Vectorial Languages and Linear Temporal Logic

Olivier Serre

Vectorial Languages and Linear Temporal Logic

Olivier Serre

LIAFA, Université Paris VII
2, place Jussieu, case 7014
F-75251 Paris Cedex 05

Abstract

Determining for a given deterministic complete automaton the sequence of visited states while reading a given word is the core of important problems with automata-based solutions, such as approximate string matching. The main difficulty is to do this computation efficiently. Considering words as vectors and working on them using vectorial operations allows to solve the problem faster than using local operations.

In this paper, we show first that the set of vectorial operations needed by an algorithm representing a given automaton depends on the language accepted by the automaton. We give precise characterizations for star-free, solvable and regular languages using vectorial algorithms. We also study classes of languages associated with restricted sets of vectorial operations and relate them with languages defined by fragments of linear temporal logic.

Finally, we consider the converse problem of constructing an automaton from a given vectorial algorithm. As a byproduct, we show that the satisfiability problem for some extensions of LTL characterizing solvable and regular languages is PSPACE-complete.

1 Introduction

Given a deterministic complete automaton and an input word, a classical question is to decide whether or not the automaton accepts the word. A more detailed information is the sequence of visited states while processing the word. Computing this sequence is the core of important problems such as approximate string matching. An easy way to solve this problem consists in simulating the run of the automaton (which is deterministic and complete) on the input word. However, approximate string matching is generally used on very long sequences (as genomic ones) and the natural algorithm, which is

linear in the length of the input word, is not performing enough. A natural solution to accelerate the computation is to consider words as vectors and therefore to compute the sequence of visited states using vectorial operations, that can be efficiently achieved using parallelism.

In this paper, we are interested in vectorial algorithms, that were introduced and investigated by A. Bergeron and S. Hamel in [3,4]. Such an algorithm computes the sequence of visited states while reading a word using a *finite* number (independent of the length of the word) of vectorial operations. The existence of an algorithm for a given automaton depends on the automaton and on the kind of vectorial operations we allow. The problem can also be studied from the language point of view: can we find a deterministic complete automaton recognizing a given language and an associated vectorial algorithm? We first exhibit a very tight connection between temporal logic operators and some vectorial operations, that will therefore be called PTL-vectorial operations. This leads to an alternative proof of the equivalence between star-free languages and vectorial algorithms, whose direct inclusion was first established in [4]. Then, we describe extensions of these algorithms, first to capture a larger subclass of regular languages, the solvable ones, and finally for the whole class of regular languages.

In the second part of the paper, we investigate fragments of algorithms based on the set of PTL-vectorial operations. Here, we want to know which subset of star-free languages can be characterized by forbidding certain vector operations. We show that these fragments are closely related with the fragments of past temporal logic defined and characterized in [5,16,17].

Finally, we consider the converse problem, that is we want to check for a given vectorial algorithm whether there exists an automaton associated with it. To solve this problem, we show how to decide the satisfiability of formulas belonging to extensions of linear temporal logic introduced in [2]. Our constructions are based on alternating automata.

2 Notations and definitions

Throughout the paper, vectors are noted in bold characters (e.g. $\mathbf{u}$) and are considered as words. Conversely, vectorial operations can be applied to words, considering them as vectors. Therefore, a word u is associated with a canonical vectorial representation $\mathbf{u}$ and a vector $\mathbf{v}$ is associated with a canonical word representation v.

Let $\mathcal{A} = (Q, A, \cdot, q_0, F)$ be a deterministic complete automaton. With each input vector $\mathbf{u} = a_1 a_2 \cdots a_m$ we associate the output vector $\mathbf{r} = r_1 r_2 \cdots r_m$

representing the sequence of states reached reading $\mathbf{u}$ (we omit the leading initial state). Therefore, $\mathbf{u}$ and $\mathbf{r}$ have the same length. For instance, consider the automaton given in Example 1. With the input vector $\mathbf{u} = bbaabbbababab$ we associate the output vector $\mathbf{r} = 2311233121212$. A *vectorial algorithm* for $\mathcal{A}$ consists of a sequence of vectorial operations of fixed length (i.e., a straight-line expression of length which is independent on $\mathbf{u}$) computing $\mathbf{r}$ from $\mathbf{u}$.

Given a word $\mathbf{u} = a_1 a_2 \cdots a_m$, we consider for every letter $a \in A$, the boolean vector $(\mathbf{u} = \mathbf{a}) = (a_1 = a) \cdots (a_m = a)$, where $(a_1 = a)$ is a boolean whereas the equality sign after $(\mathbf{u} = \mathbf{a})$ represents an assignment. Hence, $(\mathbf{u} = \mathbf{a})$ is the characteristic boolean vector of the letter a in the word $\mathbf{u}$. Just as for words, for any state $q \in Q$, with an output vector $\mathbf{r} = r_1 r_2 \cdots r_m$ we associate the boolean vector $(\mathbf{r} = \mathbf{q}) = (r_1 = q) \cdots (r_m = q)$ that is, the characteristic vector of state q. For example, $(bbaabbbababab = \mathbf{a}) = 0011000101010$ and $(2311233121212 = \mathbf{1}) = 0011000101010$.

The sequence $(\mathbf{u} = \mathbf{a})_{a \in A}$ (respectively, the sequence $(\mathbf{r} = \mathbf{q})_{q \in Q}$) is an equivalent boolean representation for the input word $\mathbf{u}$ (respectively for the output vector $\mathbf{r}$). In fact, in order to work only on boolean vectors, the vectorial algorithms presented in this paper compute the sequence of characteristic vectors $(\mathbf{r} = \mathbf{q})_{q \in Q}$ from the sequence of characteristic vectors $(\mathbf{u} = \mathbf{a})_{a \in A}$.

Let Ω be a class of vectorial operations. Vectorial algorithms based on this set of operations and on the bit-wise logical operations (combinations of $\vee$, $\wedge$ and $\neg$) are called Ω-*vectorial algorithms*. A deterministic complete automaton is called Ω-*vectorial* if there is an Ω-vectorial algorithm computing the sequence $(\mathbf{r} = \mathbf{q})_{q \in Q}$ from the sequence $(\mathbf{u} = \mathbf{a})_{a \in A}$. Finally, a language is Ω-vectorial if it is recognized by a deterministic complete Ω-vectorial automaton. Proposition 1 below shows that minimization preserves the property of being an Ω-vectorial automaton. Therefore a language is Ω-vectorial if and only if its minimal automaton is Ω-vectorial (by minimal automaton we always mean the minimal *complete* automaton). This property is very useful, because to decide whether or not a language is Ω-vectorial, it suffices to know how to decide whether or not a given automaton (the minimal one) is Ω-vectorial.

Proposition 1 *Let $\mathcal{A}$ be a deterministic complete automaton. If $\mathcal{A}$ is Ω-vectorial, then its minimal automaton $\mathcal{A}_{min}$ is also Ω-vectorial.*

Proof:

As $\mathcal{A}$ is deterministic and complete, the states of $\mathcal{A}_{min}$ correspond to the Nerode equivalence classes of states of $\mathcal{A}$. An expression characterizing a state of the minimal automaton is obtained as the disjunction of the expressions for the states of the associated equivalence class.

■

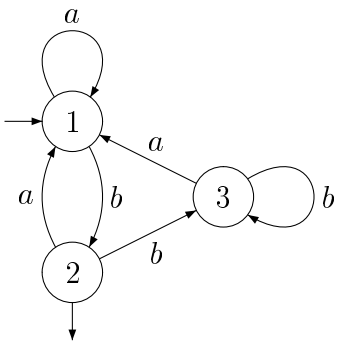
To make our algorithms precise we have to state which vectorial operations are allowed. As in [4], we first consider a basic set of vectorial operations:

- Bit-wise logical operations such as $\vee$, $\wedge$, $\neg$ and the atomic formulas $(\mathbf{u} = \mathbf{a}) = (u_1 = a) \cdots (u_m = a)$.
- Right shift: $\uparrow_i u_1 \cdots u_m = i u_1 \cdots u_{m-1}$, $i \in \{0, 1\}$.
- Binary addition between two vectors of same length: we perform the usual binary addition from left to right but we do not keep the highest bit (carry) if the length of the result exceeds the initial vectors' ones. For example $\uparrow_0 110101 + 101011 = 110100$.

vectorial algorithms using only these operations will be called in this paper *PTL-vectorial algorithms*. PTL stands for Past Temporal Logic and we will show that there exists a close relation between vectorial operations of this kind and past temporal logic operators. A language is therefore a *PTL-vectorial language* if its minimal automaton is PTL-vectorial.

Example 1 [3]

The language recognized by the following automaton is a PTL-vectorial language.



Actually, the state to which a word leads depends only on the last two letters of the word: if the last letter is a the state is 1, if the length 2 suffix is ab or if we are working on the first letter of the word and this letter is b (in this case we cannot consider the length 2 suffix) the state is 2 and otherwise it is 3. Therefore, we have the following PTL-vectorial algorithm for this automaton:

$$r = \begin{cases} (\mathbf{r} = \mathbf{1}) = (\mathbf{u} = \mathbf{a}) \\ (\mathbf{r} = \mathbf{2}) = (\mathbf{u} = \mathbf{b}) \wedge \uparrow_1 (\mathbf{u} = \mathbf{a}) \\ (\mathbf{r} = \mathbf{3}) = (\mathbf{u} = \mathbf{b}) \wedge \uparrow_0 (\mathbf{u} = \mathbf{b}) \end{cases}$$

3 Past temporal logic

The main result of the next section states that a language is a PTL-vectorial language if and only if it is star-free. Recall that a language is star-free if and only if its syntactic monoid is aperiodic [11]. Furthermore a language is star-free if and only if it can be defined by first-order logic of the linear order FO[<] [6,7]. Our proof relies a third characterization in terms of past temporal logic (PTL). We first recall the syntax of PTL:

We use atomic propositions p_a for each letter $a \in A$ of a given alphabet A, boolean connectives ($\vee$ and $\neg$) and past temporal operators (Yesterday, denoted $\mathbf{Y}$, and Since, denoted $\mathbf{S}$).

The formulas are constructed inductively according to the following rules:

(1) For every $a \in A$, p_a is a formula.
(2) If φ_1 and φ_2 are formulas, so are $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$, $\mathbf{Y}\varphi_1$ and $\varphi_1 \,\mathbf{S}\, \varphi_2$.

Semantics is defined by induction on the rules. Given a word $w \in A^+$ and an integer $n \in \{1, 2, \ldots, |w|\}$, we define that "$w$ satisfies φ at position n", denoted $(w, n) \models \varphi$, as follows:

(1) $(w, n) \models p_a$ if the nth letter of w is a.
(2) $(w, n) \models \varphi_1 \vee \varphi_2$ if $(w, n) \models \varphi_1$ or $(w, n) \models \varphi_2$.
(3) $(w, n) \models \neg\varphi_1$ if $(w, n) \not\models \varphi_1$.
(4) $(w, n) \models \mathbf{Y}\varphi_1$ if $n > 1$ and $(w, n-1) \models \varphi_1$.
(5) $(w, n) \models \varphi_1 \,\mathbf{S}\, \varphi_2$ if there exists $m \leq n$ such that $(w, m) \models \varphi_2$ and, for every k such that $m < k \leq n$, $(w, k) \models \varphi_1$.

With each PTL-formula φ, we associate the language of finite words satisfying φ:

$$L_\varphi = \{u \in A^+ \mid (u, |u|) \models \varphi\}$$

Recall that a regular language L is star-free if and only if there exists a PTL-formula φ such that $L = L_\varphi$ (see for instance [5]), i.e. if and only if L is PTL-definable.

4 PTL-vectorial languages are equivalent to star-free languages

Our aim in this section is to prove the following characterization:

Theorem 1 *A regular language is star-free if and only if it is PTL-vectorial.*

The equivalence between PTL-definable languages and star-free languages implies that Theorem 1 is equivalent with the following result:

Theorem 2 *A regular language is PTL-vectorial if and only if it is PTL-definable.*

The proof of this result is splitted into two parts: passing from PTL-formulas to PTL-vectorial algorithms and from PTL-vectorial algorithms to PTL-formulas. The first part is done in Section 4.1 and the second one in Section 4.2.

4.1 Star-free languages are PTL-vectorial

Let L be a star-free language and let $\mathcal{A} = (Q, A, \cdot, q_0, F)$ be its minimal automaton. Since L is star-free, $\mathcal{A}$ is counter-free (note that this property is independent from the final states). Let q be a state of $\mathcal{A}$, then the language $L_q = \{u \mid (u, |u|) \models \varphi\}$ is recognized by an automaton obtained from $\mathcal{A}$ by letting q be the unique final state. Therefore, L_q is star-free (because it is recognized by a counter-free automaton) and thus PTL-definable: there exists a PTL-formula φ_q such that $q_0 \cdot u = q$ if and only if $(u, |u|) \models \varphi_q$.

