 2. It encodes outgoing transitions using BDD variables of the form $\mathrm{Var}[p]$, $\mathrm{Nxt}[f]$, $\mathrm{P}[f]$, created as needed to represent respectively atomic propositions, $\mathsf{X}\,f$ formulas, and $\mathsf{P}\,f$ promises. The given definition assumes that the LTL formula is specified into *negative normal form*, where negation operators appear only in front of atomic propositions.

Applying r on our example, we obtain:

$$\begin{aligned}
r((\mathsf{X}\,a) \land (b\,\mathsf{U}\,\neg a)) &= r(\mathsf{X}\,a) \land r(b\,\mathsf{U}\,\neg a) \\
&= \mathrm{Nxt}[a] \land (r(\neg a) \lor (r(b) \land \mathrm{Nxt}[b\,\mathsf{U}\,\neg a] \land \mathrm{P}[\neg a])) \\
&= \mathrm{Nxt}[a] \land (\neg\mathrm{Var}[a] \lor (\mathrm{Var}[b] \land \mathrm{Nxt}[b\,\mathsf{U}\,\neg a] \land \mathrm{P}[\neg a])) \\
&= (\neg\mathrm{Var}[a] \land \mathrm{Nxt}[a]) \lor (\mathrm{Var}[b] \land \mathrm{Nxt}[a] \land \mathrm{Nxt}[b\,\mathsf{U}\,\neg a] \land \mathrm{P}[\neg a])
\end{aligned}$$

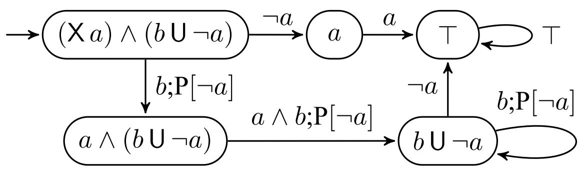
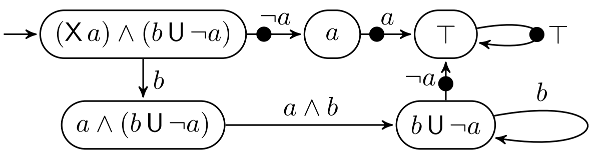


Figure 3 Translation of $(\mathsf{X}\,a) \wedge (b\,\mathsf{U}\,\neg a)$ using promises.



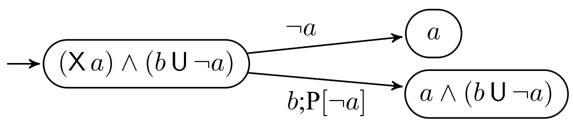
$(\mathsf{X}\,a) \wedge (b\,\mathsf{U}\,\neg a)$
$\neg a$
a
a
$\top$
$\top$
b
$\neg a$
b
$a \wedge b$
$a \wedge (b\,\mathsf{U}\,\neg a)$
$b\,\mathsf{U}\,\neg a$

Figure 4 Translation of $(\mathsf{X}\,a) \wedge (b\,\mathsf{U}\,\neg a)$ as a TGBA.

```
ltl_to_tgba_fm($f$):
  $todo \leftarrow \{f\}$; $all\_acc \leftarrow \emptyset$
  $a \leftarrow$ new automaton; $a$.set_initial_state($f$)
  while ($todo \neq \emptyset$)
    $here \leftarrow todo$.remove_one()
    forall $i$ in $prime\_implicants\_of(r(here))$
      Put $i$ as $\bigwedge_{v\in V} \mathrm{Var}[v] \wedge \bigwedge_{v\in V'} \neg\mathrm{Var}[v] \wedge \bigwedge_{a\in A} \mathrm{P}[a] \wedge \bigwedge_{n\in N} \mathrm{Nxt}[n]$
      $dest \leftarrow \bigwedge_{n\in N} n$
      if $\neg a$.has_state($dest$)
        $todo$.insert($dest$)
      $a$.add_edge(src: $here$, dst: $dest$, cond: $\bigwedge_{v\in V} v \wedge \bigwedge_{v\in V'} \neg v$, promises: $A$)
      $all\_acc \leftarrow all\_acc \cup A$
  forall $t$ in $a$.edges()
    $t$.acceptance_marks $\leftarrow all\_acc \setminus t$.promises
  return $a$
```

Figure 5 Pseudo-code of the algorithm of Couvreur (1999) to translate an LTL formula f into a TGBA. The function $r(here)$ is defined on Fig. 2.

Which corresponds to:



$(\mathsf{X}\,a) \wedge (b\,\mathsf{U}\,\neg a)$
$\neg a$
a
b;P$[\neg a]$
$a \wedge (b\,\mathsf{U}\,\neg a)$

There are several ways to turn a BDD into a sum of products, but because each term of the sum corresponds to a transition in the automaton, redundant terms should be avoided. Furthermore, Nxt[] and P[] variables should never be negated. We compute prime implicants using an algorithm from Minato (1992) to that effect.

The complete translation is shown on Fig. 3. This automaton is still not a TGBA because it uses promises instead of acceptance marks. To guarantee that a promise holds, the accepted runs of the automaton should never make promises continuously: in other words for each promise $\mathsf{P}\,\varphi$, accepted runs should visit infinitely many transitions that do not make such a promise.[1] This can be encoded as a TGBA by labeling all transitions that do not make promise $\mathsf{P}\,\varphi$ by an acceptance mark associated to φ. There will be as many acceptance marks as promises. Fig. 4 shows the final TGBA.

The pseudo-code for the complete translation algorithm is shown on Fig. 5.

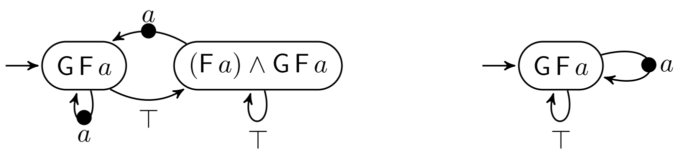


Figure 6 Two translations of $\mathsf{G\,F}\,a$. Since $r(\mathsf{G\,F}\,a) = r((\mathsf{F}\,a) \wedge \mathsf{G\,F}\,a)$ the two states of the first automaton can be merged, yielding the second automaton.

At this point it should be clear that the use of BDDs simplifies every Boolean formulas that label edges. For instance we cannot have an edge labeled by $b \wedge a \wedge \neg b$ because such a conjunction would be simplified by the BDD representation.

Similarly the conversion of the BDD into a sum of prime implicants helps to reduce the number of outgoing arcs of each node.

We experimented with different BDD variable orders, and found it was better to introduce variables in the order they are discovered while applying r recursively.

3.2 Using r to identify states

A powerful BDD-based optimization is to use r to identify some equivalent formulas. Because BDDs have a unique representation, two formulas φ and ψ are equivalent if their rewritings are the same BDDs $r(\varphi) = r(\psi)$. The converse does not hold because two equivalent subformulas prefixed with X might be represented by different Nxt[] variables. Since $r(\varphi)$ encodes the outgoing edges (labels, promises, and destinations) of the state ψ, if $r(\varphi) = r(\psi)$ then the states φ and ψ have exactly the same successors and can be merged. Such a reduction occurs when translating $\mathsf{G\,F}\,a$:

$$r(\mathsf{G\,F}\,a) = ((\mathrm{Nxt}[\mathsf{F}\,a] \wedge \mathrm{P}[a]) \vee \mathrm{Var}[a]) \wedge \mathrm{Nxt}[\mathsf{G\,F}\,a]$$

$$r((\mathsf{F}\,a) \wedge \mathsf{G\,F}\,a) = ((\mathrm{Nxt}[\mathsf{F}\,a] \wedge \mathrm{P}[a]) \vee \mathrm{Var}[a]) \wedge \mathrm{Nxt}[\mathsf{G\,F}\,a]$$

The result of $r(\mathsf{G\,F}\,a)$ implies that $\mathsf{G\,F}\,a$ should have two successors, $\mathsf{G\,F}\,a$ and $(\mathsf{F}\,a) \wedge \mathsf{G\,F}\,a$, as shown in the first automaton of Fig. 6. However $r((\mathsf{F}\,a) \wedge \mathsf{G\,F}\,a) = r(\mathsf{G\,F}\,a)$ so these states can be merged.

