Fast LTL to Büchi Automata Translation

Paul Gastin and Denis Oddoux

LIAFA, Université Paris 7, Paris, France
{Paul.Gastin,Denis.Oddoux}@liafa.jussieu.fr

Abstract. We present an algorithm to generate Büchi automata from LTL formulae. This algorithm generates a very weak alternating co-Büchi automaton and then transforms it into a Büchi automaton, using a generalized Büchi automaton as an intermediate step. Each automaton is simplified on-the-fly in order to save memory and time. As usual we simplify the LTL formula before any treatment. We implemented this algorithm and compared it with Spin: the experiments show that our algorithm is much more efficient than Spin. The criteria of comparison are the size of the resulting automaton, the time of the computation and the memory used. Our implementation is available on the web at the following address: `http://verif.liafa.jussieu.fr/ltl2ba`

1 Introduction

To prove that a program satisfies some property, a standard method is to use Linear Time Logic (LTL) model checking. When the property is expressed with an LTL formula, the model checker usually transforms the negation of this formula into a Büchi automaton, builds the product of that automaton with the program, and checks this product for emptiness. In this paper we focus on the generation of a Büchi automaton from an LTL formula, trying to improve the time and space of the computation and the size of the resulting automaton.

Spin [4] is a very popular LTL model checker. However, the algorithm it uses to generate a Büchi automaton from an LTL formula, presented in [3], may be quite slow and may need a large amount of memory, even for some usual LTL formulae. In particular, this algorithm has a very bad behaviour on formulae with fairness conditions: it is almost impossible to use Spin to generate a Büchi automaton from a formula containing 5 or more fairness conditions, both because of the computation time and of the memory needed. For example, consider a simple response formula $\mathrm{G}(q \rightarrow \mathrm{F}\, r)$ with n fairness conditions:

$$\theta_n = \neg((\mathrm{G}\,\mathrm{F}\, p_1 \wedge \ldots \wedge \mathrm{G}\,\mathrm{F}\, p_n) \rightarrow \mathrm{G}(q \rightarrow \mathrm{F}\, r)) \ . \quad (1)$$

A formula of this type is very often encountered in LTL model checking. Moreover, the fairness conditions and the right-hand side property are usually more complex. The value of n is very often greater than 5. Alas, in this case, Spin fails to produce the Büchi automaton within a reasonable amount of time and memory (see Table 1).

	Spin		Wring		EQLTL	LTL2BA−		LTL2BA	
	time	space	time	space	time	time	space	time	space
θ_1	0.18	460	0.56	4,100	16	0.01	9	0.01	9
θ_2	4.6	4,200	2.6	4,100	16	0.01	19	0.01	11
θ_3	170	52,000	16	4,200	18	0.01	86	0.01	19
θ_4	9,600	970,000	110	4,700	25	0.07	336	0.06	38
θ_5			1,000	6,500	135	0.70	1,600	0.37	48
θ_6			8,400	13,000	N/A	12	8,300	4.0	88
θ_7			72,000†	43,000†		220	44,000	32	175
θ_8						4,200	260,000	360	250
θ_9						97,000	1,600,000	3,000	490
θ_{10}								36,000	970

Table 1. Comparison on the formulae θ_n for $1 \leq n \leq 10$. Time is in sec, space in kB. (N/A) : no answer from the server within 24 h. (†) : the program died, giving no result

Spin's algorithm was improved by [1] (LTL2AUT), [2] (EQLTL), [10] (Wring): these papers did not modify the basis of the algorithm, but improved it using the same core algorithm, rewriting LTL formulae, and simplifying the resulting Büchi automaton. These improvements are quite efficient but the actual transformation of the LTL formula to a Büchi automaton, which is similar to the tableau construction explained in [3], may still perform badly on some natural formulae such as θ_n. Some experiments are presented in Table 1. Note that Wring is written in Perl while Spin and LTL2BA are written in C and that EQLTL is used through a web server. Hence the figures are still relevant but should not be compared litterally. See Sect. 7 for more details.

In this paper, we present a new algorithm to generate a Büchi automaton from an LTL formula. Our algorithm is not based on the tableau construction presented in [3]. Instead, using the classical construction (see e.g. [12]), we first produce an alternating automaton from the LTL formula, with n states where n is less than the size of the formula. This alternating automaton turns out to be *very weak* as shown by Rohde [9]. Thanks to that property, instead of generating directly a Büchi automaton with $2^n \times 2^n$ states, we are able to build first a generalized Büchi automaton, that is a Büchi automaton with labels and accepting conditions on transitions instead of states, with at most 2^n states. Using a generalized Büchi automaton is one of the most important improvements of our algorithm. The best solution would be to design a model-checking algorithm using directly this generalized Büchi automaton, but in order to compare our work with other ones and to use existing model-checking algorithms, we transform this automaton into a classical Büchi automaton. The method we use is very classical, and we obtain a Büchi automaton with at most $n \times 2^n$ states.

The second main improvement stems from our simplifications of the automata. Since our construction goes in several steps, we are able to simplify the automata at each step, improving the efficiency of the following steps. The simplifications dramatically reduce the number of states and transitions of the automata, especially of the generalized Büchi automaton. Moreover, each simplification is performed on-the-fly during the construction of each automaton. This is a major improvement on a posteriori simplifications. The amount of memory used is about the size of the simplified automaton, instead of being the size of the unsimplified automaton which may be quite huge. The time needed is also reduced dramatically because we are exploring a much smaller part of the automaton during the construction.