We will show that for any input vector $\mathbf{u} = a_1 \cdots a_m$ and any PTL-formula φ, the computation of the binary vector $\mathbf{v} = v_1 \cdots v_m$, where $v_i = 1$ if and only if $(e_1 \cdots e_i, i) \models \varphi$, can be performed by a PTL-vectorial algorithm Φ.

The definition of Φ is given by induction on φ:

(1) If $\varphi = p_a$ then $\Phi = (\mathbf{u} = \mathbf{a})$.
(2) If $\varphi = \varphi_1 \vee \varphi_2$ then $\Phi = \Phi_1 \vee \Phi_2$, where Φ_1 and Φ_2 are associated respectively with φ_1 and φ_2.
(3) If $\varphi = \neg\varphi_1$ then $\Phi = \neg\Phi_1$, where Φ_1 is associated with φ_1.
(4) If $\varphi = \mathbf{Y}\varphi_1$ then $\Phi = \uparrow_0 \Phi_1$, where Φ_1 is associated with φ_1.
(5) If $\varphi = \varphi_1 \mathbf{S} \varphi_2$, then $\Phi = \Phi_2 \vee [(\uparrow_0 \Phi_2) \wedge \Phi_1] \vee [\Phi_1 \wedge \mathrm{carry}(\Phi_1, (\uparrow_0 \Phi_2) \wedge \Phi_1)]$, where Φ_1 and Φ_2 are associated respectively with φ_1 and φ_2.

The formula $\mathrm{carry}(\mathbf{v}, \mathbf{w})$ is an abbreviation for: $\neg[(\mathbf{v} \oplus \mathbf{w}) = (\mathbf{v} + \mathbf{w})]$, where $\oplus$ is the exclusive-or operator. Therefore, $\mathrm{carry}(\mathbf{v}, \mathbf{w})$ represents the value of the carry bit when adding $\mathbf{v}$ and $\mathbf{w}$. For instance, $\mathrm{carry}(101101, 100110) = 010011$.

Let us justify the construction for the since operator, (5). A word u is such that $(u, n) \models \varphi_1 \mathbf{S} \varphi_2$ in one of the three cases:

(i) $(u, n) \models \varphi_2$. This case is treated by the algorithm Φ_2.

(ii) $(u, n-1) \models \varphi_2$ and $(u, n) \models \varphi_1$. This case is treated by the algorithm $[(\uparrow_0 \Phi_2) \wedge \Phi_1]$.

(iii) There is an integer $m < n-1$ (which is chosen maximal) such that $(u, m) \models \varphi_2$ and, for all k such that $m < k \leq n$, $(u, k) \models \varphi_1$. The vector $\mathrm{carry}(\Phi_1, (\uparrow_0 \Phi_2) \wedge \Phi_1]$ is such that its n-th position is 1 if and only if there exists $m < n-1$ such that $(u, m) \models \varphi_2$ and, for all k such that $m < k \leq n-1$, $(u, k) \models \varphi_1$. Consequently, the algorithm $\Phi_1 \wedge \mathrm{carry}(\Phi_1, (\uparrow_0 \Phi_2) \wedge \Phi_1)$ exactly characterizes our case. Note that the first component Φ_1 is necessary because the carry has a non immediate effect: it affects the first position after it is generated. For the same reason, the second case had to be treated separately.

This inductive construction concludes the proof that any star-free language is PTL-vectorial.

The following example illustrates the preceding cases for the value $\Phi(\mathbf{u})$ of Φ applied to a vector $\mathbf{u}$:

$\Phi_2(\mathbf{u})$	=	0	0	1	0	0	1	0	0
$\uparrow_0 [\Phi_2(\mathbf{u})]$	=	0	0	0	1	0	0	1	0
$\Phi_1(\mathbf{u})$	=	1	1	0	1	1	1	0	1
$[(\uparrow_0 \Phi_2) \wedge \Phi_1](\mathbf{u})$	=	0	0	0	1	0	0	0	0
$[carry(\Phi_1, (\uparrow_0 \Phi_2) \wedge \Phi_1)](\mathbf{u})$	=	0	0	0	0	1	1	1	0
$[\Phi_1 \wedge carry(\Phi_1, (\uparrow_0 \Phi_2) \wedge \Phi_1)](\mathbf{u})$	=	0	0	0	0	1	1	0	0
$\Phi(\mathbf{u})$	=	0	0	1	1	1	1	0	0

4.2 PTL-vectorial languages are star-free

Let L be a PTL-vectorial language. By Proposition 1, its minimal automaton $\mathcal{A} = (Q, A, \cdot, q_0, F)$ is a PTL-vectorial automaton. To prove that L is star-free (equivalently, PTL-definable) we construct for each state q of $\mathcal{A}$, a PTL-formula φ_q such that for any word $u \in A^+$, $q_0 \cdot u = q$ if and only if $(u, |u|) \models \varphi_q$. Therefore, L will be characterized by the disjunction $\bigvee_{q \in F} \varphi_q$, which is a PTL-formula.

For each state q of $\mathcal{A}$ we have a PTL-vectorial algorithm Φ_q computing, from each input vector $\mathbf{u}$, the characteristic output vector of state q, $(\mathbf{r} = \mathbf{q})$. The formula φ_q which "translates" Φ_q is defined by induction on the structure of Φ_q. Moreover, the logical formula φ as defined below for a vectorial algorithm

Φ, satisfies the following property: the i-th entry of the vector obtained by applying Φ to $\mathbf{u}$ is 1 if and only if $(u, i) \models \varphi$.

(1) If $\Phi = (\mathbf{u} = \mathbf{a})$, where a is a letter, then $\varphi = p_a$.
(2) If $\Phi = \Phi_1 \vee \Phi_2$ then $\varphi = \varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are associated respectively with Φ_1 and Φ_2.
(3) If $\Phi = \neg\Phi_1$ then $\varphi = \neg\varphi_1$, where φ_1 is associated with Φ_1.
(4) If $\Phi = \uparrow_0 \Phi_1$ then $\varphi = \mathbf{Y}\varphi_1$, where φ_1 is associated with Φ_1. The case $\Phi = \uparrow_1 \Phi_1$ follows from the preceding one, because $\uparrow_1 \Phi_1 = \neg(\uparrow_0 \neg\Phi_1)$.
(5) If $\Phi = \Phi_1 + \Phi_2$, we let

$$c = \mathbf{Y}\{[\varphi_1 \vee \varphi_2] \ \mathbf{S} \ [\varphi_1 \wedge \varphi_2]\}$$

where φ_1 and φ_2 are associated respectively with Φ_1 and Φ_2. Note that the formula c characterizes the value of the carry bit while summing the vectors obtained from Φ_1 and Φ_2. Thus, the translation of $\Phi_1 + \Phi_2$ is the logical formula:

$$\varphi = [\varphi_1 \oplus \varphi_2] \longleftrightarrow \neg c$$

Finally, the global formula for L is:

$$\varphi_{\mathcal{A}} = \bigvee_{q \in F} \varphi_q,$$

and for each word u, we have $u \in L = L(\mathcal{A})$ if and only if $(u, |u|) \models \varphi_{\mathcal{A}}$.

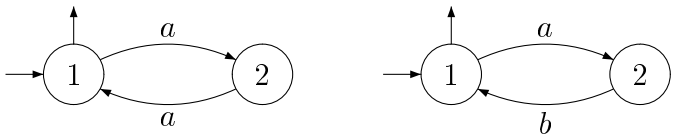
This concludes the proof.

■

5 Extensions of PTL-vectorial algorithms

A natural question is to extend PTL-vectorial algorithms, in order to capture larger classes of regular languages by parallel operations. To achieve this goal we need to introduce new operations that are strictly more powerful than the PTL operations. In a first extension we will characterize solvable regular languages (a regular language is called solvable if its syntactic monoid does not contain any non solvable group) and in a second extension all regular languages.

The crucial point in defining extensions of PTL-vectorial algorithms is the choice of the new operations allowed. To determine them, let us give another proof of the fact that PTL-vectorial languages are star-free. A well known example of non star-free language is $L_1 = (aa)^*$ whereas, on the alphabet $A = \{a, b\}$, $L_2 = (ab)^* = A^* \setminus [bA^* + A^*a + A^*(aa + bb)A^*$ is a star-free one. Is there a "vectorial" difference between them? L_1 and L_2 are recognized by the following automata:



$\mathcal{A}_1$ $\mathcal{A}_2$

Recall that a period of a word $u = a_1 a_2 \cdots a_n$ is an integer $p \leq n/2$ such that for any position i, $1 \leq i \leq n - p$, $a_{i+p} = a_i$. A word having period p is said to be of periodicity p. For example *abbaabbaabbaab* is of periodicity 4. Finally, a word is ultimately periodic of period p if it has a suffix of periodicity p. For instance, *abbbabababababab* is ultimately periodic of period 2.

Words recognized by $\mathcal{A}_1$ are of periodicity 1 whereas the associated sequence of states is of periodicity 2. This is not the case for the automaton $\mathcal{A}_2$. One can show that any PTL-vectorial algorithm applied to an ultimately periodic word gives an ultimately periodic result of the *same* period (PTL-vectorial operations preserve periods). With any non counter-free automaton one can associate an ultimately periodic word of period p such that the associated sequence of states is not of ultimate period p. Thus, a PTL-vectorial language has to be star-free.

To extend PTL-vectorial algorithms, we need to introduce operators that do not preserve the period. For every integers k, l such that $0 \leq l < k$, the modular operator $S_{l,k}$ is defined by

$$S_{l,k}(x_1 x_2 \cdots x_m) = (s_1 \cdots s_m) \text{ where } s_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{i} x_j = l \pmod{k}, \\ 0 & \text{otherwise.} \end{cases}$$

vectorial algorithms using only the PTL-vectorial operations plus the modular operations $S_{l,k}$ will be called *MTL-vectorial algorithms*. A language is an *MTL-*

vectorial language if there is an MTL-vectorial algorithm for a deterministic complete automaton which recognizes it.

With modular operators, one can easily define MTL-vectorial algorithms for the minimal automaton of languages of the form $L(a, k, p^n) = \{u \in A^* |\ |u|_a \equiv k \pmod{p^n}\}$, where a is a letter, $|u|_a$ is the number of occurrences of $a \in A$ in u, p a prime number and k, n strictly positive numbers.

Since MTL-vectorial languages are a boolean algebra (just consider the product automaton to compute an algorithm for intersection or union of two MTL-vectorial languages) and languages of the form $K(a, r) = \{u \in A^* |\ |u|_a = r\}$ are PTL-vectorial (since star-free), the boolean algebra $\mathcal{M}_{Com}$ generated by languages $K(a, r)$ and $L(a, k, p^n)$, which is the set of languages whose syntactic monoid is commutative [8], is a subset of MTL-vectorial languages.

For characterizing the family of solvable languages we need a further operation on automata, the *cascade product*. Let A be a finite alphabet and let $\mathcal{B}_1 = (Q_1, A, \delta_1)$ and $\mathcal{B}_2 = (Q_2, Q_1 \times A, \delta_2)$ be two finite automata. Their cascade product $\mathcal{C} = (Q, A, \delta)$, denoted $\mathcal{B}_1 \circ \mathcal{B}_2$ is defined as follows:

- $Q = Q_1 \times Q_2$
- $\delta(\langle q_1, q_2 \rangle, a) = \langle \delta_1(q_1, a), \delta_2(q_2, \langle q_1, a \rangle) \rangle$

We can also define the cascade product of more than two automata by the following recursive formula:

$$\mathcal{B}_1 \circ \mathcal{B}_2 \circ \cdots \circ \mathcal{B}_k = (\ldots((\mathcal{B}_1 \circ \mathcal{B}_2) \circ \mathcal{B}_3) \circ \ldots) \circ \mathcal{B}_k$$

This automata version of the wreath product [9] of aperiodic semigroups, combined with the Krohn-Rhodes Decomposition Theorem [14], was used to construct PTL-vectorial algorithms from counter-free automata [3]. It is not difficult to prove that a family of automata corresponding to a class of vectorial algorithms, for instance counter-free automata or automata recognizing MTL-vectorial languages, is closed under cascade product and under homomorphisms. Therefore, as all star-free languages are MTL-vectorial languages, any language recognized by an automaton obtained by applying a homomorphism to a cascade product of counter-free automata and automata recognizing languages in $\mathcal{M}_{Com}$ is an MTL-vectorial language. In addition, any solvable language is recognized by an automaton obtained by applying an homomorphism to a cascade product of counter-free automata and automata recognizing languages in $\mathcal{M}_{Com}$ [13], and therefore this proves that solvable languages are MTL-vectorial languages.

In fact, the converse is true as well, any MTL-vectorial language is a solvable language. We have thus the following result:

Theorem 3 *A regular language is solvable if and only if it is MTL-vectorial.*

For the proof, we use an extension of the past temporal logic introduced in [2], the modular temporal logic (MTL). By modular temporal logic we mean past temporal logic augmented with the unary operators $\mathrm{Mod}_{l,k}$ for integers $0 \leq l < k$. The new modular operators have the following natural semantics: given an MTL formula φ, we have $(u,i) \models \mathrm{Mod}_{l,k}\varphi$ if, there are l positions i', $1 \leq i' \leq i$ (modulo k) such that $(u,i') \models \varphi$.