One way to implement this "r-quotienting" automatically is to index the states of the automaton by the BDD $r(\varphi)$ instead of by the LTL formula φ (the pseudo-code from Fig. 5 does *not* perform this reduction).

This automatic simplification may fail to merge states that have the same successors except for a self-loop because the Nxt[] variable representing the destination of the self-loop will be different in each state. Babiak et al. (2012) have suggested to improve this case by introducing a unique dummy BDD variable to represent the current state. This optimization is implemented in their LTL translator, `ltl3ba`, but not yet in Spot.

3.3 Better determinism

The determinism of the automata from Fig. 6 can be improved using a trick based on the BDD representation of states. Instead of converting the equation $r(\mathsf{G\,F}\,a) = ((\mathrm{Nxt}[\mathsf{F}\,a] \wedge \mathrm{P}[a]) \vee \mathrm{Var}[a]) \wedge \mathrm{Nxt}[\mathsf{G\,F}\,a]$ into a sum of products to discover the labels and destinations, we can instead fix one label to discover its destination(s).

Where shall we go if we read a? $\quad r(\mathsf{G\,F}\,a) \wedge \mathrm{Var}[a] = \mathrm{Var}[a] \wedge \mathrm{Nxt}[\mathsf{G\,F}\,a]$.
If we read $\neg a$? $\quad r(\mathsf{G\,F}\,a) \wedge \neg\mathrm{Var}[a] = \neg\mathrm{Var}[a] \wedge \mathrm{Nxt}[\mathsf{F}\,a] \wedge \mathrm{P}[a] \wedge \mathrm{Nxt}[\mathsf{G\,F}\,a]$.

$$\begin{array}{rlrl} r_{\mathsf{G}}(f \wedge g) &= r_{\mathsf{G}}(f) \wedge r_{\mathsf{G}}(g) & r_{\mathsf{G}}(f \mathbin{\mathsf{W}} g) &= r(g) \vee r(f) \\ r_{\mathsf{G}}(\mathsf{F}\, f) &= r(f) \vee \mathrm{P}[f] & r_{\mathsf{G}}(f \mathbin{\mathsf{R}} g)) &= r_{\mathsf{G}}(g) \\ r_{\mathsf{G}}(f \mathbin{\mathsf{U}} g) &= r(g) \vee (r(f) \wedge \mathrm{P}[g]) & r_{\mathsf{G}}(f \mathbin{\mathsf{M}} g)) &= r_{\mathsf{G}}(g) \wedge (r(f) \vee \mathrm{P}[f]) \\ r_{\mathsf{G}}(f) &= r(f) \text{ in all other cases} & & \end{array}$$

Figure 7 Recursive rules to translate LTL subformulas of G.

These equations show that all instances of $\top$ in Fig. 6 can be replaced by $\neg a$, yielding two deterministic automata.

In an automaton over n atomic propositions (Var$[a]$, Var$[b]$,...), there are 2^n labels to consider. However the structure of the BDD encoding the formula helps to ignore useless labels; and in real-world formulas, n is usually small enough to make the enumeration of these labels not perceptible.

While an automaton constructed this way is usually *more* deterministic, it is not necessarily a deterministic automaton. The result of $r(\varphi) \wedge A$ for some A could feature a disjunction, i.e., multiple destinations. (The reader is invited to compute $r(\mathsf{F}\,\mathsf{G}\,a) \wedge$ Var$[a]$ for an example.)

In an experiment we translated 92 LTL formulas taken from the literature and compared their translations with and without this optimization, by synchronizing the resulting automata with random state spaces. This technique reduced the number of transitions in the product by 40%, and the number of states by only 0.33%.

3.4 Speeding up the translation of G *formulas*

In his original paper, Couvreur (1999) discussed an optimization of this translation using a specific rule for formulas of the form $\mathsf{G}\,\mathsf{F}\,f$: $r(\mathsf{G}\,\mathsf{F}\,f) = (r(f) \vee \mathrm{P}[f]) \wedge \mathrm{Nxt}[\mathsf{G}\,\mathsf{F}\,f]$.

This rule avoids the creation of the state $\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,a$ during the translation of $\mathsf{G}\,\mathsf{F}\,a$. From a size perspective, it is entirely optional since the r-quotienting discussed in section 3.2 will already identify the two states. However from a time point of view, it is more efficient to construct a single state directly, and avoid many BDD operations. (Spot's translator spends more than half of its run time performing BDD operations.)

We generalized this rule to apply to any subformula that is guaranteed to be repeated in the next state. We modify the G rule of Fig. 2 as: $r(\mathsf{G}\,f) = r_{\mathsf{G}}(f) \wedge \mathrm{Nxt}[\mathsf{G}\,f]$ where r_{G} is the recursive function defined by Fig. 7. These r_{G} rules, called inside $r(\mathsf{G}\,f)$, avoid the creation of the Nxt$[f]$ variables that would be implied by Nxt$[\mathsf{G}\,f]$ anyway. In particular, this optimization halves the time spent translating subformulas of the form $\bigwedge_i \mathsf{G}\,\mathsf{F}\,p_i$ or of the equivalent (but preferred) form $\mathsf{G} \bigwedge_i \mathsf{F}\,p_i$, either of which occur when expressing weak fairness properties.

3.5 Simplifying promises

Consider the BDD rewriting of $a \mathbin{\mathsf{U}} (b \mathbin{\mathsf{U}} c)$ whose complete automaton is represented with promises on Fig. 8:
$r(a \mathbin{\mathsf{U}} (b \mathbin{\mathsf{U}} c)) = \mathrm{Var}[c] \vee (\mathrm{Var}[b] \wedge \mathrm{Nxt}[b \mathbin{\mathsf{U}} c] \wedge \mathrm{P}[c]) \vee (\mathrm{Var}[a] \wedge \mathrm{Nxt}[a \mathbin{\mathsf{U}} (b \mathbin{\mathsf{U}} c)] \wedge \mathrm{P}[b \mathbin{\mathsf{U}} c])$

This BDD encodes three transitions, two of which use different promises: P$[c]$ and P$[b \mathbin{\mathsf{U}} c]$. However these promises are always issued sequentially: first P$[b \mathbin{\mathsf{U}} c]$ forbids

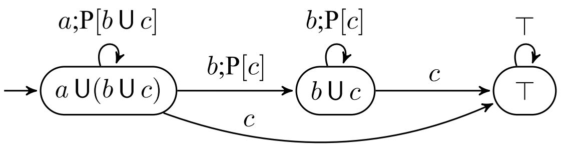


Figure 8 Translation of $a \mathsf{U} (b \mathsf{U} c)$ using promises (and without using the determinization improvement of Section 3.3).

runs that continuously stay in the initial state, then if the state $b \mathsf{U} c$ is reached, P$[c]$ rejects runs that would stay infinitely in that state. In practice, we could have made the same promise, for instance P$[c]$ (the name does not even matter), on all these transitions. If we interpret P$[f]$ as a promise to fulfill f eventually, it is clear that P$[b \mathsf{U} c]$ and P$[c]$ are two equivalent promises.

Along these lines, we implement the following simplifications to limit the number of promises introduced: P$[\mathsf{F}\, f]$ = P$[f]$, P$[f \mathsf{U} g]$ = P$[g]$, and P$[f \mathsf{M} g]$ = P$[f]$.

Furthermore, if the top-level formula is a syntactic persistence, only one promise need to be used during the translation and we rewrite any P$[f]$ as P$[\top]$. This optimization and the class of syntactic persistence formulas are described by Černá & Pelánek (2003).