Using our new algorithm, we built a tool which is available on the web at `http://verif.liafa.jussieu.fr/ltl2ba`. Our tool is much more efficient than any other tool we have tried, in computation time and especially in memory. The results of our algorithm on the formulae θ_n with on-the-fly simplifications (LTL2BA) and with a posteriori simplifications (LTL2BA–) are detailed in Table 1. More experimental results are presented in Sect. 7. There we also discuss the size of the generated automaton. From this point of view also our algorithm is usually better than Spin though occasionally it may produce a bigger automaton. Note that Spin, LTL2BA– and LTL2BA give exactly the same resulting automaton on the formulae θ_n. Wring and EQLTL give bigger automata.

The paper is organized as follows. Section 2 begins with some preliminaries defining linear temporal logic and its semantics. Sections 3 to 5 describe our algorithm and some proofs of its correctness. Section 6 presents our simplification methods and Sect. 7 describes some experimental results.

2 Preliminaries: Linear Temporal Logic (LTL)

LTL was introduced to specify the properties of the executions of a system. A finite set Prop contains all atomic properties of states. With the standard Boolean operators ($\neg$, $\wedge$, $\vee$) we can only express static properties. For dynamical properties, we use temporal operators such as X (next), U (until), R (release), F (eventually) and G (always).

Definition 1 (syntax). *The set of* LTL formulae *on the set* Prop *is defined by the grammar* $\varphi ::= p \mid \neg\,\varphi \mid \varphi \vee \varphi \mid \mathrm{X}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi$, *where* p *ranges over* Prop.

The semantics of LTL usually defines whether an execution σ of a given system satisfies a formula. Actually the semantics only depends on the atomic propositions that stand in each state of σ. Then for our purpose we consider only sequences of sets of atomic propositions.

Definition 2 (semantics). *Let* $u = u_0u_1\ldots$ *be a word in* Σ^ω *with* $\Sigma = 2^{\mathrm{Prop}}$. *Let* φ *be an LTL formula. The relation* $u \models \varphi$ *(*u models φ*) is defined as follows:*

- $u \models p$ *if* $p \in u_0$,
- $u \models \neg\,\varphi_1$ *if* $u \not\models \varphi_1$,
- $u \models \varphi_1 \vee \varphi_2$ *if* $u \models \varphi_1$ *or* $u \models \varphi_2$,
- $u \models \mathrm{X}\,\varphi_1$ *if* $u_1u_2\ldots \models \varphi_1$,
- $u \models \varphi_1\,\mathrm{U}\,\varphi_2$ *if* $\exists k \geq 0$, $u_ku_{k+1}\ldots \models \varphi_2$ *and* $\forall 0 \leq i < k$, $u_iu_{i+1}\ldots \models \varphi_1$.

Only basic operators have been defined above. We will of course also use the derived operators defined by:

$$\mathrm{tt} \stackrel{def}{=} p \vee \neg p \ , \ \ \mathrm{ff} \stackrel{def}{=} \neg\, \mathrm{tt} \ , \ \ \varphi_1 \wedge \varphi_2 \stackrel{def}{=} \neg(\neg \varphi_1 \vee \neg \varphi_2) \ , \tag{2}$$

$$\varphi_1 \,\mathrm{R}\, \varphi_2 \stackrel{def}{=} \neg(\neg \varphi_1 \,\mathrm{U}\, \neg \varphi_2) \ , \ \ \mathrm{F}\,\varphi \stackrel{def}{=} \mathrm{tt}\,\mathrm{U}\,\varphi \ \text{ and } \ \mathrm{G}\,\varphi \stackrel{def}{=} \mathrm{ff}\,\mathrm{R}\,\varphi = \neg\,\mathrm{F}\,\neg\,\varphi \ . \tag{3}$$

An LTL formula that is neither a disjunction ($\vee$) nor a conjunction ($\wedge$) is called a *temporal formula*.

An LTL formula can be written in *negative normal form*, using only the predicates in Prop, their negations, and the operators $\vee$, $\wedge$, X, U, and R. Notice that this operation does not change the number of temporal operators of the formula. From now on, we suppose that every LTL formula is in negative normal form.

Example 1. Let $\theta = \neg(\mathrm{G}\,\mathrm{F}\,p \rightarrow \mathrm{G}(q \rightarrow \mathrm{F}\,r))$ be our running example along the paper. The negative normal form of θ is $(\mathrm{ff}\,\mathrm{R}\,(\mathrm{tt}\,\mathrm{U}\,p)) \wedge (tt\,\mathrm{U}\,(q \wedge (\mathrm{ff}\,\mathrm{R}\,\neg\, r)))$.

Before any construction our algorithm simplifies the formula, using a set of rewriting rules that reduce the number of temporal operators. This is relevant since the complexity of our algorithm is based on this number. Some of these rules are presented in [2],[10]. We will not discuss them in this paper.

3 LTL to Very Weak Alternating Automata (VWAA)

This section explains a classical construction: building a VWAA from an LTL formula. Alternating automata have been introduced by Muller and Schupp in [6],[7],[8]. Then in [9], Rohde defined VWAA as he needed them for a work on transfinite words. VWAA were also described in [5]. However, our definition is somewhat different from the classical one.