It was shown in [2] that a language is expressible in modular temporal logic if and only if it is solvable. But, with an MTL-vectorial language one can associate an MTL formula in a straightforward way. This shows that MTL vectorial languages are solvable languages and achieves the proof of the equivalence between MTL-vectorial languages and solvable languages. Moreover, as solvability is a syntactic property, the property of being an MTL-vectorial language does not depend actually on a specific automaton recognizing the language.

5.2 A vectorial characterization of regular languages

MTL-vectorial algorithms do not characterize all regular languages. Therefore, we propose an extension to MTL-vectorial algorithms, which we denote as *GTL-vectorial algorithms*, which captures all regular languages.

We will use again an extension of modular temporal logic introduced in [2]. This extension of temporal logic is obtained by augmenting modular temporal logic with group temporal operators $\Gamma_{g,G}$ for any finite group G and any element $g \in G$. The operator $\Gamma_{g,G}$ always binds $|G| - 1$ formulas.

Let us now explain the semantics of $\Gamma_{g,G}$ for a given finite group G and an element $g \in G$. We first have to order the elements of the group G (this order will not be modified afterward), say as $g_1, g_2, \ldots, g_q = id$. Let u be an element of A^+ and let $\varphi_1, \varphi_2, \ldots, \varphi_{q-1}$ be GTL-formulas. With each j, $1 \leq j \leq |u|$ we associate an element of G, denoted $\langle \varphi_1, \varphi_2 \ldots \varphi_{q-1} \rangle \langle u, j \rangle$, defined by:

$$\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle \langle u, j \rangle = g_k$$

where $k = \min\{l \mid (u,j) \models \varphi_l\}$ with the convention that $\min \emptyset = q$.

Finally, we define $(u,i) \models \Gamma_{g,G}\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle$ to mean that:

$$\prod_{j=1}^{i} \langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle \langle u, j \rangle = g$$

It was shown in [2] that a language is expressible in group temporal logic if and only if it is regular.

Thus, to have a vectorial characterization of all regular languages it suffices to find vectorial operations equivalent to $\Gamma_{g,G}$ for all finite groups G. Let G be a finite group of cardinality q. We consider an isomorphic copy H of G defined as follows. The elements of H are boolean vectors of length q containing exactly one 1. Therefore, $(1,0,\ldots,0)$ is associated with g_1, $(0,1,0,\ldots,0)$ with g_2 and so on. The product $\times$ on H is defined by the isomorphism σ between G and H. For each group G and each element $g \in G$, we introduce the operator $P_{g,G}$ defined by:

$$P_{g,G}(\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_q}) = (s_1 \ldots s_m) \text{ with } s_i = \begin{cases} 1 & \text{if } \bigotimes_{j=1}^{i}(v_{1,j}, v_{2,j}, \ldots, v_{q,j}) = \sigma(g), \\ 0 & \text{otherwise.} \end{cases}$$

where we mean by $\bigotimes$ an iteration of $\times$ and by $v_{k,j}$ the j-th bit of vector $\mathbf{v_k}$.

Remark 1 In our definition of $P_{g,G}$, we implicitly suppose that for each j, $1 \leq j \leq i$, there is exactly one vector $\mathbf{v_k}$ such that $v_{k,i} = 1$. If this is not the case, we have a problem in defining the product $\bigotimes$ because some $(v_{1,j}, v_{2,j}, \ldots, v_{q,j})$ do not belong to H. The solution consists in defining the product $\times$, and thus the iterated product $\bigotimes$, for all boolean vectors of length q. To achieve this we define an equivalence relation $\sim$ for boolean vectors saying that $\mathbf{x} \sim \mathbf{y}$ if and only if $\mathbf{x}$ and $\mathbf{y}$ have their first 1 in the same position. In addition, the vector $(0,0,\ldots,0)$ is equivalent to the neutral element $(0,0,\ldots,0,1)$. Finally we define $\times$ for all vectors using the equivalence relation $\sim$.

The equivalence relation $\sim$ works like the operator $\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle$, it just looks for the first 1 (validate formula) that appears on the vector.

Therefore with $\varphi = \Gamma_{g,G}(\varphi_1, \ldots, \varphi_{q-1})$ we associate Φ:

$$\Phi = P_{g,G}(\Phi_1, \Phi_2, \Phi_3, \ldots, \Phi_{q-1}, \neg\Phi_1 \wedge \cdots \wedge \neg\Phi_{q-1}),$$

where Φ_i is the translation of φ_i. Denoting by $\Phi_i^j(\mathbf{u})$ the j-th bit of the result of the algorithm Φ_i applied to $\mathbf{u}$ we then have:

$$\begin{aligned} &\sigma(\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle \langle u, j \rangle) \sim \\ &(\Phi_1^j(\mathbf{u}), \Phi_2^j(\mathbf{u}), \Phi_3^j(\mathbf{u}), \ldots, \Phi_{q-1}^j(\mathbf{u}), \neg\Phi_1^j(\mathbf{u}) \wedge \cdots \wedge \neg\Phi_{q-1}^j(\mathbf{u})) \end{aligned}$$

and therefore we have $(u, i) \models \varphi$ if and only if the i-th bit of the result of Φ applied to $\mathbf{u}$ is 1.

Conversely, with $\Phi = P_{g,G}(\Phi_1, \ldots, \Phi_q)$ we associate:

$$\varphi = \Gamma_{g,G}[\varphi_1, \varphi_2, \varphi_3, \ldots, \varphi_{q-1}]$$

where φ_i is the translation of Φ_i.Denoting again by $\Phi_i^j(\mathbf{u})$ the j-th bit of the result of Φ_i applied to $\mathbf{u}$ we obtain:

$$\Phi_1^j(\mathbf{u}), \Phi_2^j(\mathbf{u}), \Phi_3^j(\mathbf{u}), \ldots, \Phi_{q-1}^j(\mathbf{u}), \neg\Phi_1^j(\mathbf{u}) \wedge \cdots \wedge \neg\Phi_{q-1}^j(\mathbf{u})) \sim$$
$$\sigma(\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle\langle u, j\rangle)$$

and therefore the i-th bit of the result of Φ applied to $\mathbf{u}$ is 1 if and only if $(u, i) \models \varphi$.

Thus, the operators $\Gamma_{g,G}$ and $P_{g,G}$ have the same expressive power. We can now characterize all regular languages by vectorial algorithms. Vectorial algorithms using only the MTL-vectorial operations plus the group operations $P_{g,G}$ will be called *GTL-vectorial algorithms*. A language is a *GTL-vectorial language* if there is a GTL-vectorial algorithm for a deterministic complete automaton which recognizes it.

Combining the preceding results, we have the following characterization of regular languages:

Theorem 4 *A language is regular if and only if it is GTL-vectorial.*

Effectively, a GTL-vectorial language is regular (because it is recognized by a finite automaton). Conversely, to any regular language one can associate a finite deterministic complete recognizing it. To any state of the automaton, one can associate a regular language (and thus a GTL-formula) representing the set of words that lead to this state in the automaton. Translating these GTL-formulas gives us a GTL-vectorial algorithm for the automaton. Hence any regular language is a GTL-vectorial language. Regularity being a syntactic property, the property of being a GTL-vectorial language does not depend actually on a specific automaton.

6 Fragments of PTL-vectorial languages

In the preceding sections we attempted to extend PTL-vectorial algorithms to characterize regular languages more complex than star-free ones. For that we needed new vectorial operators. However, the price to pay is that the extended vectorial operations are not obviously realizable from a hardware point of view.

A dual investigation is to study fragments of PTL-vectorial languages: given a set of vectorial operators (that can be efficiently performed) we want to determine which kind of (deterministic complete) automata can be characterized by algorithms using the given set of vectorial operations. A similar problem has been studied for temporal logic in [5,16,17] and we will relate it to our problem.

6.1 Definitions

We first introduce a new vectorial operation called *right*, denoted by $^{\rightarrow}$ and defined as: $^{\rightarrow}\mathbf{v} = \mathbf{v} \vee [\neg(\mathbf{v} + \mathbf{1})]$. It is easily seen that $^{\rightarrow}\mathbf{0} = \mathbf{0}$ and that $^{\rightarrow}0\cdots01?\cdots? = 0\cdots01\cdots1$. That is, right matches the first one (from the left) by completing the vector with ones (to the right) after the first one. We note that $^{\rightarrow}$ is the vectorial equivalent of the past temporal operator $\mathbf{P}$ (past) whose semantics is defined by: $(w,n) \models \mathbf{P}\varphi$ if there exists $m \leq n$ such that $(w,m) \models \varphi$.

We also define a strict version of the $^{\rightarrow}$ operation, the *strict right*, denoted by $^{\rightarrowtail}$ and defined by $^{\rightarrowtail}\mathbf{v} = \uparrow_0 (^{\rightarrow}\mathbf{v})$. Thus, it is easily seen that $^{\rightarrowtail}\mathbf{0} = \mathbf{0}$ and that $^{\rightarrowtail}\underbrace{0\cdots0}1?\cdots? = \underbrace{0\cdots0}01\cdots1$. For example, we have: $^{\rightarrowtail}0001101001 = 0000111111$ and $^{\rightarrowtail}001 = 000$. The $^{\rightarrowtail}$ operator is the vectorial equivalent of the strict version $\mathbf{YP}$ of the operator $\mathbf{P}$, defined by: $(w,n) \models \mathbf{YP}\varphi$ if there exists $m < n$ such that $(w,m) \models \varphi$.

Given a class Ω of vectorial operations, we write VA[Ω] for the set of vectorial algorithms using only bit-wise logical operations and operations in Ω. For convenience, we omit the braces: we write VA[$\rightarrow, +$] instead of VA[$\{\rightarrow, +\}$]. For example PTL-vector algorithms are exactly the ones in VA[$\uparrow_0, +$].

We use the same notation for languages: we will denote by V$\mathcal{L}$[Ω] the set of languages for which there is an automaton and a corresponding algorithm in VA[Ω]. Therefore V$\mathcal{L}$[$\uparrow_0, +$] describes the PTL-vectorial languages.

For each fragment of PTL-vectorial languages, we would like to have an efficient algorithm to decide whether or not a given language belongs to the fragment. For this, we will use characterizations of fragments of past temporal logic given in [5,16,17]. A fragment of past temporal logic is defined as follows: given a class Λ of temporal modalities, we write PTL[Λ] for the set of temporal formulas in which modalities other than ones from N do not occur. For convenience, we omit the braces, e.g., we write PTL[$\mathbf{Y}, \mathbf{YP}$] instead of PTL[$\{\mathbf{Y}, \mathbf{YP}\}$]. We can also associate with Λ a set of languages noted $\mathcal{L}[\Lambda]$ such that a language L belongs to $\mathcal{L}[\Lambda]$ if and only if there exists a formula $\varphi \in$ PTL[Λ] such that L is definable by φ.

6.2.1 Preliminaries

Intuitively, there exists a tight link between languages defined by logical conditions and languages defined by "equivalent" vectorial conditions. For example, in Section 4 we have seen that PTL-vectorial languages are the same as languages defined by past temporal logic. But, whereas logical satisfiability depends exclusively on the language, vectorial characterizations seem to be closely related with a specific automaton. Vectorial characterizations are stronger than logical characterizations because in order to have a vectorial algorithm for a given automaton one must be able to characterize *any* state, hence any language recognized by the automaton obtained by setting a given state as unique final state. For a logical formula one just needs to exhibit the set of final states needed for the given language.

But under some assumptions, logical fragments and vectorial fragments define the same class of languages. Let us be more explicit. Given a set Ω of vectorial operations and a set Λ of logical operators, we will say that Ω and Λ are *equivalent* if they verify the following conditions:

(1) To any vectorial algorithm Φ using only operations in Ω one can associate a PTL-formula φ using only operators in Λ such that for any word u and any positive integer i smaller than $|u|$, the i-th entry of the vector obtained by applying Φ to u is 1 if and only if $(u, i) \models \varphi$.
(2) To any PTL-formula φ using only operators in Λ, one can associate a vectorial algorithm Φ using only operators in Ω such that for any word $u = u_1 \cdots u_m$, the computation of the binary vector $\mathbf{v}_\varphi = v_1 \cdots v_m$, where $v_i = 1 \Leftrightarrow (u_1 \cdots u_i, i) \models \varphi$, is performed by the algorithm Φ.

For example, we have seen in Section 4 that the set of vectorial operations $\Omega = \{\uparrow_0, +\}$ is equivalent to the set of logical operators $\Lambda = \{\mathbf{Y}, \mathbf{S}\}$. Moreover, in this case we have that $\mathrm{V}\mathcal{L}[\Omega] = \mathcal{L}[\Lambda]$. Several fragments of temporal logic have been studied and characterized in [5,16,17] and therefore to characterize a fragment of PTL-vectorial languages, a solution consists in finding an equivalent fragment in temporal logic. We have to find a condition on two equivalent sets Ω and Λ to have $\mathrm{V}\mathcal{L}[\Omega] = \mathcal{L}[\Lambda]$.