4 Pre-processings

Pre-processing the LTL formula before it is translated helps to speed-up the translation, and to produce smaller automata. Spot distinguishes different kinds of LTL rewritings:

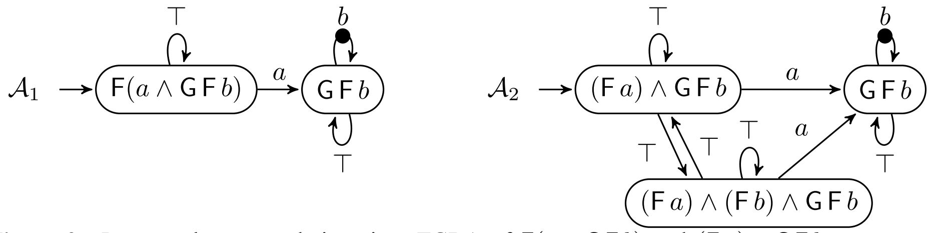
Trivial identities are applied at any time during the construction of a formula (e.g., while they are parsed). These are all based on idempotence of some operators (e.g., $\mathsf{F}\,\mathsf{F}\, a \equiv \mathsf{F}\, a$), or neutral/absorbent operands (e.g., $\mathsf{X} \bot \equiv \bot$, $f \wedge \bot \equiv \bot$, $f \wedge \bot \equiv \bot$, etc.).

Basic rewritings Are unconditional rewriting rules, such as $\mathsf{G}\,\mathsf{X}\, f \equiv \mathsf{X}\,\mathsf{G}\, f$.

Eventual and universal rewritings apply only when some subformulas are *purely universal* Etessami & Holzmann (2000), are *pure eventualities* Etessami & Holzmann (2000), or are what Babiak et al. (2012) have called *alternating* formulas. As an example $\mathsf{F}\,\mathsf{G}\,\mathsf{F}\, a$ can be rewritten as $\mathsf{G}\,\mathsf{F}\, a$ because the latter is a pure eventuality.

Implication-based rewritings apply only in cases where one subformula can be shown to imply another subformula. For instance under the hypothesis that $f \rightarrow g$, we have $f \mathsf{U} g \equiv g$. There are two ways to detect such implications: they can be approximated syntactically (Somenzi & Bloem 2000), or decided exactly using automata-based language containment checks (Tauriainen 2006).

Spot implements many (but not all) rewriting rules taken from the aforementioned sources, plus some of its own. A complete listing of all these rules is distributed along with Spot[2] and is too long to be reproduced here. We only discuss a couple of them to illustrate the point that these rewritings should be selected from the point of view of the translation algorithm that will be used next.



$\mathcal{A}_1$
$\mathsf{F}(a \land \mathsf{G}\,\mathsf{F}\,b)$
$\mathsf{G}\,\mathsf{F}\,b$
a
b
$\top$
$\top$
$\mathcal{A}_2$
$(\mathsf{F}\,a) \land \mathsf{G}\,\mathsf{F}\,b$
$\mathsf{G}\,\mathsf{F}\,b$
$(\mathsf{F}\,a) \land (\mathsf{F}\,b) \land \mathsf{G}\,\mathsf{F}\,b$
a
a
b
$\top$
$\top$
$\top$
$\top$
$\top$

Figure 9 Paper-and-pen translations into TGBA of $\mathsf{F}(a \land \mathsf{G}\,\mathsf{F}\,b)$ and $(\mathsf{F}\,a) \land \mathsf{G}\,\mathsf{F}\,b$.

4.1 A harmful rewriting rule

As a first example, we do not apply the rule $\mathsf{F}(\varphi \land \mathsf{G}\,\mathsf{F}\,\psi) \equiv (\mathsf{F}\,\varphi) \land (\mathsf{G}\,\mathsf{F}\,\psi)$ suggested by Somenzi & Bloem (2000). Intuitively, this rule is dubious because $\mathsf{F}(\varphi \land \mathsf{G}\,\mathsf{F}(\psi))$ appears less complex to translate. Indeed, translating $\mathsf{F}\,\Phi$ is just a matter of creating an initial state that accepts any letter for a finite number of step, and non-deterministically jumps into a state that will recognize Φ when a letter matching the beginning of Φ is found. However, translating a formula such as $(\mathsf{F}\,\varphi) \land \mathsf{G}\,\mathsf{F}\,\psi$ is harder because in the initial state you have four choices to consider: either the input can be the start of φ, or it is the start of ψ, or it is both, or it is none. When φ and ψ are atomic propositions as in Fig. 9, these four cases can be reduced to three. It turns out that on the automaton $\mathcal{A}_2$ from Fig. 9 the states $(\mathsf{F}\,a) \land \mathsf{G}\,\mathsf{F}\,b$ and $(\mathsf{F}\,a) \land (\mathsf{F}\,b) \land \mathsf{G}\,\mathsf{F}\,b$ have exactly the same outgoing transitions: they can be merged. Thanks to the BDD identification discussed in Sec. 3.2, Spot will actually output an automaton similar to $\mathcal{A}_1$ for both formulas $\mathsf{F}(a \land \mathsf{G}\,\mathsf{F}\,b)$ or $(\mathsf{F}\,a) \land \mathsf{G}\,\mathsf{F}\,b$. This is not the case when φ and ψ are more complex.

This rewriting rule, which we applied in the past, also prevented other useful rules to apply. E.g., Spot 0.5 would rewrite the formula $\mathsf{F}(\varphi_1 \land \mathsf{G}\,\mathsf{F}\,\psi_1) \lor \mathsf{F}(\varphi_2 \land \mathsf{G}\,\mathsf{F}\,\psi_2)$ as $((\mathsf{F}\,\varphi_1) \land \mathsf{G}\,\mathsf{F}\,\psi_1) \lor (\mathsf{F}\,\varphi_2) \land \mathsf{G}\,\mathsf{F}\,\psi_2)$ missing the opportunity to apply the rule $\mathsf{F}(\Psi_1) \lor \mathsf{F}(\Psi_2) = \mathsf{F}(\Psi_1 \lor \Psi_2)$. Since Spot 0.6, we rewrite this formula as $\mathsf{F}((\varphi_1 \land \mathsf{G}\,\mathsf{F}\,\psi_1) \lor (\varphi_2 \land \mathsf{G}\,\mathsf{F}\,\psi_2))$, which is easier to translate for similar reasons.

4.2 Handling the W *and* M *operators*

Figure 2 includes rules to translate the W (weak until) and M (strong release) LTL operators. Many tools dealing with LTL formulas do not implement these operators or treat them as syntactic sugar: they do not add expressive power and can be rewritten using other operators. To illustrate the importance of the rewriting rules from the point of view of the translator algorithm, we consider different rewritings for these operators.

The formula $a\,\mathsf{W}\,b$ is usually rewritten into $(a\,\mathsf{U}\,b) \lor \mathsf{G}\,a$. For instance this is the implementation of the W operator in Spin 6.2.2. From the point of view of an LTL translator based on a tableau method, this is not a very good rewriting as it requires a non-deterministic choice between $a\,\mathsf{U}\,b$ and $\mathsf{G}\,a$ at the very beginning. A better rewriting is $a\,\mathsf{W}\,b \equiv a\,\mathsf{U}(b \lor \mathsf{G}\,a)$, as it postpones the choice between b and $\mathsf{G}\,a$. This latter rewriting was used by Dwyer et al. (1998), although they now changed their web site[3] to use W for simplicity. An even better choice, although less intuitive, is $a\,\mathsf{W}\,b \equiv b\,\mathsf{R}(a \lor b)$, since no promise have to be introduced. The TGBAs corresponding to the translation of these different rewritings are shown on Fig. 10. Similar rewritings for M exist: $a\,\mathsf{M}\,b \equiv (a\,\mathsf{R}\,b) \land (\mathsf{F}\,a) \equiv a\,\mathsf{R}(b \land \mathsf{F}\,a) \equiv b\,\mathsf{U}(a \land b)$.

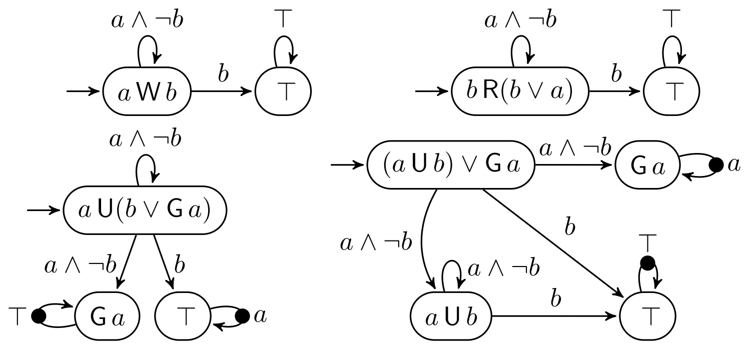


Figure 10 Four formulas equivalent to $a \mathsf{W} b$ and their corresponding automata.