Definition 3. *A co-Büchi* very weak alternating co-Büchi automaton *is a five-tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where:*

- *Q is the set of states,*
- *Let Q' be the set of conjunctions of elements of Q. The empty conjunction is denoted by tt. We identify Q' with 2^Q in the following,*
- *Σ is the alphabet, and we let $\Sigma' = 2^\Sigma$,*
- *$\delta : Q \rightarrow 2^{\Sigma' \times Q'}$ is the transition function,*
- *$I \subseteq Q'$ is the set of initial states,*
- *$F \subseteq Q$ is the set of final states (co-Büchi),*
- *there exists a partial order on Q such that $\forall q \in Q$, all the states appearing in $\delta(q)$ are lower or equal to q.*

The definition of a classical alternating automaton would be the same except for the last condition on the partial order.

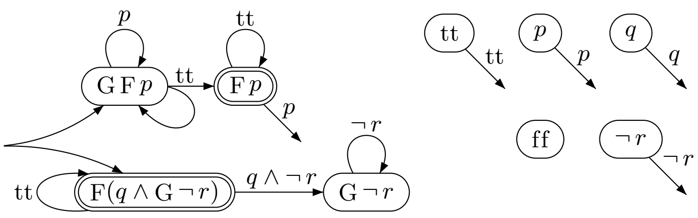

Fig. 1. Automaton $\mathcal{A}_\theta$. Some states (*right*) are unaccessible, they will be removed

Remark 1. The transition function looks different from the usual definition ($\Delta : Q \times \Sigma \to \mathcal{B}^+(Q)$). We made those changes for implementation reasons, in order to ease the manipulation of the data structures and to save time and space during the computation. The classical representation of our transition function is given by:

$$\Delta(q,a) = \bigvee_{\substack{(\alpha,e)\in\delta(q)\\ a\in\alpha}} e \ . \tag{4}$$

Conversely we may obtain our definition from the classical one, essentially by taking the disjunctive normal form. Hence the two definitions are equivalent.

Notice that in the transition function we use Σ' instead of Σ: so that transitions that differ only by the action can be gathered. In practice, this usually reduces a lot the number of transitions. However the automaton still reads words in Σ^ω.

Example 2. You can see the representation of a VWAA on Fig. 1. States in F are circled twice. Notice that arrows with the same origin represent one transition to a conjunction of states. In this example, we have:

- $I = \{\mathrm{G\,F}\,p \wedge \mathrm{F}(q \wedge \mathrm{G}\,\neg r)\}$,
- $\delta(p) = \{(\Sigma_p, \mathrm{tt})\}$ where $\Sigma_p = \{a \in \Sigma \mid p \in a\}$,
- $\delta(\mathrm{G\,F}\,p) = \{(\Sigma_p, \mathrm{G\,F}\,p), (\Sigma, \mathrm{G\,F}\,p \wedge \mathrm{F}\,p)\}$.

A *run* σ of $\mathcal{A}$ on a word $u_0u_1 \ldots \in \Sigma^\omega$ is a labeled DAG (V, E, λ) such that :

- V is partitioned in $\bigcup_{i=0}^{\infty} V_i$ with $E \subseteq \bigcup_{i=0}^{\infty} V_i \times V_{i+1}$,
- λ: $V \to Q$ is the labeling function,
- $\lambda(V_0) \in I$ and $\forall x \in V_i$, $\exists(\alpha, e) \in \delta(\lambda(x))$, $u_i \in \alpha$ and $e = \lambda(E(x))$.

A run σ is *accepting* if any (infinite) branch in σ has only a finite number of nodes labeled in F (co-Büchi acceptance condition). $\mathcal{L}(\mathcal{A})$ is the set of words on which there exists an accepting run of $\mathcal{A}$. Note that, Büchi and co-Büchi acceptance conditions are equivalent for VWAA; one only has to replace F by $Q \setminus F$.

Example 3. Here is an example of an accepting run of the automaton $\mathcal{A}_\theta$:

$$\begin{tikzcd}[column sep=small]
\mathrm{G\,F}\,p \arrow[r, "\emptyset"] \arrow[dr] & \mathrm{G\,F}\,p \arrow[r, "\{q,r\}"] \arrow[dr] & \mathrm{G\,F}\,p \arrow[r, "\{p,q\}"] & \mathrm{G\,F}\,p \arrow[r, "\{p\}"] & \mathrm{G\,F}\,p \arrow[r, dashed, "\{p\}"] & {} \\
& \mathrm{F}\,p \arrow[r] & \mathrm{F}\,p & & & \\
\mathrm{F}(q \wedge \mathrm{G} \neg r) \arrow[r] & \mathrm{F}(q \wedge \mathrm{G} \neg r) \arrow[r] & \mathrm{F}(q \wedge \mathrm{G} \neg r) \arrow[dr] & & & \\
& & & \mathrm{G} \neg r \arrow[r] & \mathrm{G} \neg r \arrow[r, dashed] & {}
\end{tikzcd}$$

In the definition of the VWAA associated with an LTL formula, we use two new operators. $\otimes$ helps treating conjunctions, and $\overline{\psi}$ gives roughly the DNF of ψ, allowing us to restrict the states of the automaton to the temporal subformulae of φ.