A set Λ of logical operators will be called *finally stable* if for every language L that belongs to $\mathcal{L}(\Lambda)$, any language recognized by an automaton obtained from the minimal automaton of L by letting some arbitrary state to be the unique final state, belongs to $\mathcal{L}(\Lambda)$.

For example, any set Λ such that $\mathcal{L}(\Lambda)$ is a variety of languages is finally stable. Formally, if L is a language in $\mathcal{L}(\Lambda)$ and $\mathcal{A}$ its minimal automaton,

any automaton $\mathcal{A}'$ obtained by modifying the final states of $\mathcal{A}$ recognizes a language L' of $\mathcal{L}(\Lambda)$ because the syntactic monoid of $\mathcal{A}'$ divides the syntactic monoid of $\mathcal{A}$.

The notion of final stability gives us the following lemma:

Lemma 1 *Let Ω be a set of vectorial operations and let Λ be an equivalent set of logical operators. Then Λ is finally stable if and only if $\mathrm{V}\mathcal{L}(\Omega) = \mathcal{L}(\Lambda)$.*

Proof:

First assume that Λ is a finally stable set of logical operators equivalent to a set Ω of vectorial operations. The inclusion $\mathrm{V}\mathcal{L}(\Omega) \subseteq \mathcal{L}(\Lambda)$ is not difficult. To prove the converse inclusion, $\mathrm{V}\mathcal{L}(\Omega) \supseteq \mathcal{L}(\Lambda)$, let us consider a language $L \in \mathcal{L}(\Lambda)$ and its minimal automaton $\mathcal{A} = (Q, A, \cdot, q_i, F)$. For any state q of $\mathcal{A}$, the automaton $\mathcal{A}_q$ obtained from $\mathcal{A}$ by letting q be the unique final state, recognizes the language L_q that belongs to $\mathcal{L}(\Lambda)$. Therefore we have a formula φ_q in $\mathrm{PTL}(\Lambda)$ that defines L_q. Therefore, we have that $q_i \cdot u = q$ if and only if $(u, |u|) \models \varphi_q$. We obtain a simple algorithm in $\mathrm{VA}(\Omega)$ characterizing the state q in the automaton $\mathcal{A}$ just by translating φ_q (and this is possible by the equivalence between Ω and Λ).

Conversely, let us assume that Λ and Ω are equivalent and such that $\mathcal{L}(\Lambda) = \mathrm{V}\mathcal{L}(\Omega)$ and let us show that Λ is finally stable. For this, consider a language $L \in \mathcal{L}(\Lambda)$. Then L also belongs to $\mathrm{V}\mathcal{L}(\Omega)$, and therefore its minimal automaton $\mathcal{A}_{min}$ is Ω-vectorial. We have thus an algorithm for any state q of $\mathcal{A}_{min}$ (such that its translation into a PTL-formula belongs to $\mathrm{PTL}(\Lambda)$) that characterizes the language recognized by the automaton obtained from $\mathcal{A}_{min}$ by choosing q as unique final state. Therefore these languages belong to $\mathcal{L}(\Lambda)$, what proves the final stability of Λ and achieves the proof.

■

Remark 2 The preceding lemma and the results about temporal logic given in [2] yield a generic proof of the results of the preceding sections by noting that star-free languages, solvable languages and regular languages are varieties and that their associated sets of logical operators are finally stable.

In [5,16,17] several characterizations of fragments of past temporal logic are stated. We will use them to characterize fragments of PTL-vectorial languages. But first of all we need some definitions. The characterizations of fragments of past temporal logic use the minimal automaton and the presence, or absence, of specific structures, called *forbidden patterns*. For instance for star-free languages we consider a characterization that forbids counting patterns.

Given a set N, an N-labeled digraph is a tuple (V, E) where V is an arbitrary finite set and E a subset of $V \times N \times V$. The closure of a deterministic finite automaton $\mathcal{A}$, denoted $C_{\mathcal{A}}$, is the A^+-labeled digraph (V, E) where $E = \{(q, u, q \cdot u) \mid q \in Q \text{ and } u \in A^+\}$. Therefore, the closure of any deterministic finite automaton is an infinite graph (it has infinitely many edges, but only finitely many vertices).

Finally, a *pattern* is a labeled digraph whose vertices are state variables, usually denoted $p, q, \ldots$, and whose edges are labeled with variables for labels of two different types: variables for nonempty strings, usually denoted $u, v, \ldots$, and variables for letters, usually denoted $a, b, \ldots$. In addition, a pattern comes with side conditions stating which state variables are to be interpreted by distinct states. We draw patterns just as we draw graphs and adopt the convention that all states drawn solid must be distinct.

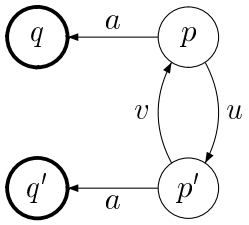
We say that an A^+-labeled digraph *matches a pattern* if there is an assignment to the variables obeying the type constrains and the side conditions, so that the digraph obtained by replacing each variable by the value assigned to it is an induced subgraph of the given digraph.

6.2.2 Characterizing $\mathrm{V}\mathcal{L}[\rightarrowtail]$

We are now ready to characterize our first fragment of PTL-vectorial languages:

Theorem 5 *Let L be a regular language over some alphabet A. Then the following assertions are equivalent:*

(1) *L belongs to $\mathrm{V}\mathcal{L}[\rightarrowtail]$.*
(2) *L belongs to $\mathcal{L}[\mathbf{YP}]$.*
(3) *The closure of the minimal automaton $\mathcal{A}_{min}(L)$ of L does not match the following pattern:*



The equivalence between (2) and (3) is shown in [16,17]. The other equivalences come from Lemma 1, from the equivalence between **YP** and $\rightarrowtail$, and from the following lemma (which implies that **YP** is finally stable):

Lemma 2 *Let us consider a deterministic complete automaton that does not*

match the pattern of Theorem 5. Then its minimal automaton does not match it either.

Proof:

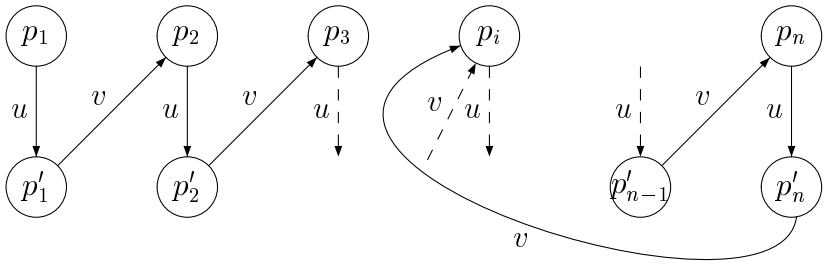
We show the result by contradiction. Let us consider a deterministic complete automaton $\mathcal{A}$ that does not match the pattern of Theorem 5 and let us assume that its minimal automaton $\mathcal{A}_{min}$ contains the pattern. Thus there exist four states P, Q, P' and Q' of $\mathcal{A}_{min}$, a letter a and two words u and v such that $P \cdot a = Q$, $P' \cdot a = Q'$, $P \cdot u = P'$, $P' \cdot v = P$ and $Q \neq Q'$. As $\mathcal{A}$ is a deterministic complete automaton, we can identify the states of $\mathcal{A}_{min}$ with the Nerode equivalence classes of $\mathcal{A}$. In the following we will not make any distinction between the states of $\mathcal{A}_{min}$ and the Nerode equivalence classes of $\mathcal{A}$. We have the following consequences:

(1) For any states $q \in Q$, $q' \in Q'$ we have $q \neq q'$.
(2) For any state $p \in P$, we have $p \cdot u \in P'$.
(3) For any state $p' \in P'$, we have $p' \cdot v \in P$.
(4) For any state $p \in P$ we have $p \cdot a \in Q$ and for any state $p' \in P'$ we have $p' \cdot a \in Q'$. Therefore, for any states $p \in P$,$p' \in P'$ we have $p \cdot a \neq p' \cdot a$.

Our aim is to prove the existence of two words z and t, of four states p, p', q and q' where $p \in P$, $p' \in P'$, $q \in Q$ and $q' \in Q'$ such that $p \cdot a = q$, $p' \cdot a = q'$, $p \cdot z = p'$ and $p' \cdot t = p$. Therefore we will have a contradiction with the fact that $\mathcal{A}$ does not contain the pattern.

Let us assume that $|P| \leq |P'|$ (the symmetric case is identical) and consider a state $p_1 \in P$. Thus the state $p'_1 = p_1 \cdot u$ belongs to P' and $p_2 = p'_1 \cdot v$ belongs to P. As $\mathcal{A}$ does not contain the pattern, we have $p_2 \neq p_1$. For the same reason the state $p'_2 = p_2 \cdot u$ belongs to P' and is different from p'_1, the state $p_3 = p'_2 \cdot v$ belongs to P and is different from p_2 and p_1 (because we have $p_1 \cdot uvu = p'_2$ and $p'_2 \cdot v = p_3$). Iterating this reasoning orders the states of $P = \{p_1, p_2, \ldots, p_n\}$ and the states of a subset $R = \{p'_1, p'_2, \ldots, p'_n\}$ of P'. Moreover, this order is such that for any $i \leq j \leq n$, there exists a word $\tilde{u}$ such that $p_i \cdot \tilde{u} = p'_j$ and for any $i < j \leq n$ there exists a word $\tilde{v}$ such that $p'_i \cdot \tilde{v} = p_j$. Let us now consider the state $p = p'_n \cdot v \in P$: there exists i, $1 \leq i \leq n$ such that $p = p_i$. We thus have a contradiction because there exists a non empty word z such that $p_i \cdot z = p'_n$ and $p'_n \cdot v = p_i$ and $p_i \cdot a \neq p'_n \cdot a$, hence $\mathcal{A}$ contains the pattern.

The proof can be resumed by the following diagram:



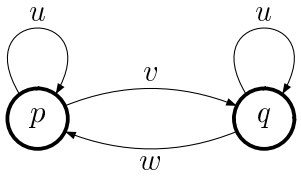
■

6.2.3 Characterizing $\mathrm{V}\mathcal{L}(\uparrow_0, \rightarrow)$

The class $\mathrm{V}\mathcal{L}(\uparrow_0, \rightarrow)$ corresponds to the logical fragment that uses only yesterday and past as operators:

Theorem 6 *Let L be a regular language over some alphabet A. Then the following assertions are equivalent:*

(1) *L belongs to $\mathrm{V}\mathcal{L}[\uparrow_0, \rightarrow]$.*
(2) *L belongs to $\mathcal{L}[\mathbf{Y}, \mathbf{P}]$.*
(3) *The closure of the minimal automaton $\mathcal{A}_{min}(L)$ of L does not match the following pattern:*



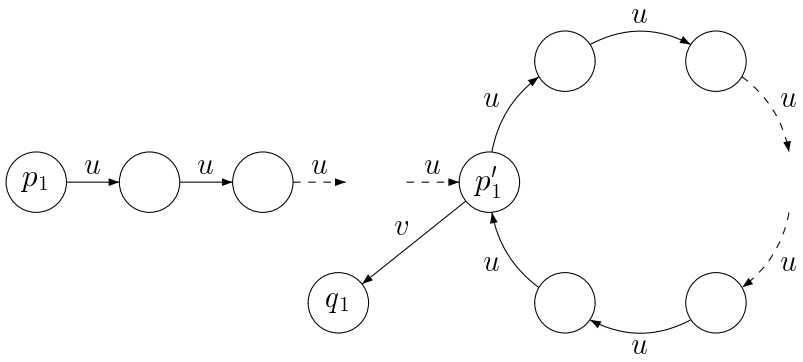
The equivalence between (2) and (3) can be found in [16,17]. The other equivalences come from Lemma 1 and from the following lemma, that implies that $\{\mathbf{Y}, \mathbf{P}\}$ is finally stable:

Lemma 3 *Consider a deterministic complete automaton that does not match the pattern of Theorem 6, then its minimal automaton does not match the pattern, either.*

As for the proof of Lemma 2 we reason by contradiction. Let us consider a deterministic automaton $\mathcal{A}$ that does not match the pattern and assume that its minimal automaton $\mathcal{A}_{min}$ matches the pattern. We thus have two distinct states P and Q of $\mathcal{A}_{min}$ and three words u, v and w such that: $P \cdot u = P$, $Q \cdot u = Q$, $P \cdot v = Q$ and $Q \cdot w = P$. As in the preceding proofs, we identify the states of $\mathcal{A}_{min}$ with Nerode equivalence classes of states of $\mathcal{A}$ and therefore we obtain that:

(1) For each state $p \in P$ we have $p \cdot u \in P$ and $p \cdot v \in Q$.
(2) For each state $q \in Q$ we have $q \cdot u \in Q$ and $q \cdot w \in P$.