Since Spot fully supports the W and M operators, our basic rewriting rules actually perform the reverse of all the previous rewritings (e.g., we rewrite into $a \mathsf{W} b$ the formulas $(a \mathsf{U} b) \lor \mathsf{G} a$, $a \mathsf{U}(b \lor \mathsf{G} a)$, and $b \mathsf{R}(a \lor b)$).

4.3 Implementation of LTL formulas

The implementation of all these rewriting rules benefit greatly from our representation of a set of LTL formulas as a forest of “syntax DAGs” with sharing of subformulas.

LTL formulas are reference counted and a unicity table makes sure that two equal formulas (or subformulas) will share the same address. The operators $\land$ and $\lor$ are handled as n-ary operators, and their operands are always sorted. We can therefore easily detect that $a \land \mathsf{X}(b) \land \mathsf{F}(c)$ is equivalent to $\mathsf{F}(c) \land \mathsf{X}(b) \land a$ because the two formula objects will have the same address.

The uniqueness of each subformula also helps to speed up rewriting algorithms, as they use a cache when processing subformulas recursively.

5 Post-processings

Once an LTL formula has been translated into a TGBA as described in Section 3, Spot implements different kinds of post-processings. We first describe each processing independently before explaining when there are used and how they are chained.

5.1 SCC pruning

It may happen that the TGBA constructed by the translation contains states that do not contribute to its language. A classical optimization is therefore to remove all non-accepting SCCs that cannot reach an accepting SCC (Somenzi & Bloem 2000).

Since we have to traverse the entire automaton to classify its SCCs as accepting or non-accepting, we can also perform a few other improvements along the way. The acceptance marks of transitions that do not belong to an accepting SCC can be removed. Similarly, acceptance marks that are always present at the same time as another acceptance mark can be simplified.

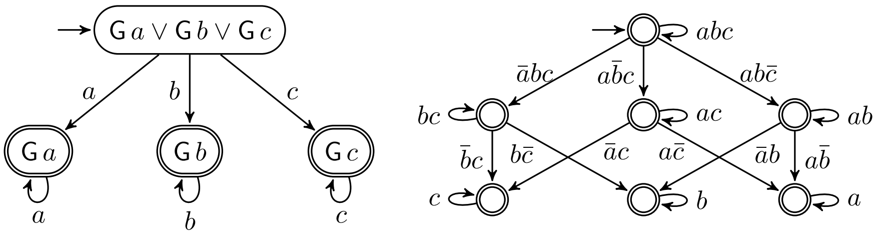


Figure 11 Left: translation of $\mathsf{G}\,a \vee \mathsf{G}\,b \vee \mathsf{G}\,c$. Right: its minimal WDBA.

5.2 Minimization of Weak Deterministic Büchi Automata

A Büchi automaton is weak if, in each SCC, either all the cycles are accepting, or all cycles are non-accepting.

It is well known that not all Büchi automata can be determinized (Vardi 1996, prop. 8). There is a subclass of properties that can be represented by Weak Deterministic Büchi Automata (WDBA), and for which there exists an algorithm to compute the minimal WDBA recognizing the property (Löding 2001). This class corresponds to the "*obligations*" in the temporal hierarchy of Manna & Pnueli (1990) and includes a large number of LTL formulas used for model checking.For instance 40 formulas out of the 55 formulas from Dwyer et al. (1998) are obligations.

Dax et al. (2007) showed how to implement this minimization without knowing *a priori* if the translated property actually is an obligation: the correctness of the minimization is tested *a posteriori* using a language equivalence test (easy to implement because a WDBA can be complemented like deterministic finite automata, and the original TGBA can be complemented by translating the negation of the property).

Dax et al. (2007) did a comparison of the size produced by different translators (not Spot, which they did not know) with the size of the minimal WDBA. This revealed that although it was deterministic, the minimal WDBA usually had a number states smaller or equal to that of the automata produced by the translators.

This WDBA minimization has since been integrated into Spot, and we completed the benchmark of Dax et al. (2007) in the previous version of this paper (Duret-Lutz 2011). Our implementation takes a TGBA, and outputs a deterministic Büchi automaton when the WDBA-minimization is valid. We avoid the language equivalence test in a number of cases by testing whether the translated formula actually belongs to the syntactic obligation class (Černá & Pelánek 2003).

While being able to output a minimal deterministic automaton for some class of LTL formulas is appreciable, we have found a few cases were using such a deterministic output was not desirable because the deterministic automaton was too large.

As an example consider the family of LTL formulas $\mathsf{G}\,p_1 \vee \mathsf{G}\,p_2 \vee \ldots \mathsf{G}\,p_n$. Figure 11 shows the result of the translation for $n = 3$ before and after WDBA minimization. The non-deterministic automaton has $n + 1$ states, while the minimal deterministic automaton has $2^n - 1$ states. Experiments on actual model checking problems show that the smaller of these two automata has to be preferred, despite its non-determinism, when the full product with the system must be constructed.

5.3 Simulation-based reductions

Spot implements the simulation-based reductions described by (Somenzi & Bloem 2000), which are easily adjusted to work on a TGBA. Intuitively direct simulation can merge states based on the inclusion of the sets of infinite runs *starting from* these states, while reverse simulation would merge states based on the inclusion between sets of finite runs *leading to* these states.

Our implementation has the same structure as the *StrongFairSimulation* algorithm of Etessami & Holzmann (2000), except that we represent the class (or color) of a state using BDD variables to ease inclusion checks. More details about our implementation are given by Babiak et al. (2013).

5.4 Degeneralization

A degeneralization algorithm takes a generalized automaton with n states and m acceptance marks, and produces a Büchi automaton with at most $n(m+1)$ states. The classical algorithm used to transform Generalized Büchi Automata into Büchi automata (Clarke et al. 2000, section 9.2.2) can be adapted to transform TGBA into Büchi automata (Giannakopoulou & Lerda 2002, Gastin & Oddoux 2001) as follows.

If $\mathcal{T} = \langle AP, \mathcal{Q}, q_0, \mathcal{F}, \delta\rangle$ is a TGBA with m acceptance marks $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$, then an equivalent Büchi automaton $\mathcal{T} = \langle AP, \mathcal{Q}', q'_0, \mathcal{F}', \delta'\rangle$ can be constructed as follows:

- $\mathcal{Q}' = \mathcal{Q} \times \{0, \ldots, m\}$, i.e., the original automaton is cloned in $m+1$ levels,
- $\mathcal{F}' = \mathcal{Q} \times \{m\}$, i.e., states from the last level are accepting,
- $\delta' = \{((s,j), l, (d, \mathit{level}_j(F))) \mid (s,l,F,d) \in \delta\}$
  where $\mathit{level}_j(F) = \begin{cases} 0 & \text{if } j = m \\ j+1 & \text{if } j < m \text{ and } f_{j+1} \in F, \\ j & \text{otherwise} \end{cases}$
  i.e., for each level $j < m$ the outgoing transitions that carry f_{j+1} are redirected to the next level and all transitions from the last level are redirected to level 0,
- $q'_0 = (q_0, 0)$, i.e., the initial state is on the first level (but any other level would also be correct).

This setup guarantees that any accepting path in the degeneralized automaton will correspond to an infinite path that sees all acceptance marks infinitely often in the original automaton. The classical optimization is to "jump levels", i.e., when a transition from level $i < m$ carries acceptance marks f_{i+1}, f_{i+2}, and f_{i+3}, it can be redirected to the level $i+3$. This corresponds to the following redefinition of $\mathit{level}_j(F)$:

$$\mathit{level}_j(F) = \begin{cases} \max\{n \in \{j, \ldots, m\} \mid \forall k \in \{j+1, \ldots, n\}, f_k \in F\} & \text{if } j < m, \\ \max\{n \in \{0, \ldots, m\} \mid \forall k \in \{1, \ldots, n\}, f_k \in F\} & \text{if } j = m. \end{cases}$$

The automaton $\mathcal{B}_1$ from Fig. 1 was degeneralized from $\mathcal{T}_1$ with this definition, in the order $f_1 = \circ$, $f_2 = \bullet$, and setting the initial state in the last level.