Definition 4. *For $J_1, J_2 \in 2^{\Sigma' \times Q'}$ we define*
$J_1 \otimes J_2 = \{(\alpha_1 \cap \alpha_2, e_1 \wedge e_2) \mid (\alpha_1, e_1) \in J_1$ *and* $(\alpha_2, e_2) \in J_2\}$,
For an LTL formula ψ we define $\overline{\psi}$ by: $\overline{\psi} = \{\psi\}$ if ψ is a temporal formula, $\overline{\psi_1 \wedge \psi_2} = \{e_1 \wedge e_2 \mid e_1 \in \overline{\psi_1}$ and $e_2 \in \overline{\psi_2}\}$ and $\overline{\psi_1 \vee \psi_2} = \overline{\psi_1} \cup \overline{\psi_2}$.

Here is the first step of our algorithm, building a VWAA from an LTL formula. Notice that the number of states of this automaton is at most the size of the formula.

Step 1. Let φ be an LTL formula on a set Prop. We define the VWAA $\mathcal{A}_\varphi$ by:

- Q is the set of temporal subformulae of φ,
- $\Sigma = 2^{\mathrm{Prop}}$,
- $I = \overline{\varphi}$,
- F is the set of until subformulae of φ, that is formulae of type $\psi_1 \,\mathrm{U}\, \psi_2$,
- δ is defined as follows (Δ extends δ to all subformulae of φ):

$$\begin{cases}
\delta(\mathrm{tt}) = \{(\Sigma, \mathrm{tt})\} \\
\delta(p) = \{(\Sigma_p, \mathrm{tt})\} \text{ where } \Sigma_p = \{a \in \Sigma \mid p \in a\} \\
\delta(\neg p) = \{(\Sigma_{\neg p}, \mathrm{tt})\} \text{ where } \Sigma_{\neg p} = \Sigma \backslash \Sigma_p \\
\delta(\mathrm{X}\,\psi) = \{(\Sigma, e) \mid e \in \overline{\psi}\} \\
\delta(\psi_1 \,\mathrm{U}\, \psi_2) = \Delta(\psi_2) \ \cup (\Delta(\psi_1) \otimes \{(\Sigma, \psi_1 \,\mathrm{U}\, \psi_2)\}) \\
\delta(\psi_1 \,\mathrm{R}\, \psi_2) = \Delta(\psi_2) \ \otimes (\Delta(\psi_1) \cup \{(\Sigma, \psi_1 \,\mathrm{R}\, \psi_2)\})
\end{cases}$$
$$\begin{cases}
\Delta(\psi) = \delta(\psi) \text{ if } \psi \text{ is a temporal formula} \\
\Delta(\psi_1 \vee \psi_2) = \Delta(\psi_1) \cup \Delta(\psi_2) \\
\Delta(\psi_1 \wedge \psi_2) = \Delta(\psi_1) \otimes \Delta(\psi_2)
\end{cases}$$

Using the partial order "subformula of" it is easy to prove that $\mathcal{A}_\varphi$ is very weak.

Remark 2. One can notice that the elements of Σ' used in our definition are intersections of the sets Σ, Σ_p and $\Sigma_{\neg p}$. Hence, they can be denoted by conjunctions of literals, as in the following examples : $p \wedge q \wedge \neg r$ for $\Sigma_p \cap \Sigma_q \cap \Sigma_{\neg r}$, tt for Σ. Note that intersection and test for inclusion can be easily performed with this representation.

Example 4. Figure 1 shows the result of Step 1 on the formula θ defined in Ex. 1.

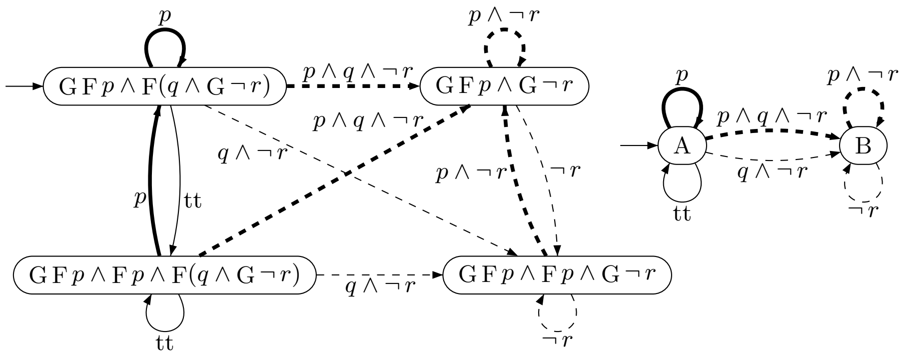

Fig. 2. Automaton $\mathcal{G}_{\mathcal{A}_\theta}$, before (*left*) and after (*right*) simplification

Theorem 1. $\mathcal{L}(\mathcal{A}_\varphi) = \{u \in \Sigma^\omega \mid u \models \varphi\}$.

Proof. The idea of the proof is to show recursively that for any subformula ψ of φ, the language accepted by $\mathcal{A}_\varphi$ with $I = \overline{\psi}$ is equal to $\{u \in \Sigma^\omega \mid u \models \psi\}$. The main difficulties are encountered for $\psi = \psi_1 \,\mathrm{U}\, \psi_2$ (this is where the acceptance condition comes into play) and $\psi = \psi_1 \,\mathrm{R}\, \psi_2$. □

4 VWAA to Generalized Büchi Automata (GBA)

At that point we have obtained a VWAA for our LTL formula φ. The problem is that the usual method to transform an alternating automaton into a Büchi automaton produces an automaton that is much too big. This is why we generate first a GBA, which is a Büchi automaton with *several* acceptance conditions on *transitions* instead of states.