Let us consider the Nerode equivalence class associated with state P (the reasoning is the same for Q). Since $P \cdot u = P$, we can decompose it into components of states that are obtained by iterating the action of word u on a beginning state (as in Pollard's ρ method). So an equivalence class can be seen as a union of components having the following form:



Now, let us consider a state $p_1 \in P$. There exist $k, k' \geq 0$ and a state $p'_1 \in P$ such that $p_1 \cdot u^k = p'_1$ and $p'_1 \cdot u^{k'} = p'_1$ (p'_1 belongs to the loop of the component containing p_1). The state $q_1 = p'_1 \cdot v$ belongs to Q. Let us consider the state q'_1 defined from q_1, v as we have defined p'_1 from p_1: there exists a word $\widetilde{v}$ such that $p'_1 \cdot \widetilde{v} = q'_1$ and a word $\widetilde{u} = u^h$ (where h is the least common multiple of the lengths of the loops containing the states p'_1 and q'_1) such that $p'_1 \cdot \widetilde{u} = p'_1$ and $q'_1 \cdot \widetilde{v} = q'_1$. Therefore, as $\mathcal{A}$ does not match the pattern, $q'_1 \cdot w$ must not belong to the component of p'_1. We can iterate this reasoning as in the proof of Lemma 2 and we find a similar contradiction. In fact this proof is analogous but we must work on components instead of states.

6.2.4 Characterizing $\mathrm{V}\mathcal{L}(\uparrow_0)$

To characterize $\mathrm{V}\mathcal{L}(\uparrow_0)$, we can use either a result about languages definable using the yesterday operator, or give a direct proof (which gives us therefore a characterization of $\mathcal{L}(\mathbf{Y})$).

We begin with the direct proof because it illustrates the use of vectorial languages. Intuitively, if we have for a given deterministic complete automaton an algorithm using only the right-shift operation $\uparrow_0$, let us say k times, this means that for any word and for any position in this word we have to consider only the $k+1$ last letters for knowing the state reached by the automaton. Formally:

Theorem 7 *An automaton has an associated algorithm in* $\mathrm{VA}[\uparrow]$ *if and only*

if it is trivial (any letter loops on any state) or if there exists an integer k such that the transition functions defined by the words of length k are constant.

Effectively, let us consider a non trivial automaton $\mathcal{A}$ having an algorithm in VA[$\uparrow$]. We then have an algorithm computing the output vector $\mathbf{r}$ of the visited states from the input vector $\mathbf{u}$ and using only bit-wise logical operations and the right-shift. Let k be the number of right shift operations used. Therefore, it is easily seen that the nth position of $\mathbf{r}$ only depends on the positions $n, n-1, \ldots, n-k$ of $\mathbf{u}$. Thus, if u is a word of length $k+1$, then u leads to a state independent of the initial state, i.e, u defines a constant mapping in Q^Q, where Q is the set of states of $\mathcal{A}$, what proves the first implication.

Conversely, let us consider an automaton having this property and let us construct an algorithm in VA[$\uparrow$] for it. The case of the trivial automaton is not difficult and we will no longer deal with it.

For any word v of length k, we compute the characteristic vector $\mathbf{e}_v$ of v:

$$\mathbf{e}_v = \bigwedge_{i=0}^{k-1} (\uparrow_0^i (\mathbf{u} = \mathbf{a_{k-1-i}})$$

where $v = a_0 \cdots a_{k-1}$ and note by $\uparrow_0^i$ the operation $\uparrow_0$ iterated i times.

For any state q , we design by L_q^k the set of words of length k sending any state on q. Therefore, the vector $(\mathbf{r}' = \mathbf{q}) = \bigvee_{v \in L_q^k} \mathbf{e}_v$ matches, except possibly on the $k-1$ first terms, the characteristic vector $(\mathbf{r} = \mathbf{q})$. But it is easy to compute the $k-1$ first terms of $(\mathbf{r} = \mathbf{q})$: it suffices to consider the words w of length less or equal than $k-1$ that lead to q from the initial state. Therefore, we just have to compute their characteristic vectors $\mathbf{e}_w$ and to take their disjunction. Thus we obtain the vector $(\mathbf{r}'' = \mathbf{q})$ that matches $(\mathbf{r} = \mathbf{q})$ on the $k-1$ first terms. The vector $(\mathbf{r} = \mathbf{q})$ is finally given then by:

$$(\mathbf{r} = \mathbf{q}) = [(\mathbf{r}' = \mathbf{q}) \wedge \mathbf{x}] \vee [(\mathbf{r}'' = \mathbf{q}) \wedge \neg \mathbf{x}]$$

where we let $\mathbf{x} = \uparrow_0^k \mathbf{1} = 0^k 1^*$. Therefore we obtain an algorithm for $\mathcal{A}$ in VA[$\uparrow_0$].

We can give a corollary of this result in algebraic terms:

Corollary 1 *A regular language belongs to $\mathcal{L}[\uparrow_0]$ if and only if its syntactic semigroup belongs to the variety $\mathcal{D}$ of semigroups defined by the equation $yx^\omega = x^\omega$.*

In fact the equation $yx^\omega = x^\omega$ is associated with languages of the form $A^*X \cup Y$ where X and Y are finite sets of non-empty words on an alphabet A [8]. It

is therefore easy to verify, using Theorem 7, that the languages of $\mathcal{L}[\uparrow_0]$ are exactly those associated with the variety of semigroups $\mathcal{D}$. Effectively, let us consider an automaton for which any word of a given length k defines a constant mapping in Q^Q. Let e be an idempotent of the transition semigroup. As $e = e^k$, e can be associated with a word of length greater or equal than k and therefore e is associated with a constant mapping and thus it is right absorbing, i.e., for any element v of the transition semigroup we have $ve = e$. Consequently, the transition semigroup verifies the equation $yx^\omega = x^\omega$. Conversely, let us consider a language recognized by an automaton (that can be chosen deterministic and complete) such that its transition semigroup verifies the equation $yx^\omega = x^\omega$. To any state q of the automaton, we can associate a language L_q composed of all words that lead from the initial state to q. The syntactic semigroup of this language divides the transition semigroup of the given automaton and thus verifies the equation $yx^\omega = x^\omega$. Therefore, $L_q = A^*X \cup Y$ where X and Y are two finite sets of words. The elements of X define constant mappings that send any state on q. Making this reasoning for all states gives us for any state a set of characteristic words. Considering the longest word of these sets we find an integer k such that any word of length k defines a constant transition function.

Using a result on a fragment of temporal logic [16,17] and Lemma 1 we have the following characterizations:

Theorem 8 *Let L be a regular language over some alphabet A. Then the following assertions are equivalent:*

(1) *L belongs to $\mathrm{V}\mathcal{L}[\uparrow_0]$.*
(2) *L belongs to $\mathcal{L}[\mathbf{Y}]$.*
(3) *The closure of the minimal automaton of L, $\mathcal{A}_{min}(L)$ does not match the following pattern:*

(4) *The syntactic semigroup of L belongs to the variety $\mathcal{D}$ defined by the equation $yx^\omega = x^\omega$.*

6.2.5 Characterizing unambiguous languages

In this section we give a characterization of unambiguous languages using a fragment of PTL-vectorial languages. Let us consider an alphabet A. A product of the form $A_0^*a_1A_1^*a_2 \cdots a_kA_k^*$, where A_i is a subset of A and a_i is a

letter, is called *unambiguous* if for any word u on the alphabet A, if u belongs to the product then there is a unique decomposition $u_0, u_1, \ldots, u_k$ such that $u = u_0 a_1 u_1 a_2 \cdots a_k u_k$ with $u_i \in A_i^*$. An unambiguous language is a finite, disjoint union of unambiguous products.

Unambiguous languages are well studied. We will use there two results: the fact that unambiguous languages form a variety of languages and a characterization using a symmetric fragment of temporal logic. A symmetric fragment of temporal logic is defined as a classical fragment except that the use of future operators (and not only past operators) is allowed [16,17]. The symmetric fragment $\mathcal{L}[|\mathbf{XF}|]$ associated with unambiguous languages is the one allowing the use of the strict operators past ($\mathbf{YP}$) and future ($\mathbf{XF}$). The operator $\mathbf{XF}$ has the following semantics: $(w, n) \models \mathbf{XF}\varphi$ if there exists $n < m \leq |u|$ such that $(w, m) \models \varphi$.

Defining the operation strict left $\leftarrowtail$ as a symmetric version of $\rightarrowtail$, using Lemma 1 and the equivalence between unambiguous languages and the symmetric fragment $\mathcal{L}[|\mathbf{XF}|]$, we have the following result:

Theorem 9 *Let L be a regular language over some alphabet A. Then the following assertion are equivalent:*

(1) *L is unambiguous.*
(2) *L belongs to $\mathcal{L}[|\mathbf{XF}|]$.*
(3) *L belongs to $\mathrm{V}\mathcal{L}[\rightarrowtail, \leftarrowtail]$*

7 Reconstructing an automaton from a PTL-vectorial algorithm

In the preceding sections we wanted to find a vectorial algorithm from a given automaton. We now consider the converse problem, that is we want to check for a given PTL-vectorial algorithm whether there exists a deterministic complete automaton associated with it (and determine an automaton, if this is the case). This question becomes interesting for instance when we modify a given vectorial algorithm (associated with a deterministic automaton) and we want to check afterward that the new algorithm is equivalent to the old one. We will show that the complexity of this test is actually the same as testing the satisfiability of an LTL-formula (PSPACE-complete).

Vectorial algorithms are associated with deterministic complete automata and therefore depend on the initial state (and not only on the underlying labeled graph structure of the given automaton). We will thus suppose that the initial state is part of the input.

To begin with, let us consider a *valid* PTL-vectorial algorithm (i.e. an algorithm for which there exists a corresponding deterministic complete automaton) and let us explain how to construct such an associated automaton. Let $A = \{a_1, \ldots, a_k\}$ be the alphabet of the automaton and let n be the number of states (we will identify them with the integers $1 \ldots n$). To compute an associated automaton $\mathcal{A}_\Phi$ from a given PTL-vectorial algorithm Φ we perform a depth-first search of $\mathcal{A}_\Phi$, that is we start from the initial state q_0 and compute the states that can be reached by reading a letter from q_0 and then we repeat this step with the new states found so far. We are done when we have explored all reachable states. With this method we explore all the transitions of the accessible part of the automaton. We just have to explain how to compute the reachable states from a given state. In our algorithm we maintain a vector, *state_direction*, giving for any state encounter q a word u leading from the initial state to q. Therefore, when considering a state q, and a letter a to compute the transition from q reading a we have to apply Φ to the word ua and consider the $|u|+1$ component of the result, denoted $\Phi^{|u|+1}(ua)$.

We thus have the following algorithm:

- Variables and initialization:
  - δ: $(n \times k)$-vector.
  - $new\_states = [1]$: LIFO structure.
  - $known\_states = \{1\}$: Set structure.
  - $state\_direction = \underbrace{[\varepsilon, \varepsilon, \ldots, \varepsilon]}_{n}$.
- Main loop:
  While $new\_states \neq \emptyset$ **Do**
    Let q =Delete element from new_states.
    Let $u = state\_direction.(q)$.
    Let $h = |u|$.
    For $i = 1$ to k **Do**
      Let $q' = \Phi^{h+1}(ua_i)$.
      Let $\delta(q, i) = q'$.
      If $q' \notin known\_states$ **Then**
        Add q' to new_states and to $known\_states$.
        Set $state\_direction.(q') = ua$.
      End If.
    End For.
  End While.
- Return δ.

To test the validity of a given algorithm Φ we will first use the preceding algorithm to compute the automaton $\mathcal{A}_\Phi$ associated with Φ, if it is valid. If the algorithm does not work (that is if $\Phi^{h+1}(ua_i)$ is not defined for a given step of the algorithm) this implies that Φ is not valid. Otherwise we need to use

the validity test stated in the theorem below. For any state q, let $L(q)$ denote the regular language defined by the logical formula obtained as in Section 4.2 from the algorithm computing $(\mathbf{r} = \mathbf{q})$.

Theorem 10 *Let Φ be a PTL-vectorial algorithm and let $\mathcal{A}_\Phi$ be the deterministic complete automaton constructed by the algorithm above. Then Φ is a valid algorithm associated with $\mathcal{A}_\Phi$ if and only if:*

(1) *For any non reachable state q of $\mathcal{A}_\Phi$, we have $L(q) = \emptyset$.*
(2) *For any reachable state q, we have that $L(q) \neq \emptyset$. In addition, the following assertions are equivalent:*
   (i) *$L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$, where $E_q = \{\varepsilon\}$ if q is the initial state and $E_q = \emptyset$ otherwise. Moreover, a_j is a letter and each q_j is a reachable state.*
   (ii) *$\{(q_1, a_1), \ldots, (q_i, a_i)\}$ is exactly the set of the pairs (q_j, a_j) such that $q_j.a_j = q$ in $\mathcal{A}_\Phi$.*

Proof:

First let us assume that Φ is valid. This implies that for any state q, the word u belongs to $L(q)$ if and only if $q_0 \cdot u = q$, where q_0 denotes the initial state of $\mathcal{A}_\Phi$. Therefore, we easily obtain that $L(q) = \emptyset$, for any non reachable state q of $\mathcal{A}_\Phi$.