Another optimization this is implemented in Spot is a "*pulling of acceptance marks*". When all outgoing transitions of a state s have a set Y of acceptance marks in common, this set can be added to the acceptance marks of all the incoming transitions. This is correct because if a run traverses s it will necessarily see all acceptance marks from Y; it makes no difference if its sees them twice.

This degeneralization procedure offers $m!$ possible ways to order the acceptance marks, and there are $m + 1$ possible levels on which the initial state can be located. Changing these parameters might make some states from $Q \times \{0, \ldots, m\}$ unreachable, and can thus reduce the automaton. For one TGBA, we therefore have $m!(m + 1)$ possible degeneralizations using only this definition.

In Spot, the order of acceptance of sets used for the degeneralization correspond to the order in which the corresponding promises where introduced during the translation, and the initial state is always on the first level. There is definitely room for improvement here, since the initial submission of this paper, we have been working with the authors of `ltl3ba` to improve the situation (Babiak et al. 2013).

Oddoux (2003, section 6.1.2) mentions another kind of degeneralization in which the acceptance marks can be taken in any order and where each state of the degeneralized automaton has to retain the set of all acceptance marks that are waited for. This can potentially multiply the size of the original automaton with 2^m if m acceptance marks are used. But this might be worth a try when m is very small.

5.5 The complete post-processing chain

Because it is not always clear in which context the translated automaton will be used, Spot 1.0 introduces two different options to specify the intent of the translation.

`--deterministic` is used to indicate that an output that is (as much as possible) deterministic is desired. E.g., the right automaton of Fig. 11 should be preferred. In this case, we first prune useless SCC and acceptance marks in the translated TGBA, then we apply WDBA-minimization. If the latter succeeded, we output its result (a Büchi automaton) as-is. In case where WDBA-minimization was not applicable, we reduce the TGBA by iterating both direct and reverse simulation until the automaton is not reduced any more. The simulation-reduced TGBA is then degeneralized if requested.

`--small` is used to indicate that an output with less states should be favored. We shall still strive to make it deterministic, but if a choice like that of Fig. 11 happens, we will prefer the left automaton.
The post-processing for this intent also starts by pruning useless SCCs and acceptance marks. Then we compute two different automata, and return the smallest: the first automaton is the result of WDBA-minimization (if that result exists), and the second is the result of the iterated simulation (optionally degeneralized). If the two automata have an equal number of states, we keep the WDBA because it is guaranteed to be deterministic.

6 Benchmarks

The following sections present different benchmarks comparing Spot with other translators that are publicly available (including older versions of Spot).

These translators (presented in chronological order) are:

- The Spin model checker. Its `-f` option converts an LTL formula into a never claim representing a (degeneralized) Büchi automaton. Spin's LTL translator is based on the tableau construction of Gerth et al. (1996). Spin has some trivial and unconditional rewriting rules for LTL, and includes simple post-processings.
- LBT (Rönkkö 1999) also implements the translation of Gerth et al. (1996), but produces a generalized Büchi automaton. LBT only apply trivial rewriting rules. It has no post-processings.
- `wring` (Somenzi & Bloem 2000) implements some unconditional LTL rewritings, as well as some implication-based checks. Using a tableau construction it builds a Generalized Büchi Automaton with labels on states (rather than transitions). This GBA is simplified using SCC-based and simulation-based reductions.
- `ltl2ba` (Gastin & Oddoux 2001) is a descendant of Spin's translator in the sense that it reused the same code base. However the translation algorithm has been completely rewritten. LTL formulas are reduced using all classes of rewriting rules (the implication checks are syntactic), translated into an intermediate alternating Büchi automaton, which is then converted into a TGBA, which is finally degeneralized into a Büchi automaton. Some simplifications (like removing redundant transitions and useless SCCS) are performed at all these steps.
- `modella` (Sebastiani & Tonetta 2003) uses a tableau construction, implements all classes of rewriting rules (with syntactic implication checks), it also implements simulation-based reductions on the Büchi automaton. One of the main points of Modella's authors was that it is worth improving the determinism of the automaton at the expense of its size, because this will pay off when this automaton is later synchronized with a system to check.
- `ltl2nba` (Fritz 2003) translates LTL formulas into alternating Büchi automata with ε-transitions, and performs simulation reductions directly on that. These alternating automata are then converted into Büchi automaton using the Miyano-Hayashi construction. No pre- or post- processing are performed.
- `ltl3ba` (Babiak et al. 2012) is a reimplementation of `ltl2ba` in C++ with better data structures and additional optimizations. It implements many LTL rewriting rules, including some new ones based on a class for formulas called alternating formulas. For instance it uses BDDs to simplify the guards of transitions. It implements a technique called suspension that would effectively solve the problem described in 4.1: when `ltl3ba` translates $(\mathsf{F}\,\varphi) \wedge (\mathsf{G}\,\mathsf{F}\,\psi)$, it suspends the translation of $\mathsf{G}\,\mathsf{F}\,\psi$ until a point where φ must hold. This translator also implements a direct-simulation reduction on the final Büchi automaton (option `-S`), and has an option to improve determinism (option `-M`).

`ltl2nba` and `wring` are scripts written respectively in Python and Perl. All other tools are compiled from C or C++. All the following experiments were ran under GNU/Linux on an Intel Core2 Q9550 running at 2.83GHz with 8GB of RAM.

6.1 184 LTL formulas from the literature

The benchmark consists in 92 LTL formulas:

- 55 formulas from Dwyer et al. (1998) (where $a\,\mathsf{W}\,b$ was written as $a\,\mathsf{U}(b \vee \mathsf{G}\,a)$),
- 25 formulas from Somenzi & Bloem (2000) — their paper shows 27 formulas but two of them are already the negations of other formulas in the list,
- 12 formulas from Etessami & Holzmann (2000).

With their negations this makes a total of 184 formulas.

A summary of the translation of these formulas is presented in Table 1. The statistics displayed in this table were gathered using `ltlcross`, a Spot-based reimplementation of LBTT (Tauriainen & Heljanko 2002) that cross compares translators in order to detect errors (for our extensive test suite) and collect statistics (for our papers). The tools have been clustered by type of automaton produced, with Spot appearing in two groups depending on whether it was configured to output BA or TGBA.

As shown by the *count* column, Spin failed to translate 11 formulas within the 10 minutes limit we had set up (the machine was swapping before the end of these 10 minutes, meaning spin needed more than the available memory). Wring aborted in three cases with an error message from Perl. Modella produced one incorrect automaton.

Modella and `ltl2nba` output automata in a format in which states need not be declared accepting if they all are, this explains why they show less acceptance sets/marks than translated formulas (but this difference is not important). For other LTL-to-BA translators we used the never claim output.

The product with a random state space gives some idea of the behavior of the automaton during model checking. The intuition is that if this state space was that of a real model to verify, the model checking procedure would need space proportional to the number of state in the product, and time proportional to the number of transitions in that product. This can be used to argue for instance that although `modella`'s automata are bigger than `ltl2ba`'s, they will yield less transition in the product, and therefore make a faster verification. A similar effect can be seen with `ltl3ba`'s `-M` option: it improves the determinism at the expense of the number of states, but this pays off in terms of transitions in the products. This interpretation of the last columns should be mitigated by the fact that these random state spaces are not real models (Pelánek 2008), and that an actual model checker will implement other techniques to avoid constructing the entire product.

The statistics for Wring are slightly unfair because the nature of the automata it generates (state-based labels, and multiple initial states) is very different from the automata produced by other translators. In order to integrate Wring in our benchmark, we had to add a fake initial state (connected to all the former initial states) to each produced automata, and move the label of each state onto all its incoming transitions. This quick transformation adds one more state per automaton, and is also accounted for in the total run time.