Definition 5. *A* generalized Büchi automaton *is a five-tuple* $\mathcal{G} = (Q, \Sigma, \delta, I, \mathcal{T})$ *where :*

- Q *is the set of states,*
- Σ *is the alphabet, and we let* $\Sigma' \subseteq 2^{\Sigma}$,
- $\delta : Q \to 2^{\Sigma' \times Q}$ *is the transition function,*
- $I \subseteq Q$ *is the set of initial states,*
- $\mathcal{T} = \{T_1, \ldots, T_r\}$ *where* $T_j \subseteq Q \times \Sigma' \times Q$ *are the accepting transitions.*

Example 5. The automata on Fig. 2 are examples of GBAs. In these examples, $r = 2$: dashed transitions are in T_1 and bold transitions are in T_2. An accepting run has to use infinitely many dashed transitions and infinitely many bold transitions.

A *run* σ of $\mathcal{G}$ on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a sequence $q_0, q_1, \ldots$ of elements of Q such that $q_0 \in I$ and $\forall i \geq 0$, $\exists \alpha_i \in \Sigma'$ such that $u_i \in \alpha_i$ and $(\alpha_i, q_{i+1}) \in \delta(q_i)$. A run σ is *accepting* if for each $1 \leq j \leq r$ it uses infinitely many transitions from T_j. $\mathcal{L}(\mathcal{G})$ is the set of words on which there exists an accepting run of $\mathcal{G}$.

Here is the second step of our algorithm, building a GBA from a co-Büchi VWAA. It can be of course applied to any VWAA, and not only to an automaton issued from Step 1. $\mathcal{G}_\mathcal{A}$ has at most $2^{|Q|}$ states and $|F|$ acceptance sets.

Step 2. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a VWAA with co-Büchi acceptance conditions. We define the GBA $\mathcal{G}_\mathcal{A} = (Q', \Sigma, \delta', I, \mathcal{T})$ where:

- $Q' = 2^Q$ is identified with conjunctions of states as explained in Definition 3,
- $\delta''(q_1 \wedge \ldots \wedge q_n) = \bigotimes_{i=1}^{n} \delta(q_i)$,
- δ' is the set of $\preccurlyeq$-minimal transitions of δ'' where the relation $\preccurlyeq$ is defined by $t' \preccurlyeq t$ if $t = (e, \alpha, e')$, $t' = (e, \alpha', e'')$, $\alpha \subseteq \alpha'$, $e'' \subseteq e'$, and $\forall T \in \mathcal{T}$, $t \in T \Rightarrow t' \in T$,
- $\mathcal{T} = \{T_f \mid f \in F\}$ where
  $T_f = \{(e, \alpha, e') \mid f \notin e' \text{ or } \exists(\beta, e'') \in \delta(f),\ \alpha \subseteq \beta \text{ and } f \notin e'' \subseteq e'\}$.

Remark 3. One may notice that using $f \notin e$ instead of $f \notin e'$ in the definition of T_f would have been more intuitive, since it corresponds to the case where in the run of $\mathcal{A}$ there is no edge with both ends labeled by f. But our definition is also correct. The proof of the following main theorem is more complicated with this definition but the experimental results are much better with it, especially regarding the simplifications.

Example 6. Figure 2 shows the result of Step 2 on the automaton $\mathcal{A}_\theta$ of Fig. 1.

Theorem 2. $\mathcal{L}(\mathcal{G}_\mathcal{A}) = \mathcal{L}(\mathcal{A})$.

Remark 4. This is the point where we need the alternating automaton to be very weak (this theorem is false for classical alternating automata). Consider an infinite branch in a run of $\mathcal{A}$ on a given word : since $\mathcal{A}$ is very weak, the sequence of the labels on this branch is decreasing, and has to be ultimately constant since Q is finite. Then "having only a finite number of nodes labelled in F" is equivalent to "having an infinite number of nodes labelled in $Q \backslash F$". This is crucial in the proof.

Proof. Let $\sigma = (V, E, \lambda)$ be an accepting run of $\mathcal{A}$ on a word $u = u_0 u_1 \ldots$ $V = \bigcup_{i \geq 0} V_i$, $E = \bigcup_{i \geq 0} E_i$, with $E_i \subseteq V_i \times V_{i+1}$. We are first going to build a new run of $\mathcal{A}$ on u, redefining gradually the sets V_i and E_i to $V'_i \subseteq V_i$ and E'_i, $\forall i \geq 0$.

Let $V'_0 = V_0$. Now suppose that V'_i has been defined. By definition of a run, $\forall x \in V'_i$, $\exists \alpha_x$ such that $u_i \in \alpha_x$, and $(\alpha_x, e_x) \in \delta(\lambda(x))$ where $e_x = \lambda(E_i(x))$. Let $\alpha = \bigcap_{x \in V'_i} \alpha_x$ and $e = \bigcup_{x \in V'_i} e_x$.