Let us now consider the case of a reachable state q and assume that (i) holds: $L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$. For any pair (q_j, a_j), as q_j is reachable, there exists $v \in L(q_j)$ such that $q_0 \cdot v = q_j$ and $va_j \in L(q)$. Consequently we have $q_j \cdot a_j = (q_0 \cdot v) \cdot a_j = q_0 \cdot u = q$. Conversely, consider a pair (q', a) such that $q' \cdot a = q$ and let us prove that $L(q) \supseteq L(q')a$. Let us consider a word $w \in L(q')$. As Φ is valid this implies that $q_0 \cdot w = q'$ and therefore $q_0 \cdot wa = q' \cdot a = q$, what shows that $L(q')a \subseteq L(q)$. We have thus shown that (i) implies (ii).

Let us now assume that (ii) holds. We will prove that $L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$ where $\{(q_1, a_1), \ldots, (q_i, a_i)\}$ is exactly the set of the pairs (q_j, a_j) such that $q_j \cdot a_j = q$. So let us consider a word $u \in L(q)$. As Φ is valid, we have that $q_0 \cdot u = q$ and therefore if $|u| \geq 2$, we can write u as $u = va$ with $q_0 \cdot u = (q_0 \cdot v) \cdot a$. So there exists j such that $(q_0 \cdot v, a) = (q_j, a_j)$ and $v \in L(q_j)$ and thus $u \in L(q_j)a_j$. The cases $|u| = 0$ and $|u| = 1$ are immediate as ε belongs to $L(q_0)$. Conversely, if we consider a word $u = va_j \in L(q_j)a_j$ we have that $q_0 \cdot u = (q_0 \cdot v) \cdot a_j = q_j \cdot a_j = q$ and thus $u \in L_q$. We have thus proved that (ii) implies (i).

Suppose now that for any non reachable state q, $L(q) = \emptyset$ and that for any reachable state q, the set $L(q)$ is non empty and that (i) and (ii) are equivalent.

Let us prove that this implies the validity of Φ. We work by contradiction assuming that Φ is not valid. We have two cases:

(1) There exists $u \in L(q)$ and $q_0 \cdot u \neq q$. We can choose u of minimal length. With this property, as $u \in L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$, we have $u = va_j$ (the case $|u| = \varepsilon$ is immediate) where $1 \leq j \leq i$ and $v \in L(q_j)$ (v can be empty). By minimality of u we must have $q_0.v = q_j$ and therefore, $q_0 \cdot u = (q_0 \cdot v) \cdot a_j = q_j \cdot a_j = q$ (by equivalence between (i) and (ii)) what leads to a contradiction.

(2) There exists a word u such that $q_0 \cdot u = q$ and $u \notin L(q)$. We can choose again u of minimal length. The case $u = \varepsilon$ is immediate and we can therefore decompose u as $u = va$ (where v can be empty). The minimality of u implies that $v \in L(q')$ where we set $q' = q_0 \cdot v$. But we also have that $q' \cdot a = q$ and thus, by equivalence between (i) and (ii) we have that $u = va \in L(q')a \subseteq L(q)$, what leads to a contradiction with $u \notin L(q)$.

We have thus proved that Φ is valid and so it is associated with $\mathcal{A}_\Phi$.

■

We can now give a method to test the validity of a PTL-vectorial algorithm Φ:

(1) We apply the depth-first search algorithm described above to Φ. If the algorithm does not yield a deterministic automaton $\mathcal{A}_\Phi$, then Φ is not a valid algorithm and we can stop. Otherwise we go to the next step.

(2) We determine the reachable states and the non reachable states of the automaton $\mathcal{A}_\Phi$ constructed in the preceding step.

(3) For every non reachable state q we translate the associated component in Φ into a PTL-formula φ_q and test whether or not it can be satisfied (see Section 8 and [12]). If φ_q is satisfiable for a non reachable-state q then Φ is not valid and we stop. Otherwise we go to the next step.

(4) For every reachable state q we determine the set $\{(q_1, a_1), \ldots, (q_i, a_i)\}$ of the pairs (q_j, a_j) such that $q_j \cdot a_j = q$ in $\mathcal{A}_\Phi$ and we verify that $L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$. To achieve this efficiently we can determine for every j a PTL-formula associated with $L(q_j)a_j$. It suffices to consider the formula $p_{a_j} \wedge \mathbf{Y}\varphi_{q_j}$ where φ_{q_j} is the translation of the component of Φ associated with q_j. Then, we can construct a PTL-formula for the language $L(q)\Delta[L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q]$, where Δ holds for the symmetric difference, and verify that it cannot be satisfied, what is equivalent to the equality $L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$. If the test does not fail, then Φ is valid and associated with $\mathcal{A}_\Phi$, otherwise Φ is not valid.

Let us now give the complexity of this algorithm. We will prove that determining whether or not a PTL-algorithm is valid is a PSPACE-complete problem. We first show that this test can be achieved in polynomial space.

The first step, the depth-first search algorithm, calculates all the transitions of the reachable part of $\mathcal{A}_\Phi$. As $\mathcal{A}_\Phi$ is a deterministic complete automaton, there are $O(n|A|)$ transitions, where n denotes the number of states of $\mathcal{A}_\Phi$ and A is the alphabet of $\mathcal{A}_\Phi$. The result of Φ applied to a given word can be computed in logarithmic space. Effectively the PTL-operations are logarithmic space operations and logarithmic space operations are closed by composition. Therefore, as $n = O(|\Phi|)$ and $|A| = O(|\Phi|)$ (the size of the algorithm is the size of the PTL-formula plus the size of A), the first step can be achieved in polynomial time. The second step, is performed also in polynomial time (and thus in polynomial space). In the third step, the construction and the size of φ_q is polynomial in $|\Phi|$. Determining whether or not a PTL-formula can be satisfied, is known to be a PSPACE-complete problem (see Section 8.5 and [12]). As $|\varphi_q|$ is polynomial in $|\Phi|$, this step can be achieved in polynomial space. For the same reasons the fourth step can also be achieved in polynomial space.

We have thus proved:

Proposition 2 *Deciding whether or not a PTL-vectorial algorithm is valid can be done in polynomial space.*

In fact, we can give a more precise result:

Theorem 11 *Deciding whether or not a PTL-vectorial algorithm is valid is a PSPACE-complete problem.*

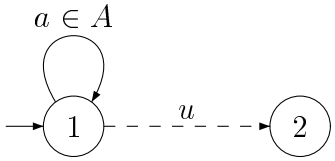
Proof:

We just have to prove the PSPACE-hardness. For this, we reduce the PSPACE-complete problem of deciding whether or not a PTL-formula can be satisfied. So let us consider a PTL-formula φ over some alphabet A. We will consider an automaton with two states, 1 and 2. Let Φ be the translation of φ into a PTL-vectorial formula. Then, we define a PTL-vectorial algorithm Φ' by:

$$\Phi' = \begin{cases} (\mathbf{r} = \mathbf{1}) = true \\ (\mathbf{r} = \mathbf{2}) = \Phi \end{cases}$$

where the initial state is 1.

We have that φ can be satisfied if and only if Φ' is not valid. The automaton constructed using the depth-first search algorithm is the solid part of the following automaton:



If φ can be satisfied, say by a word u, the algorithm Φ' does not give a correct result on u. Conversely, if Φ' is not correct, using Theorem 10, we have two cases:

(1) $L(2)$ is non empty, that is φ can be satisfied.
(2) $L(1)$ is empty (what is wrong) or $L(1) \neq L(1)a \cup \{\varepsilon\}$ (what is also wrong).

Therefore we have proved that φ can be satisfied if and only if Φ' is not valid. This proves that determining whether or not a PTL-vectorial algorithm is valid is a PSPACE-complete problem.

■

We now consider the same problem but for fragments of PTL-vectorial languages. For instance we have the following result for algorithms in $\mathrm{VA}(\uparrow_0)$:

Theorem 12 *Deciding whether or not an algorithm in* $\mathrm{VA}(\uparrow_0)$ *is valid is an NP-complete problem.*

As for the general problem we first use the depth-first search algorithm to determine an automaton such that our algorithm is valid if and only if it is associated with this automaton.

Using Theorem 8 it is easily seen that an algorithm in $\mathrm{VA}(\uparrow_0)$ is associated with a given automaton if and only if it is associated with it for words of length less or equal than $k+1$, where k designs the maximum number of nested shift operations. This implies the membership in NP (we have to determine k and then to guess a word of length less or equal than $k+1$ and finally to test the correctness of the algorithm for it).

In order to prove the NP-hardness we reduce the problem of deciding whether or not a formula in $\mathrm{PTL}(\mathbf{Y})$ can be satisfied to our problem. For this we use the same reduction as in Theorem 11. We conclude the proof using the following lemma:

Lemma 4 *Deciding whether or not a formula in* $\mathrm{PTL}(\mathbf{Y})$ *is satisfiable, is an NP-complete problem.*

Proof:

The membership in NP is not difficult: a formula in PTL($\mathbf{Y}$) can be satisfied if and only if it can be satisfied by a word of length less or equal than $k+1$, where k designs the maximum number of nested $\mathbf{Y}$ operators (effectively the truth of a formula in PTL($\mathbf{Y}$) applied to a word u only depends on the suffix of length $k+1$ of u).

The NP-hardness is shown by a reduction from the NP-complete problem SAT. Let us consider a propositional formula F and let us construct a formula φ in PTL($\mathbf{Y}$) such that F can be satisfied if and only if φ can be satisfied. We denote by $p_1, \ldots, p_n$ the propositional variables used in F. The alphabet of the temporal formula φ is the boolean alphabet: $\{\top, \bot\}$, and φ is constructed from F by replacing each propositional variable p_i by $\underbrace{\mathbf{Y} \ldots \mathbf{Y}}_{n-i} p_\top$ ($p_\top$ is the predicate associated with the letter $\top$).

For example for $F = (p_2 \vee p_1) \wedge \neg[p_1 \vee (p_3 \wedge p_2)]$ we define:

$$\varphi = (\mathbf{Y}p_\top \vee \mathbf{YY}p_\top) \wedge \neg[\mathbf{YY}p_\top \vee (p_\top \wedge \mathbf{Y}p_\top)]$$

For any formula F, we easily have that F is satisfied by a valuation $(b_1, \ldots, b_n)$, where each b_i is a boolean ($b_i = \top$ or $\bot$), if and only if $(b_1 \ldots b_n, n) \models \varphi$. This shows that SAT can be polynomially reduced to our problem and therefore achieves the proof.

■

8 Reconstructing automata from GTL-vectorial algorithms

In the preceding section we have shown how to decide whether there exists an associated counter-free automaton with a given PTL-vectorial algorithm Φ. For this, we first construct an automaton $\mathcal{A}_\Phi$ associated with Φ, if is valid. Then, using Theorem 10 we decide whether Φ is valid. A natural investigation is to try to extend these results to MTL-vectorial and GTL-vectorial languages introduced in Sections 5.1 and 5.2. The main result of this section states that deciding the validity of a GTL-vectorial algorithm is PSPACE-complete. For obtaining this result, we review the construction of alternating automata from temporal logic formulas and show how to deal with modular and group operators and we also use that Theorem 10 does not actually depend on the vectorial operations allowed in our algorithm and can be stated in a more general way, by assuming Φ is a vectorial algorithm.

For a PTL-vectorial algorithm Φ, in order to compute the automaton $\mathcal{A}_\Phi$ we simulate a depth-first search algorithm. This algorithm can be adapted to

MTL-vectorial algorithms and to GTL-vectorial algorithms without change. Nevertheless, its complexity is not the same as the simulation of Φ in the general case of modular and group operators is more costly. Actually, we have the following result:

Lemma 5 *Let Φ be a GTL-vectorial algorithm and let u be a word. Then the computation of the result of Φ applied to u can be achieved in $O(|u|.|\Phi|)$ operations.*

Proof:

The result trivially holds in the special case of PTL-vectorial algorithms. We denote by $C(\Phi, u)$ the cost of the computation of Φ applied to u. If $\Phi = S_{l,k}(\Phi_1)$, to compute the result of Φ applied to u, we first compute the result of Φ_1 applied to u and then read it from left to right to determine the final result. We have that $C(\Phi, u) = C(\Phi_1, u) + |u|$. Therefore modular operators cost linear time. If $\Phi = P_{g,G}(\Phi_1, \ldots, \Phi_q)$, to compute the result of Φ applied to u, we first compute the results of $\Phi_1, \ldots, \Phi_q$ applied to u and then read them simultaneously from left to right to determine the final result. We have that $C(\Phi, u) = C(\Phi_1, u) + \cdots + C(\Phi_q, u) + |u|$. Thus group operators also cost linear time. Therefore, the computation of the result of Φ applied to u can be achieved in $O(|u|.|\Phi|)$ operations.