On Table 1 Spot appears better than all other translator on all accounts except its run time (which is still reasonable). The number of states in its BA output is probably even more impressive if we additionally consider the number of nondeterministic automata: the automata are smaller *and* more deterministic.

However these cumulative values hide the actual differences that may be observed when comparing the results formula by formula. There is only one formula for which Spot produces an automaton with one more states than other tools. However if we distinguish automata with equal number of states by their number of transitions, then these cases are more numerous as shown in Table 2.

Detailed results can be found at `http://www.lrde.epita.fr/~adl/ijccbs/`, and include an interactive page that can build Table 2 for different comparison criteria.

			automata sizes					non-det.			products		
	translator	count	states	edges	trans.	acc.	SCC	states	aut.	time	states	trans.	SCC
BA	`spin` 6.2.2	173	2166	17428	62771	173	1242	1861	169	964.09	287823	9983437	29575
	`ltl2ba` 1.1	184	1017	3385	30237	184	732	811	173	0.67	194684	5638095	4390
	`ltl2nba`	184	952	3065	27158	181	724	744	174	18.77	179988	5472936	3470
	`modella` 1.5.9	183	1312	4391	23998	180	861	650	119	30.78	219548	4258427	14753
	`ltl3ba` 1.0.1 `-S`	184	812	2242	21339	184	627	607	172	1.00	154567	4350636	1769
	`ltl3ba` 1.0.1 `-M`	184	875	2634	15101	184	666	357	124	1.05	163669	3090518	4133
	`ltl3ba` 1.0.1 `-M -S`	184	848	2437	14495	184	648	330	124	1.07	158517	2953167	3260
	Spot 1.0 `--deterministic`	184	683	1707	10627	184	496	93	45	4.27	132239	2409181	1453
	Spot 1.0 `--small`	184	678	1683	10517	184	495	99	50	4.47	131850	2406921	1474
GBA	`lbt` 1.2.2	184	8415	130501	884155	333	4781	7588	180	1.62	1423870	108033745	596754
	Wring 1.1.0	181	1476	4943	49416	182	1120	1027	166	31.80	253354	7765883	74934
TGBA	Spot 1.0 `--deterministic`	184	641	1573	9964	198	493	84	45	4.41	124053	2258314	1446
	Spot 1.0 `--small`	184	636	1549	9854	198	492	90	50	4.56	123664	2256054	1467

Table 1 Cumulative summary for the translation of 184 formulas by each tool. With the exception of the *count* column, smaller values are better. The first data column displays the total *count* of formulas successfully translated by each tool. The remaining columns all display accumulated values over all successful translations. The first columns display the total number of *states*, *edges*, *transitions*, *acceptance marks* (1 for BA), and *strongly connected components* in all the translated automaton. They are followed by the total number of *non-deterministic states*, and the total number of *non-deterministic automata* in these translated automata. The next column gives number of seconds it took to translate all formulas.
For each formula, one random (and deadlock free) state space of 200 states was created, and used to build a synchronized product with each translated automaton. The size of this product is shown in the last three columns. We do not have to distinguish edges from transitions in the product because all atomic propositions are valued in the state space: each edge maps to one transition.

	spin	ltl2ba	ltl2nba	modella	ltl3ba -S	ltl3ba -M	ltl3ba -M -S	Spot (det.)	Spot (small)
spin		131	132	134	135	166	168	169	169
ltl2ba	2		38	69	47	160	162	156	159
ltl2nba	5	25		65	43	147	150	158	161
modella	35	104	108		116	116	116	132	132
ltl3ba -S	0	18	31	55		138	144	153	157
ltl3ba -M	0	3	18	4	22		19	82	84
ltl3ba -M -S	0	3	16	4	15	0		79	82
Spot (det.)	0	10	9	1	10	24	25		5
Spot (small)	0	7	6	1	6	20	20	0	

Table 2 Comparing LTL-to-BA translator on 182 formulas. The value on line i and column j shows how many times the automaton produced by translator $\#i$ was strictly bigger than the automaton produced by translator $\#j$. Here "bigger" means "more states" or "equal number of states and more transitions".

Spot 0.8.3	562 seconds
Spot 0.9	315 seconds
Spot 0.9.1	198 seconds
Spot 1.0	150 seconds
ltl3ba 1.0.1	77 seconds

Table 3 Total time required to translate α_n, β_n, β'_n, φ_n, and ψ_n for $1 \leq n \leq 20$.

6.2 Some formulas for which the minimal Büchi automaton is known

Cichoń et al. (2009) studied several classes of LTL formulas for which they calculated the size (in states) of the minimal Büchi automaton that could represent the property. They compared the output of Spot 0.4 and `ltl2ba` 1.1, neither of which was able to translate all formulas efficiently. Sometimes they would take too long (hours or days), sometimes they would produce automata larger than necessary.

Here are the five families of formulas they evaluated on both tools for n ranging from 1 to 20:

$$\alpha_n = \mathsf{F}(p_1 \wedge \mathsf{F}(p_2 \wedge \mathsf{F}(\ldots \mathsf{F}\, p_n))) \wedge \mathsf{F}(q_1 \wedge \mathsf{F}(q_2 \wedge \mathsf{F}(\ldots \mathsf{F}\, q_n)))$$

$$\beta_n = \mathsf{F}(\underbrace{p \wedge \mathsf{X}(p \wedge \mathsf{X}(p \wedge \ldots))}_{n \text{ occurrences of } p}) \wedge \mathsf{F}(\underbrace{q \wedge \mathsf{X}(q \wedge \mathsf{X}(q \wedge \ldots))}_{n \text{ occurrences of } q})$$

$$\beta'_n = \mathsf{F}(\underbrace{p \wedge \mathsf{X}(p) \wedge \mathsf{X}\,\mathsf{X}(p) \wedge \ldots}_{n \text{ occurrences of } p}) \wedge \mathsf{F}(\underbrace{q \wedge \mathsf{X}(q) \wedge \mathsf{X}\,\mathsf{X}(q) \wedge \ldots}_{n \text{ occurrences of } q})$$

$$\varphi_n = \mathsf{G}\,\mathsf{F}\,p_1 \wedge \mathsf{G}\,\mathsf{F}\,p_2 \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\,p_n$$

$$\psi_n = \mathsf{F}\,\mathsf{G}\,p_1 \vee \mathsf{F}\,\mathsf{G}\,p_2 \vee \ldots \vee \mathsf{F}\,\mathsf{G}\,p_n$$

Nowadays Spot, as well as the new `ltl3ba`, are both able to translate all 100 formulas into their optimal Büchi automata, and within reasonable time. Table 3 shows the evolution of the total time required to translate the 100 formulas.

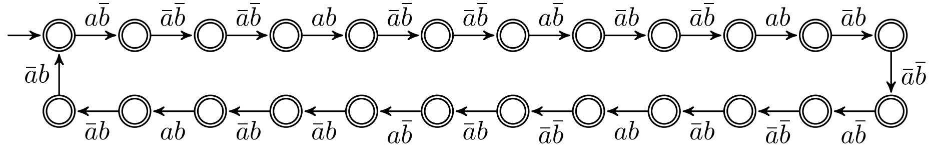


Figure 12 A Büchi automaton that recognizes C_3.

The translation of α_n, β_n, β'_n and ψ_n is nearly instantaneous: α_{20}, the longest to translate, requires 1.7s. Therefore the larger part of the run time summed in Table 3 comes from the φ_n family of formulas. For instance Spot 1.0 spends 90s computing just φ_{20}, and 40% of that time is spent in our inefficient degeneralization procedure. Comparatively, `ltl3ba`, which has some better handling for subformulas of that form, will translate φ_{20} in only 42s.

The variation between Spot 0.9 and Spot 0.9.1 corresponds to the introduction of the optimized translation of G discussed in Section 3.4, which is especially pertinent for the translation of φ_n: our LTL pre-processing will rewrite φ_n as $\mathsf{G}(\mathsf{F}(p1) \land \mathsf{F}(p2) \land \ldots \land \mathsf{F}(p_n))$ and from there the rules from Fig. 7 will avoid the creation of many useless Nxt[] variables for each of the F subformulas.