By definition of δ'', $t = (\lambda(V'_i), \alpha, e)$ is in δ'': there exists a transition $t' = (\lambda(V'_i), \alpha', e')$ in δ' such that $t' \preccurlyeq t$ and t' is minimal. Note that t' is a transition of $\mathcal{G}_\mathcal{A}$, and that $u_i \in \alpha \subseteq \alpha'$. Since $t' \in \delta' \subseteq \delta''$, $\forall x \in V'_i$, $\exists(\alpha'_x, e'_x) \in \delta(\lambda(x))$ such that $\alpha' = \bigcap_{x \in V'_i} \alpha'_x$ and $e'_x = \bigcup_{x \in V'_i} e'$.

Moreover $\forall x \in V_i'$ such that $\lambda(x) = f \in F$ and $t' \in T_f$, there exists $(\alpha_x'', e_x'') \in \delta(f)$ such that $f \notin e_x'' \subseteq e'$ and $\alpha' \subseteq \alpha_x''$. For all other elements x of V_i', let $e_x'' = e_x'$ and $\alpha_x'' = \alpha_x'$.

Let $V_{i+1}' = \{y \in V_{i+1} \mid \lambda(y) \in e'\}$ and $E_i' = \{(x, y) \in V_i' \times V_{i+1}' \mid \lambda(y) \in e_x''\}$. Note that $\lambda(E_i'(x)) = e_x''$ since $e_x'' \subseteq e' = \lambda(V_i')$ and that $E_i'(V_i')$ may be *strictly* contained in V_{i+1}'.

Claim. $\forall i \geq 0$, $\forall f \in F$, the following property holds:
if $\exists (x, y) \in E_i'$, $\lambda(x) = \lambda(y) = f$ then $\exists (x, y) \in E_i$, $\lambda(x) = \lambda(y) = f$.

Proof. If $\forall x \in V_i'$, $\lambda(x) \neq f$ then the claim is true. Otherwise $\exists x \in V_i' \subseteq V_i$, $\lambda(x) = f$. Assume that $\exists y \in E_i'(x)$ with $\lambda(y) = f$. Then we have $f \in e_x''$, and by definition of e_x'' we deduce that $t' \notin T_f$. Since $t' \preccurlyeq t$ we have $t \notin T_f$ and we deduce easily that $f \in e_x$, which proves the claim.

Let $V' = \bigcup_{i \geq 0} V_i'$, $E' = \bigcup_{i \geq 0} E_i'$ and λ' be the restriction of λ to V'. From the construction, one can easily see that $\sigma' = (V', E', \lambda')$ is a new run of $\mathcal{A}$ on u. We show first that σ' is an accepting run. Suppose that σ' is not accepting: since $\mathcal{A}$ is very weak, the labels on an infinite branch of a run are ultimately constant. Hence if σ' is not accepting, then there exists an infinite branch of σ' ultimately labeled by some $f \in F$. Using the claim, there exists in σ an infinite branch which is ultimately labeled by f. This is impossible since σ is accepting.

Let $e_i = \lambda(V_i')$, $\forall i \geq 0$. We have $e_0 = \lambda(V_0) \in I$ and from our construction we get $\forall i \geq 0$, $\exists \alpha_i$ such that $u_i \in \alpha_i$ and (e_i, α_i, e_{i+1}) is a transition of $\mathcal{G}_{\mathcal{A}}$: $\sigma'' = e_0, e_1, \ldots$ is a run of $\mathcal{G}_{\mathcal{A}}$ on u. Now let us prove that σ'' is accepting. Let $i \geq 0$ and $f \in F$. We intend to prove that at some depth $j \geq i$ the transition (e_j, α_j, e_{j+1}) is in T_f.

If $f \notin e_{i+1}$ then $j = i$ will do. Otherwise let $j > i$ be the smallest depth where $(f, f) \notin \lambda(E_j')$. Note that j exists, otherwise there would be an infinite branch in σ' ultimately labeled by f and σ' would not be accepting. Since we know that $f \in e_j$, let x be the node of V_j' labeled by f. From our construction we know that $\exists (e_x'', \alpha_x'') \in \delta(f)$, $f \notin e_x'' \subseteq e_{j+1}$ and $\alpha_j \subseteq \alpha_x''$. We can conclude that (e_j, α_j, e_{j+1}) is in T_f.

Therefore, from any accepting run σ of $\mathcal{A}$, we have built an accepting run σ'' of $\mathcal{G}_{\mathcal{A}}$ on the same word and we get the first inclusion $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{G}_{\mathcal{A}})$.

Conversely let $\sigma' = e_0, e_1, \ldots$ be an accepting run of $\mathcal{G}_{\mathcal{A}}$ on a word $u = u_0 u_1 \ldots$ Hence $e_0 \in I$ and $\forall i \geq 0$, $\exists \alpha_i$, $u_i \in \alpha_i$ and $(\alpha_i, e_{i+1}) \in \delta'(e_i)$. Let $V = \bigcup_{i \geq 0} V_i$ where $V_i = \{(p, i) \mid p \in e_i\}$ and let $\lambda(p, i) = p$ so that $\lambda(V_i) = e_i$.

By definition of δ', $\forall x \in V_i$, $\exists (\alpha_x, e_x) \in \delta(\lambda(x))$ such that $e_x \subseteq e_{i+1}$ and $\alpha_i \subseteq \alpha_x$. Moreover $\forall f \in F$, if $(e_i, \alpha_i, e_{i+1}) \in T_f$ then either $f \notin e_i$, or $\lambda(x) = f$ for some x in V_i and in that case we can choose α_x and e_x such that $f \notin e_x$. Let E be defined by $(x, y) \in E$ if $\exists i \geq 0$, $x \in V_i$, $y \in V_{i+1}$ and $\lambda(y) \in e_x$.