■

Lemma 5 implies that the computation by the depth-first search algorithm of an associated automaton $\mathcal{A}_\Phi$ with Φ can be made in polynomial time. There is another question left in order to solve our problem, that is how to use Theorem 10 for MTL-vectorial and GTL-vectorial algorithms. As for PTL-vectorial languages, the equality on languages to verity can be translated into a satisfiability problem, for GTL-formula in this case. In Section 8.5, we prove that the satisfiability problem for GTL-formula is PSPACE-complete and therefore, we have the following result:

Theorem 13 *Deciding whether or not a GTL-vectorial algorithm is valid is a PSPACE-complete problem.*

To prove the satisfiability result for GTL-formulas, we use alternating automata and reduce the satisfiability problem to a non emptiness problem for alternating automata.

An alternating automaton is a tuple $\mathcal{A} = (Q, A, \delta, q_0, F)$, where Q is a finite set of states, A is a finite alphabet, q_0 is the initial state, F is the set of final states and $\delta : Q \times A \to \mathcal{B}^+(Q)$ is the transition function, where $\mathcal{B}^+(Q)$ is the set of all negation-free boolean formulas over Q.

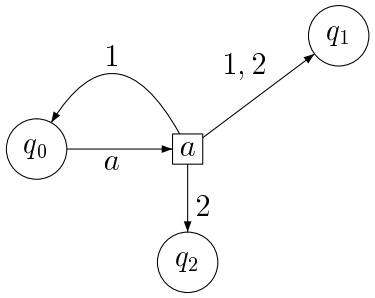
A run of an alternating automaton is a finite tree whose nodes are labeled with states of Q and edges with elements of A. The level of a node is the length of the word labeling the path from the root to this node. A run associated with a finite word $u = a_1 a_2 \cdots a_n$ is defined by induction:

(1) The root is q_0.
(2) The nodes of level n are leaves (i.e. they have no sons).
(3) If q is a state of level $i < n$ and $\delta(q, a_i) = C_1 \vee C_2 \vee \cdots \vee C_m$ with $C_j = q_{j,1} \wedge q_{j,2} \wedge \cdots \wedge q_{j,n_j}$ then q has n_j sons for some j, $1 \leq j \leq m$, labeled by $q_{1,k_1}, q_{j,1}, \ldots, q_{j,n_j}$. That is, q must have as sons all the states appearing in one of the conjunctions C_j.

Remark 3 In our definition of a run, $\delta(q, a)$ is in disjunctive normal form for any state q and any letter a. Of course, δ could be defined as a function taking its values in negation-free boolean formulas in disjunctive normal form, but the constructions given in Sections 8.3 and 8.4 would lead to consider alternating automata with an exponential number of transitions. In fact we will not be interested in computing such automata but in runs of them. Therefore, for any formula $\delta(q, a)$, a minimal model (whose size will always be linear in the number of states) will be computed whenever we need it. A model for a formula is a set R of states, such that assigning to the states in R the value *tt* and to those on $Q \setminus R$ the value *ff* makes the formula true. Nevertheless, for representing alternating automata we will work with formulas in disjunctive normal form.

A word u is *accepted* by $\mathcal{A}$ if there exists a run r associated with u such that all the leaves of r are final states. The language recognized by an alternating automaton $\mathcal{A}$ is noted $L(\mathcal{A})$.

Alternating automata will be drawn as classical automata except for the fact that the outgoing edges go first into a square (that is not a state!) that redirects the transition into groups of states (represented by the same index written on their incoming edges). For example the transition $\delta(q_0, a) = (q_1 \wedge q_0) \vee (q_1 \wedge q_2)$ is represented by:

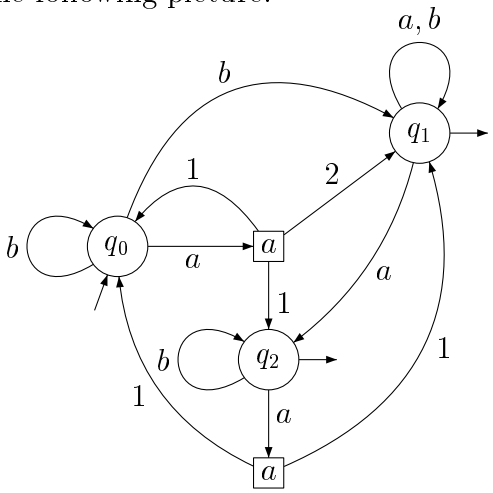


In the special case where $\delta(q, a)$ is a disjunction (that is $n_j = 1$ for all $j = 1 \dots m$) we represent the transition $\delta(q, a)$ as a classical existential (i.e. non deterministic) transition.

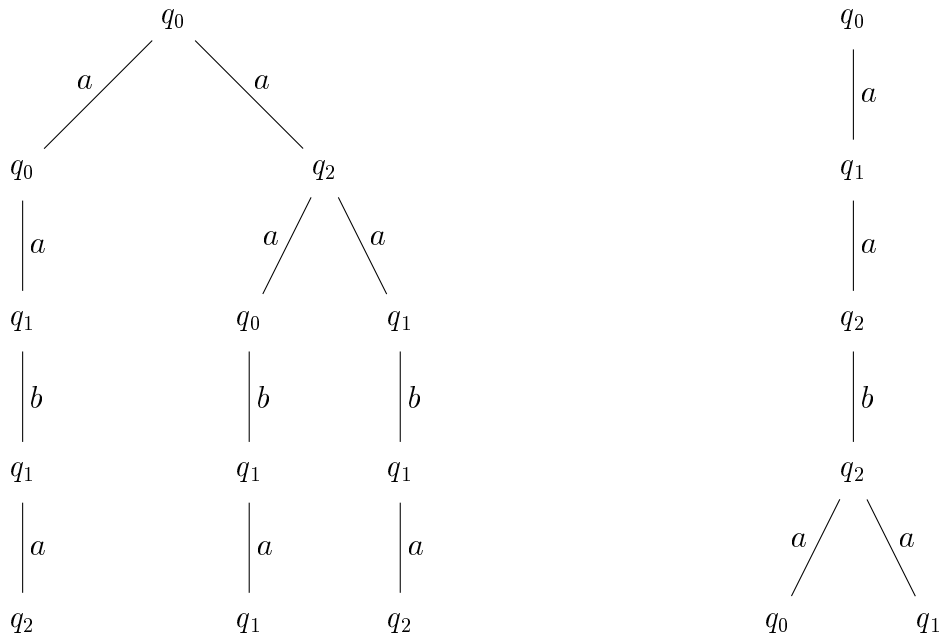
Example 2 Consider the alternating automaton $\mathcal{A} = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{q_0\}, \{q_1, q_2\})$, where we have:

- $\delta(q_0, a) = (q_0 \wedge q_2) \vee q_1$, $\delta(q_1, a) = q_1 \vee q_2$ and $\delta(q_2, a) = q_0 \wedge q_1$.
- $\delta(q_0, b) = q_1 \vee q_0$, $\delta(q_1, b) = q_1$ and $\delta(q_2, b) = q_2$.

$\mathcal{A}$ is represented by the following picture:



Let us now give two runs for the word $u = aaba$ in $\mathcal{A}$: the first one is accepting (therefore u is recognized by $\mathcal{A}$), whereas the second one is not accepting.



8.2 Linear temporal logic

Similar to the past temporal logic, the future temporal logic, called Linear Temporal Logic (LTL) is defined using the temporal operators Next (denoted **X**), and Until (denoted **U**).

X and **U** are respectively the future equivalents of the operators **Y** and **S**. Therefore their semantics is defined by:

(1) $(w, n) \models \mathbf{X}\varphi_1$ if $n < |w|$ and $(w, n+1) \models \varphi_1$.
(2) $(w, n) \models \varphi_1 \,\mathbf{U}\, \varphi_2$ if there exists $m \geq n$ such that $(w, m) \models \varphi_2$ and, for every k such that $n \leq k < m$, $(w, k) \models \varphi_1$.

An LTL-formula φ is satisfied by a word w if $(w, 1) \models \varphi$. An LTL-formula φ is called *satisfiable* if its associated language $L_\varphi = \{w \mid (w, 1) \models \varphi\}$ is not empty.

With an LTL-formula φ one can associate a PTL-formula $\widetilde{\varphi}$ by replacing the operator **X** by the operator **Y** and the operator **S** by the operator **U**. It is easily seen that, for any word w, $(w, 1) \models \varphi$ if and only if $(\widetilde{w}, |w|) \models \widetilde{\varphi}$, where $\widetilde{w}$ designs the mirror image of w. Thus, to decide whether a PTL-formula is satisfiable, it suffices to know how to solve the problem for LTL-formulas.

In the next section we recall the construction of an alternating automaton recognizing the language L_φ, where φ is an LTL-formula [15]. We need the construction in order to generalize it to the more expressive temporal logics.

For convenience, we use a new operator called Release (denoted $\mathbf{R}$). The release operator is defined by the formula $\varphi_1 \mathbf{R} \varphi_2 = \neg(\neg\varphi_1 \mathbf{U} \neg\varphi_2)$, or equivalently by: $(w,n) \models \varphi_1 \mathbf{R} \varphi_2$ if and only if for all m, $n \leq m \leq |w|$, such that $(w,m) \not\models \varphi_2$, there exists $n \leq i < m$ such that $(w,i) \not\models \varphi_1$. The release operator requires its second argument to be true, a condition that is released as soon as the first argument becomes true.

Introducing the release operator allows to construct, for any LTL-formula φ, an equivalent *positive* formula ψ, i.e. a formula that does not use the negation. The formula ψ is constructed by induction on φ and is of size $O(|\varphi|)$:

(1) If $\varphi = p_a$ where a is a letter, $\psi = \varphi$.
(2) If $\varphi = \neg p_a$ where a is a letter, $\psi = \bigvee_{b \in A \setminus \{a\}} p_b$.
(3) If $\varphi = \varphi_1 \vee \varphi_2$ then $\psi = \psi_1 \vee \psi_2$ where ψ_1 and ψ_2 are respectively constructed from φ_1 and φ_2.
(4) If $\varphi = \neg(\varphi_1 \vee \varphi_2)$ then $\psi = \psi_1 \wedge \psi_2$ where ψ_1 and ψ_2 are respectively constructed from $\neg\varphi_1$ and $\neg\varphi_2$.
(5) If $\varphi = \varphi_1 \wedge \varphi_2$ then $\psi = \psi_1 \wedge \psi_2$ where ψ_1 and ψ_2 are respectively constructed from φ_1 and φ_2.
(6) If $\varphi = \neg(\varphi_1 \wedge \varphi_2)$ then $\psi = \psi_1 \vee \psi_2$ where ψ_1 and ψ_2 are respectively constructed from $\neg\varphi_1$ and $\neg\varphi_2$.
(7) If $\varphi = \mathbf{X}\varphi_1$ then $\psi = \mathbf{X}\psi_1$ where ψ_1 is constructed from φ_1.
(8) If $\varphi = \neg\mathbf{X}\varphi_1$ then $\psi = \mathbf{X}\psi_1$ where ψ_1 is constructed from $\neg\varphi_1$.
(9) If $\varphi = \varphi_1 \mathbf{U} \varphi_2$ then $\psi = \psi_1 \mathbf{U} \psi_2$ where ψ_1 and ψ_2 are respectively constructed fromφ_1 and φ_2.
(10) If $\varphi = \neg(\varphi_1 \mathbf{U} \varphi_2)$ then $\psi = \psi_1 \mathbf{R} \psi_2$ where ψ_1 and ψ_2 are respectively constructed from $\neg\varphi_1$ and $\neg\varphi_2$.
(11) If $\varphi = \varphi_1 \mathbf{R} \varphi_2$ then $\psi = \psi_1 \mathbf{R} \psi_2$ where ψ_1 and ψ_2 are respectively constructed from φ_1 and φ_2.
(12) If $\varphi = \neg(\varphi_1 \mathbf{R} \varphi_2)$ then $\psi = \psi_1 \mathbf{U} \psi_2$ where ψ_1 and ψ_2 are respectively constructed from $\neg\varphi_1$ and $\neg\varphi_2$.

For example if $\varphi = \neg[p_a \mathbf{U} (p_b \vee \mathbf{X} p_a)]$ and $A = \{a, b, c\}$, the associated formula is $\psi = (p_b \vee p_c) \mathbf{R} [(p_a \vee p_c) \wedge \mathbf{X}(p_b \vee p_c)]$

Given a positive LTL-formula φ, there exists an alternating automaton $\mathcal{A}_\varphi = (Q, A, \delta, q_0, F)$, whose number of states is linear in the size of φ recognizing the language L_φ (see also [15]).