6.3 LTL counter

Rozier & Vardi (2007) compared 9 LTL translators, on various families of LTL formulas.

The first family of formulas they experimented on is scalable. For a given n they generated an LTL formula C_n that matches an infinite sequence of bits in which all the values of a n-bit counter have been concatenated. E.g., $C_3 = ((a \land (\mathsf{G}(a \rightarrow (\mathsf{X}(\neg a \land \mathsf{X}(\neg a \land \mathsf{X}\, a)))))) \land ((\neg b) \land \mathsf{X}(\neg b \land \mathsf{X}\neg b)) \land (\mathsf{G}((a \land \neg b) \rightarrow (\mathsf{X}((\mathsf{X}\mathsf{X}\, b) \land (((\neg a) \land (b \rightarrow \mathsf{X}\mathsf{X}\mathsf{X}\, b) \land ((\neg b) \rightarrow (\mathsf{X}\mathsf{X}\mathsf{X}\neg b))) \mathbin{\mathsf{U}} a))))) \land (\mathsf{G}((a \land b) \rightarrow (\mathsf{X}((\mathsf{X}\mathsf{X}\neg b) \land ((b \land (\neg a) \land \mathsf{X}\mathsf{X}\mathsf{X}\neg b) \mathbin{\mathsf{U}} (a \lor ((\neg a) \land (\neg b) \land (\mathsf{X}((\mathsf{X}\mathsf{X}\, b) \land (((\neg a) \land (b \rightarrow \mathsf{X}\mathsf{X}\mathsf{X}\, b) \land ((\neg b) \rightarrow \mathsf{X}\mathsf{X}\mathsf{X}\neg b)) \mathbin{\mathsf{U}} a))))))))))))$.[4] Such a formula will match a sequence consisting of $\begin{array}{l} a\text{: } 1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0\,\ldots \\ b\text{: } 0\,0\,0\,1\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,1\,1\,\ldots \end{array}$ repeated infinitely. Variable a signals the start of each value, while variable b iterates over the 3 bits of each value from least to most significant bit (000, 100, 010, …).

From this description it should be clear that the smallest automaton that can recognize C_n is a deterministic loop with $n2^n$ states and as many transitions. Fig. 12 shows this automaton for C_3. Any translator that constructs such an automaton explicitly will have a run time that is worse than exponential in n.

Figure 13 shows the run time taken by `ltl2ba`, `ltl3ba`, and three versions of Spot to translate C_n for increasing n. Executions were limited to 15min. Other tools are not shown, as they already fail to translate C_4 (sometimes even C_1) within this limit.

All these tools have been configured to not perform any pre- and post-processings. Also we patched `ltl2ba`, `ltl3ba`, so they would stop right after having constructing a TGBA without constructing a Büchi automaton. Therefore we are only measuring the scalability of the core LTL-to-TGBA translation algorithm in each of these tools. (We verified that each tool was slower with pre-processing turned on, which means that LTL rewriting are of no help on the C_n family of formulas.)

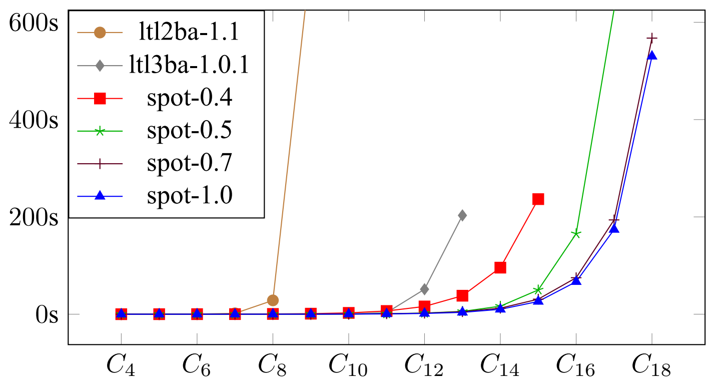


Figure 13 Run time of different tools on the translation of LTL counter formulas. Spot 0.4 was the version used by Rozier & Vardi (2007) in their experiments. Spot 0.7 was the version used for our experiments at VECOS'11 (Duret-Lutz 2011).

7 Conclusion

We have presented the main ingredients of the translation module of Spot. The core of the translation is the BDD-based tableau construction of Couvreur (1999), which has been extended in several ways: more determinism (a suggestion of Couvreur himself), some simplifications of the translations of subformulas of G, and a reduction of the number of promises required. This translation is preceded by a huge number of LTL rewriting rules, and followed by several post-processings to reduce the size of the produced automaton. This entire chain produces small automata that tend to be very deterministic, although not necessarily as fast as other translators such as `ltl3ba`.

Our implementation is extensively tested using both handwritten and random LTL formulas, and cross-compared to other translators using tools such LBTT (Tauriainen & Heljanko 2002) or `ltlcross`, a Spot-based reimplementation.

The degeneralization algorithm appears to be a weak point in Spot, and the historical reason is that we seldom use it in practice. When building a model checker on top of Spot, we implement the automata-theoretic approach using TGBA directly.

Although we have not discussed this, the implementation of this translation in Spot has been extended to support PSL (Accellera 2004), and our post-processings also include algorithms to output monitors (Tabakov & Vardi 2010) and testing automata (Ben Salem et al. 2011, 2012).

It has been argued (Cichoń et al. 2009, Tsay et al. 2011) that rather than optimizing an algorithm to try to produce the best automata always, it would be useful to create a database of optimal automata for commonly used formulas. However different uses may call for different definition of *optimal automaton*. In the context of model-checking, one usually wants to reduce the size of the product of the property with the system, and translating the property into a small automaton that is the most deterministic possible usually helps (Sebastiani & Tonetta 2003), but it is not always clear if more determinism justify additional states. In the context of monitoring, where an automaton is monitoring a running process, a deterministic automaton is preferred. In the context of synthesis of reactive systems, Ehlers & Finkbeiner (2010) prefers to minimize the number of states at the expense of determinism.

Also different kinds of automata can be used for verification: model checking with TGBA is usually better than model checking with Büchi automata when the formula incur a lot of acceptance marks (Couvreur et al. 2005). Using testing automata also appears promising (Geldenhuys & Hansen 2006, Ben Salem et al. 2011). A database should therefore not be limited to Büchi automata.

While we agree that such a database, like the *Büchi Store* project (Tsay et al. 2011), is useful, we still believe that it is important to have a translation that is efficient and versatile enough to be tuned to the needs of a particular situation.

Acknowledgments

The author would like to thank Kristin Rozier (Robust Software Engineering team at NASA), Rüdiger Ehlers (Saarland University), Christian Dax (formerly at ETH Zürick), Tomàš Babiak and Jan Strejček (both at Masaryk University) for fruitful discussions and for sharing some of their tools. Felix Abecassis (Epita student) wrote a large part of the WDBA minimization in Spot. Thomas Badie (Epita student) wrote the third (and current) implementation of the simulation algorithms. Denis Poitrenaud (LIP6) and the anonymous reviewers reported several mistakes in draft versions of this paper.

References

Accellera (2004), *Property Specification Language Reference Manual v1.1.* `http://www.eda.org/vfv/`.

Babiak, T., Badie, T., Duret-Lutz, A., Křetínský, M. & Strejček, J. (2013), Compositional approach to suspension and other improvements to LTL translation, *in* 'Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13)', Vol. 7976 of *Lecture Notes in Computer Science*, Springer, pp. 81–98.

Babiak, T., Křetínský, M., Řeehák, V. & Strejček, J. (2012), LTL to Büchi automata translation: Fast and more deterministic, *in* 'Proc. of the 18th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)', Vol. 7214 of *LNCS*, Springer, pp. 95–109.

Ben Salem, A. E., Duret-Lutz, A. & Kordon, F. (2011), Generalized Büchi automata versus testing automata for model checking, *in* 'Proc. of the second International Workshop on Scalable and Usable Model Checking for Petri Net and other models of Concurrency (SUMO'11)', Vol. 626 of *Workshop Proceedings*, CEUR, Newcastle, UK.