We can easily see that $\sigma = (V, E, \lambda)$ is a run of $\mathcal{A}$ on u. Now suppose that σ is not accepting: as we proved before, there would exist in σ an infinite branch with all nodes ultimately labeled by some $f \in F$. But σ' is accepting so it has

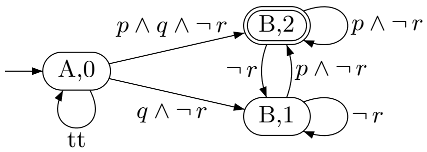

Fig. 3. Automaton $\mathcal{B}_{\mathcal{G}_{\mathcal{A}_\theta}}$ after simplification

infinitely many transitions in T_f, and for each such transition there is no edge in E with both ends labeled by f at the corresponding depth. Hence this is impossible.

Therefore from any accepting run σ' of $\mathcal{G}_\mathcal{A}$, we have built an accepting run σ of $\mathcal{A}$ on the same word, proving the converse inclusion $\mathcal{L}(\mathcal{G}_\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A})$. $\square$

5 GBA to Büchi Automata (BA)

At that point we have obtained a GBA for our LTL formula φ. We simply have to transform it into a BA to complete our algorithm. This construction is quite easy and well-known, but for the sake of completeness we explain it briefly. We will begin by defining a BA, using once more the same modifications concerning the alphabet and the transition function.

Definition 6. *A* Büchi automaton *is a five-tuple* $\mathcal{B} = (Q, \Sigma, \delta, I, F)$ *where:*

- Q *is the set of states,*
- Σ *is the alphabet, and we let* $\Sigma' \subseteq 2^\Sigma$*,*
- $\delta : Q \to 2^{\Sigma' \times Q}$ *is the transition function,*
- $I \subseteq Q$ *is the set of initial states,*
- $F \subseteq Q$ *is the set of repeated states (Büchi condition).*

A *run* σ of $\mathcal{B}$ on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a sequence $q_0, q_1, \ldots$ of elements of Q such that $q_0 \in I$ and $\forall i \geq 0$, $\exists \alpha_i \in \Sigma'$ such that $u_i \in \alpha_i$ and $(\alpha_i, q_{i+1}) \in \delta(q_i)$. A run σ is *accepting* if there exists infinitely many states in F. $\mathcal{L}(\mathcal{B})$ is the set of words on which there exists an accepting run of $\mathcal{B}$.

Here is the third step of our algorithm, building a BA from a GBA. If $\mathcal{B}$ has n states and r acceptance conditions, then $\mathcal{B}_\mathcal{G}$ has at most $(r+1) \times n$ states.

Step 3. Let $\mathcal{G} = (Q, \Sigma, \delta, I, \mathcal{T})$ be a GBA with $\mathcal{T} = \{T_1, \ldots, T_r\}$. We define the BA $\mathcal{B}_\mathcal{G} = (Q \times \{0, \ldots, r\}, \Sigma, \delta', I \times \{0\}, Q \times \{r\})$ where:

- $\delta'((q, j)) = \{(\alpha, (q', j')) \mid (\alpha, q') \in \delta(q) \text{ and } j' = next(j, (q, \alpha, q'))\}$.

with $\mathrm{next}(j, t) = \begin{cases} \max\{j \leq i \leq r \mid \forall j < k \leq i,\ t \in T_k\} \text{ if } j \neq r \\ \max\{0 \leq i \leq r \mid \forall 0 < k \leq i,\ t \in T_k\} \text{ if } j = r \end{cases}$

Example 7. Figure 3 shows the result of Step 3 on the automaton $\mathcal{G}_{\mathcal{A}_\theta}$ of Fig. 2.

Theorem 3. $\mathcal{L}(\mathcal{B}_\mathcal{G}) = \mathcal{L}(\mathcal{B})$.

Remark 5. There exist many similar algorithms transform a GBA into a BA. They often consist in building the synchronous product of the GBA with some automaton verifying that every acceptance condition is verified infinitely often. This automaton differs from one algorithm to another. We chose one that gives good results for the size of the resulting BA after simplification.

6 Simplification

Simplification is really important in our algorithm. Since each step produces a new automaton from the result of the previous step, the more we simplify each result, the faster our algorithm is and the least memory it uses.

After each step, we simplify the automaton obtained, using iteeratively three rules until no more simplification occurs:

– A state that is not accessible can be removed,
– If a transition t_1 implies a transition t_2, then t_2 can be removed,

	$t_1 = (q, \alpha_1, q_1)$ implies $t_2 = (q, \alpha_2, q_2)$ if
In a VWAA,	$\alpha_2 \subseteq \alpha_1$ and $q_1 \subseteq q_2$
In a GBA,	$\alpha_2 \subseteq \alpha_1$, $q_1 = q_2$ and $\forall t \in \mathcal{T}$, $t_2 \in T \Rightarrow t_1 \in T$
In a BA,	$\alpha_2 \subseteq \alpha_1$ and $q_1 = q_2$

– If two states q_1 and q_2 are equivalent, then they can be merged.

	q_1 and q_2 are equivalent if
In a VWAA,	$\delta(q_1) = \delta(q_2)$ and $q_1 \in F \iff q_2 \in F$
In a GBA,	$\delta(q_1) = \delta(q_2)$ and $\forall (\alpha, q') \in \delta(q_1)$, $\forall T \in \mathcal{T}$, $(q_1, \alpha, q') \in T \iff (q_2, \alpha, q') \in T$
In a BA,	$\delta(q_1) = \delta(q_2)$ and $q_1 \in F \iff q_2 \in F$

Note that for a GBA issued from Step 2, the condition $(q_1, \alpha, q') \in T_j$ does not depend on q_1 so that the condition simply becomes $\delta(q_1) = \delta(q_2)$.

This simplification procedure is really efficient to reduce the size of the automata. But the strength of our algorithm is that the last two simplification rules are also used *on-the-fly*: after a transition has been created, it is compared with the other transitions already calculated from the same state, and the ones that become useless are immediately deleted; after all the transitions of a state have been created, that state is compared with the other states that have already been created, and is merged to one of those states if possible. This method is important since usually many states and transitions are to be simplified, and simplifying them on-the-fly saves a lot of time and space.

In Table 1, the results of the algorithm with or without on-the-fly simplification are compared (LTL2BA– is our algorithm with a posteriori simplification only). For the formula θ_n defined in (1), the unsimplified GBA has 2^{n+1} states, whereas the simplified GBA has only 2 states. Using on-the-fly simplification avoids the intermediary exponential automaton which explains the great improvement, even if the time and memory used by LTL2BA are still exponential.

7 Experimental Results

In this section we compare the results of some recent algorithms transforming an LTL formula into a BA.

Spin is a model-checker developed by Bell Labs since 1980. It contains an algorithm transforming an LTL formula into a BA, presented in [3]. The program is written in C, and we used version 3.4.1. (released Aug 2000).

Wring is an algorithm presented in [10]. The program is written in Perl, so the comparison with our work cannot be read litterally, and the amount of memory used is just an approximation we made using the Unix command '`top`'.

EQLTL is an algorithm presented in [2]. The program is not publicly available, but a demo is proposed on the web. All we could do was to measure the time needed by the web interface to start responding to our request. We do not even know what type of machine handles the request. Consequently the times we gave should be taken with caution.

LTL2BA is a program writen in C as Spin, in order to make reliable comparison between the two programs. **LTL2BA–** is the same program, with a prosteriori simplification only.

Tests were made on a Sun Ultra 10 station with 1 GB of RAM.

As explained in the introduction, we compared the tools on usual LTL formulae, taking the example of the formula θ_n defined in (1). The result of the comparison is detailed in Table 1.

Another type of usual LTL formulae, often encountered in model-checking, is formulae like: $\varphi_n = \neg(p_1 \text{ U } (p_2 \text{ U } (\ldots \text{ U } p_n) \ldots)$. We made the same tests on these fomulae in Table 2. Again our algorithm outperforms the other ones.

	Spin		Wring		EQLTL	LTL2BA	
	time	space	time	space	time	time	space
φ_2	0.01	8	0.07	4,100	8	0.01	3.2
φ_3	0.03	110	0.29	4,100	8	0.01	5.5
φ_4	0.75	1,700	1.34	4,200	9	0.01	11
φ_5	43	51,000	10	4,200	11	0.01	13
φ_6	1,200	920,000	92	4,500	15	0.15	25
φ_7			720	6,000	27	9.2	48
φ_8					92	1,200	93

Table 2. Comparison on the formulae φ_n for $2 \le n \le 8$. Time is in sec, space in kB

We also compared the algorithms on random LTL formulae of a fixed size, using a tool presented in [11]. For compatibility reasons, the only comparison we could realize was between our alogrithm and Spin's. Here the results are issued from a test on 200 random formulae of size 10, where both algorithms are compared on the *same* formulae. See Table 3 for details.

	Spin		LTL2BA	
	avg.	max.	avg.	max.
time of computation (seconds)	14.23	4521.65	0.01	0.04
number of states	5.74	56	4.51	39
number of transitions	14.73	223	9.67	112

Table 3. Comparison on random formulae of a fixed size

References

1. M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In *Proc. 11th International Computer Aided Verification Conference*, pages 249–260, 1999.
2. K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Proceedings of 11th Int. Conf. on Concurrency Theory (CONCUR)*, 2000.
3. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
4. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
5. O. Kupferman and M. Vardi. Weak alternating automata are not that weak. In *Proc. 5th Israeli Symposium on Theory of Computing and Systems ISTCS'97*, pages 147–158. IEEE, 1997.
6. D. Muller and P. Schupp. Alternating automata on infinite objects: Determinacy and Rabin's theorem. In *Proceedings of the Ecole de Printemps d'Informatique Théoretique on Automata on Infinite Words*, volume 192 of *LNCS*, pages 100–107, Le Mont Dore, France, May 1984. Springer.
7. D. Muller and P. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54(2-3):267–276, October 1987.
8. D. Muller and P. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141(1–2):69–107, April 1995.
9. S. Rohde. Alternating automata and the temporal logic of ordinals. *PhD Thesis in Mathematics, University of Illinois at Urbana-Champaign*, 1997.
10. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV: International Conference on Computer Aided Verification*, 2000.
11. H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In *Workshop Concurrency, Specifications and Programming*, pages 251–262, Warsaw, Poland, 1999.
12. M. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag Inc., New York, NY, USA, 1996.