Ben Salem, A. E., Duret-Lutz, A. & Kordon, F. (2012), Model checking using generalized testing automata, *in* 'Transactions on Petri Nets and Other Models of Concurrency (ToPNoC VI)', Vol. 7400 of *LNCS*, Springer, pp. 94–112.

Bryant, R. E. (1986), 'Graph-based algorithms for boolean function manipulation', *IEEE Transactions on Computers* **35**(8), 677–691.

Černá, I. & Pelánek, R. (2003), Relating hierarchy of temporal properties to model checking, *in* 'Proc. of the 28th Int. Symp. on Mathematical Foundations of Computer Science (MFCS'03)', Vol. 2747 of *LNCS*, Springer-Verlag, Bratislava, Slovak Republic, pp. 318–327.

Cichoń, J., Czubak, A. & Jasiński, A. (2009), Minimal Büchi automata for certain classes of LTL formulas, *in* 'Proc. of the Fourth Int. Conf. on Dependability of Computer Systems (DEPCOS'09)', IEEE Computer Society, pp. 17–24.

Clarke, E. M., Grumberg, O. & Peled, D. A. (2000), *Model Checking*, The MIT Press.

Couvreur, J.-M. (1999), On-the-fly verification of temporal logic, *in* 'Proc. of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)', Vol. 1708 of *LNCS*, Springer-Verlag, Toulouse, France, pp. 253–271.

Couvreur, J.-M., Duret-Lutz, A. & Poitrenaud, D. (2005), On-the-fly emptiness checks for generalized Büchi automata, *in* 'Proc. of the 12th Int. SPIN Workshop on Model Checking of Software (SPIN'05)', Vol. 3639 of *LNCS*, Springer, pp. 143–158.

Dax, C., Eisinger, J. & Klaedtke, F. (2007), Mechanizing the powerset construction for restricted classes of ω-automata, *in* 'Proc. of the 5th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'07)', Vol. 4762 of *LNCS*, Springer.

Duret-Lutz, A. (2011), LTL translation improvements in Spot, *in* 'Proc. of the 5th Int. Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'11)', Electronic Workshops in Computing, British Computer Society, Tunis, Tunisia.

Duret-Lutz, A. & Poitrenaud, D. (2004), SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata, *in* 'Proc. of the 12th Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)', IEEE Computer Society Press, Volendam, The Netherlands, pp. 76–83.

Dwyer, M. B., Avrunin, G. S. & Corbett, J. C. (1998), Property specification patterns for finite-state verification, *in* 'Proc. of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)', ACM Press, New York, pp. 7–15.

Ehlers, R. & Finkbeiner, B. (2010), On the virtue of patience: minimizing Büchi automata, *in* 'Proc. of the 17th international SPIN conference on Model checking software (SPIN'10)', Vol. 6349 of *LNCS*, Springer, pp. 129–145.

Etessami, K. & Holzmann, G. J. (2000), Optimizing Büchi automata, *in* C. Palamidessi, ed., 'Proc. of the 11th Int. Conf. on Concurrency Theory (Concur'00)', Vol. 1877 of *LNCS*, Springer-Verlag, Pennsylvania, USA, pp. 153–167.

Fritz, C. (2003), Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata, *in* 'Proc. of the 8th Int. Conf. on Implementation and Application of Automata (CIAA'03)', Vol. 2759 of *LNCS*, Springer-Verlag, Santa Barbara, California, pp. 35–48.

Gastin, P. & Oddoux, D. (2001), Fast LTL to Büchi automata translation, *in* 'Proc. of the 13th Int. Conf. on Computer Aided Verification (CAV'01)', Vol. 2102 of *LNCS*, Springer-Verlag, Paris, France, pp. 53–65.

Geldenhuys, J. & Hansen, H. (2006), Larger automata and less work for LTL model checking, *in* 'Proc. of the 13th Int. SPIN Workshop (SPIN'06)', Vol. 3925 of *LNCS*, Springer, pp. 53–70.

Gerth, R., Peled, D., Vardi, M. Y. & Wolper, P. (1996), Simple on-the-fly automatic verification of linear temporal logic, *in* 'Proc. of the 15th Workshop on Protocol Specification Testing and Verification (PSTV'95)', Chapman & Hall, Warsaw, Poland, pp. 3–18.

Giannakopoulou, D. & Lerda, F. (2002), From states to transitions: Improving translation of LTL formulæ to Büchi automata, *in* 'Proc. of the 22nd IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'02)', Vol. 2529 of *LNCS*, Springer-Verlag, Houston, Texas, pp. 308–326.

Löding, C. (2001), 'Efficient minimization of deterministic weak ω-automata', *Information Processing Letters* **79**(3), 105–109.

Manna, Z. & Pnueli, A. (1990), A hierarchy of temporal properties, *in* 'Proc. of the sixth annual ACM Symposium on Principles of distributed computing (PODC'90)', ACM, New York, NY, USA, pp. 377–410.

Minato, S. (1992), Fast generation of irredundant sum-of-products forms from binary decision diagrams, *in* 'Proc. of the third Synthesis and Simulation and Meeting International Interchange workshop (SASIMI'92)', Kobe, Japan, pp. 64–73.

Oddoux, D. (2003), Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéaires, PhD thesis, Universitée Paris 7, Paris, France.

Pelánek, R. (2008), 'Properties of state spaces and their applications', *STTT* **10**(5), 443–454.

Rönkkö, M. (1999), 'LBT: LTL to Büchi conversion', `http://www.tcs.hut.fi/Software/maria/tools/lbt/`. Implements the tableau construction from Gerth et al. (1996).

Rozier, K. Y. & Vardi, M. Y. (2007), LTL satisfiability checking, *in* 'Proc. of the 12th Int. SPIN Workshop on Model Checking of Software (SPIN'07)', Vol. 4595 of *LNCS*, Springer, pp. 149–167.

Sebastiani, R. & Tonetta, S. (2003), "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking, *in* 'Proc. of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)', Vol. 2860 of *LNCS*, Springer-Verlag, L'Aquila, Italy, pp. 126–140.

Somenzi, F. & Bloem, R. (2000), Efficient Büchi automata for LTL formulæ, *in* 'Proc. of the 12th Int. Conf. on Computer Aided Verification (CAV'00)', Vol. 1855 of *LNCS*, Springer-Verlag, Chicago, Illinois, USA, pp. 247–263.

Tabakov, D. & Vardi, M. Y. (2010), Optimized temporal monitors for SystemC, *in* 'Proc. of the 1st Int. Conf. on Runtime Verification (RV'10)', Vol. 6418 of *LNCS*, Springer, pp. 436–451.

Tauriainen, H. (2006), Automata and Linear Temporal Logic: Translation with Transition-based Acceptance, PhD thesis, Helsinki University of Technology, Espoo, Finland.

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

Tsay, Y.-K., Tsai, M.-H., Chang, J.-S. & Chang, Y.-W. (2011), Büchi store: An open repository of büchi automata, *in* 'Proc. of the 17th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)', Vol. 6605 of *LNCS*, Springer, pp. 262–266.

Vardi, M. Y. (1996), An automata-theoretic approach to linear temporal logic, *in* F. Moller & G. M. Birtwistle, eds, 'Proc. of the 8th Banff Higher Order Workshop (Banff'94)', Vol. 1043 of *LNCS*, Springer-Verlag, Banff, Alberta, Canada, pp. 238–266.

Vardi, M. Y. (2007), Automata-theoretic model checking revisited, *in* 'Proc. of the 8th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'07)', Vol. 4349 of *LNCS*, Springer, Nice, France. Invited paper.

Notes

[1]Promises should not be mistaken for co-Büchi acceptance conditions. A co-Büchi acceptance condition $\mathcal{F}$ accepts runs that stay in $\mathcal{F}$ continuously; conversely a promise accepts runs that do not make the promise continuously.

[2]See the file `doc/tl/tl.pdf` in the Spot distribution.

[3]`http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

[4]After installing Spot, this formula can be generated with `genltl --rv-counter-linear=3`.