(1) The alphabet A of $\mathcal{A}_\varphi$ is the alphabet of the words on which φ is evaluated.
(2) The states of $\mathcal{A}_\varphi$ are the sub-formulas appearing in φ and their negations $\overline{\varphi}$ (written without using the negation as described in Section 8.2) plus the constants tt (True) and ff (False).
(3) $q_0 = \varphi$.
(4) $F = \{tt\} \cup \{\varphi = \varphi_1 \mathbf{R} \varphi_2 \mid \varphi \in Q\}$.
(5) δ is inductively defined by the following rules:
   (i) $\delta(tt, a) = tt$ and $\delta(ff, a) = ff$ for any letter a.
   (ii) $\delta(p_a, b) = \begin{cases} tt & \text{if } a = b, \\ ff & \text{otherwise.} \end{cases}$
   (iii) $\delta(\varphi_1 \vee \varphi_2, a) = \delta(\varphi_1, a) \vee \delta(\varphi_2, a)$.
   (iv) $\delta(\varphi_1 \wedge \varphi_2, a) = \delta(\varphi_1, a) \wedge \delta(\varphi_2, a)$.
   (v) $\delta(\mathbf{X}\varphi, a) = \varphi$ for all $a \in A$.
   (vi) $\delta(\varphi_1 \mathbf{U} \varphi_2, a) = \delta(\varphi_2, a) \vee [\delta(\varphi_1, a) \wedge (\varphi_1 \mathbf{U} \varphi_2)]$.
   (vii) $\delta(\varphi_1 \mathbf{R} \varphi_2, a) = \delta(\varphi_2, a) \wedge [\delta(\varphi_1, a) \vee (\varphi_1 \mathbf{R} \varphi_2)] = [\delta(\varphi_2, a) \wedge \delta(\varphi_1, a)] \vee [\delta(\varphi_2, a) \wedge (\varphi_1 \mathbf{R} \varphi_2)]$.

We have the following result what is shown in [15]. The detailed proof can be found in appendix.

Theorem 14 *Let φ be a positive LTL-formula and let $\mathcal{A}_\varphi$ be the automaton associated with φ. Then $L_\varphi = L(\mathcal{A}_\varphi)$.*

8.4 From GTL-formulas to equivalent alternating automata

As alternating automata allow to recognize all regular languages, a natural investigation consists in associating an alternating automaton to formulas using modular or group operators. These operators were introduced in Sections 5.1 and 5.2. The modular operators were defined as past temporal operators. As we want to decide whether or not a temporal formula can be satisfied, we will work with the dual operators, as defined for LTL. Therefore to decide whether a MTL-formula or a GTL-formula can be satisfied it suffices to decide the same problem for the dual formula.

We thus give the definitions of the modular and group temporal operators for LTL (we will not change the notation with past temporal logic as no confusion can be made here):

- With any pair (l,k) of integers such that $0 \leq l < k$ we associate a unary modular operator $\mathrm{Mod}_{l,k}$ such that for any word u, we have $(u,i) \models \mathrm{Mod}_{l,k}(\varphi)$ if and only if, modulo k, there are l positions $j \geq i$ such that $(u,j) \models \varphi$.
- With any pair (g,G), where G is a group and g is an element of G, we associate a group operator $\Gamma_{g,G}$ that always binds $|G|-1$ formulas. The elements of G must have been ordered, say as $g_1, g_2, \ldots, g_q = id$ (the last element must be the identity). Let u be an element of A^+ and let $\varphi_1, \varphi_2, \ldots, \varphi_{q-1}$ be logical formulas. With each j, $1 \leq j \leq |u|$ we associate an element of G, denoted $\langle \varphi_1, \varphi_2 \ldots \varphi_{q-1} \rangle \langle u, j \rangle$, defined by:

$$\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle \langle u, j \rangle = g_k$$

  where $k = \min\{l \mid (u,j) \models \varphi_l\}$ with the convention that $\min \emptyset = q$.
  Finally we have $(u,i) \models \Gamma_{g,G}\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle$ if and only if

$$\prod_{j=i}^{|u|} \langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle \langle u, j \rangle = g$$

LTL extended by the modular operators will be denoted as MLTL. The extension by the group operators will be denoted GLTL. We have the following extension of Theorem 14:

Theorem 15 *Let φ be a GLTL-formula. Then there exists an alternating automaton $\mathcal{A}_\varphi$ such that $L_\varphi = L(\mathcal{A}_\varphi)$. In addition, the number of states of $\mathcal{A}_\varphi$ is quadratic in the size of φ.*

Proof:

The modular operator is a special case of group temporal operators using only cyclic groups $(\mathbb{Z}/k\mathbb{Z}, +)$, as $(u,i) \models \mathrm{Mod}_{l,k}(\varphi)$ if and only if we have $(u,i) \models \Gamma_{l,(\mathbb{Z}/k\mathbb{Z},+)}\langle \varphi, ff, ff, \ldots, ff \rangle$ Therefore it suffices to consider the general case of GLTL.

To keep working only with negation-free formulas, we have to explain, as in Section 8.2, how to construct a positive formula ψ from a formula $\varphi = \neg\Gamma_{g,G}\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle$. Here it suffices to take:

$$\psi = \bigvee_{g' \neq g} \Gamma_{g',G}\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle$$

The alternating automaton $\mathcal{A}_\varphi = (Q, A, \delta, q_0, F)$ recognizing L_φ is defined almost as in Section 8.3:

(1) The alphabet A of $\mathcal{A}_\varphi$ is the alphabet of the words on which φ is evaluated.

(2) The states of $\mathcal{A}_\varphi$ are the sub-formulas appearing in φ and their negations $\overline{\varphi}$ (written without using the negation, as described in Section 8.2) plus the constants tt (True) and ff (False). In addition, for any sub-formula $\Gamma_{g,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$ appearing in φ we add, for any $g' \neq g$, the state $\Gamma_{g',G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$ and its "positive negation".

(3) $q_0 = \varphi$.

(4) $F = \{\mathit{tt}\} \cup \{\varphi = \varphi_1 \mathbf{R} \varphi_2 \mid \varphi \in Q\} \cup \{\varphi = \Gamma_{id,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle \mid \varphi \in Q\}$.

(5) δ is inductively defined by the following rules:

(i) $\delta(\mathit{tt}, a) = \mathit{tt}$ and $\delta(\mathit{ff}, a) = \mathit{ff}$ for any letter a.

(ii) $\delta(p_a, b) = \begin{cases} \mathit{tt} & \text{if } a = b, \\ \mathit{ff} & \text{otherwise.} \end{cases}$

(iii) $\delta(\varphi_1 \vee \varphi_2, a) = \delta(\varphi_1, a) \vee \delta(\varphi_2, a)$.

(iv) $\delta(\varphi_1 \wedge \varphi_2, a) = \delta(\varphi_1, a) \wedge \delta(\varphi_2, a)$.

(v) $\delta(\mathbf{X}\varphi, a) = \varphi$ for all $a \in A$.

(vi) $\delta(\varphi_1 \mathbf{U} \varphi_2, a) = \delta(\varphi_2, a) \vee [\delta(\varphi_1, a) \wedge (\varphi_1 \mathbf{U} \varphi_2)]$

(vii) $\delta(\varphi_1 \mathbf{R} \varphi_2, a) = \delta(\varphi_2, a) \wedge [\delta(\varphi_1, a) \vee (\varphi_1 \mathbf{R} \varphi_2)] = [\delta(\varphi_2, a) \wedge \delta(\varphi_1, a)] \vee [\delta(\varphi_2, a) \wedge (\varphi_1 \mathbf{R} \varphi_2)]$

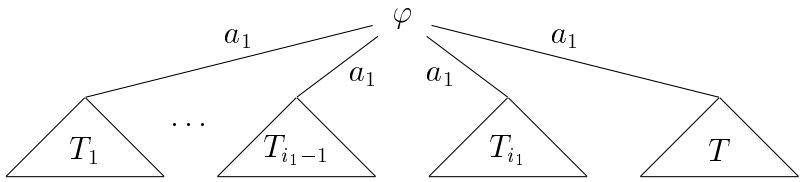
(viii) $\delta(\mathrm{Mod}_{k,l}(\varphi), a) = [\delta(\varphi, a) \wedge \mathit{Mod}_{k-1,l}(\varphi)] \vee [\delta(\overline{\varphi}, a) \wedge \mathit{Mod}_{k,l}(\varphi)]$ for all $a \in A$.

(ix) $\delta(\Gamma_{g,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle, a) = \bigvee_{g_i g_j = g} [\Delta(i, a) \wedge \Gamma_{g_j,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle]$, where $\Delta(i, a) = \delta(\overline{\varphi_1}, a) \wedge \delta(\overline{\varphi_2}, a) \wedge \cdots \wedge \delta(\overline{\varphi_{i-1}}, a) \wedge \delta(\varphi_i, a)$, where $\varphi_q = \mathit{tt}$.

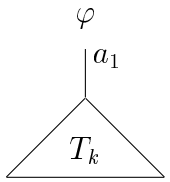
where (viii) is in fact a special case of (ix).

The number of states of $\mathcal{A}_\varphi$ is effectively linear in the size of φ times the sum of the cardinalities of the groups used in modular operators appearing in φ and as this sum is linear in the size of φ, it follows that the number of states of $\mathcal{A}$ is quadratic in the size of φ.

The proof is the same as for Theorem 14. We reason by induction on the formula φ. The only new case to consider is the one of $\varphi = \Gamma_{g,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$. Any run of $\mathcal{A}_\varphi$ on a non-empty word u whose first letter is a_1 has the following form:



where T is a run of $\mathcal{A}_{\psi_j}$, $\psi_j = \Gamma_{g_j,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$ and $g_{i_1} g_j = g$, and where any tree of the following form is a run of $\mathcal{A}_{\gamma_k}$ for $k = 1 \ldots i_1$, where $\gamma_k = \neg\varphi_k$ if $k < i_1$ and $\gamma_{i_1} = \varphi_{i_1}$ (recall that $\varphi_q = \not\Vdash$):



By induction hypothesis, this run is accepting if and only if $(u, 1) \models \neg\varphi_1 \wedge \neg\varphi_2 \wedge \cdots \wedge \neg\varphi_{i_1-1} \wedge \varphi_{i_1}$ and $(u, 2) \models \Gamma_{g_j,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$ that is if and only if $\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle\langle u, 1\rangle = g_{i_1}$ and $(u, 2) \models \Gamma_{g_j,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$.

Iterating this construction shows that with an accepting run of $\mathcal{A}_\varphi$ on a word $u = a_1 \cdots a_n$ one can associate a sequence $g_{i_1}, \ldots g_{i_n}, g_{i_{n+1}}$ of elements of G such that $g = g_{i_1} g_{i_2} \ldots g_{i_n} g_{i_{n+1}}$, for any $k = 1 \ldots n$, $\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle\langle u, k\rangle = g_{i_k}$ and such that $(u, n) \models \Gamma_{g_{i_{n+1}},G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$. But, as the unique final state of the form $\Gamma_{g_j,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$ is $\Gamma_{id=g_q,G}\langle\varphi_1, \varphi_2, \ldots, \varphi_{q-1}\rangle$ it follows that $g_{i_{n+1}}$ must be equal to the neutral element g_q of G and therefore that $g = g_{i_1} g_{i_2} \ldots g_{i_n}$, that is $(u, n) \models \varphi$. Conversely, with a word satisfying φ it is easily seen how to construct an accepting run in $\mathcal{A}$, what shows that we have $L(\mathcal{A}_\varphi) = L_\varphi$ and achieves the proof.

■

Example 3 Consider the MTL-formula $\varphi = [\mathrm{Mod}_{1,3}(\mathbf{X}p_a \vee p_b)] \wedge [p_a \,\mathbf{U}\, p_b]$ and let us describe the transition function δ associated with $\mathcal{A}_\varphi$:

q	$\delta(q,a)$	$\delta(q,b)$
p_a	tt	ff
p_b	ff	tt
$\mathbf{X}p_a$	p_a	p_a
$\overline{\mathbf{X}p_a} = \mathbf{X}p_b$	p_b	p_b
$p_a \,\mathbf{U}\, p_b$	$p_a \,\mathbf{U}\, p_b$	tt
$\mathbf{X}p_a \vee p_b$	p_a	tt
$\overline{\mathbf{X}p_a \vee p_b} = \mathbf{X}p_b \wedge p_a$	p_b	ff
$\varphi_1 = \mathrm{Mod}_{1,3}(\mathbf{X}p_a \vee p_b)$	$(p_a \wedge \varphi_0) \vee (p_b \wedge \varphi_1)$	φ_0
$\varphi_2 = \mathrm{Mod}_{2,3}(\mathbf{X}p_a \vee p_b)$	$(p_a \wedge \varphi_1) \vee (p_b \wedge \varphi_2)$	φ_1
$\varphi_0 = \mathrm{Mod}_{0,3}(\mathbf{X}p_a \vee p_b)$	$(p_a \wedge \varphi_2) \vee (p_b \wedge \varphi_0)$	φ_2
φ	$[p_a \wedge \varphi_0 \wedge (p_a \,\mathbf{U}\, p_b)] \vee [p_b \wedge \varphi_1 \wedge (p_a \,\mathbf{U}\, p_b)]$	φ_0

The alternating automaton $\mathcal{A}_\varphi$ is represented in Figure 1. In fact, we represent only the reachable part and represent in dash a copy of p_a and a copy of p_b to improve readability.

8.5 Emptiness problem for alternating automata and its consequences

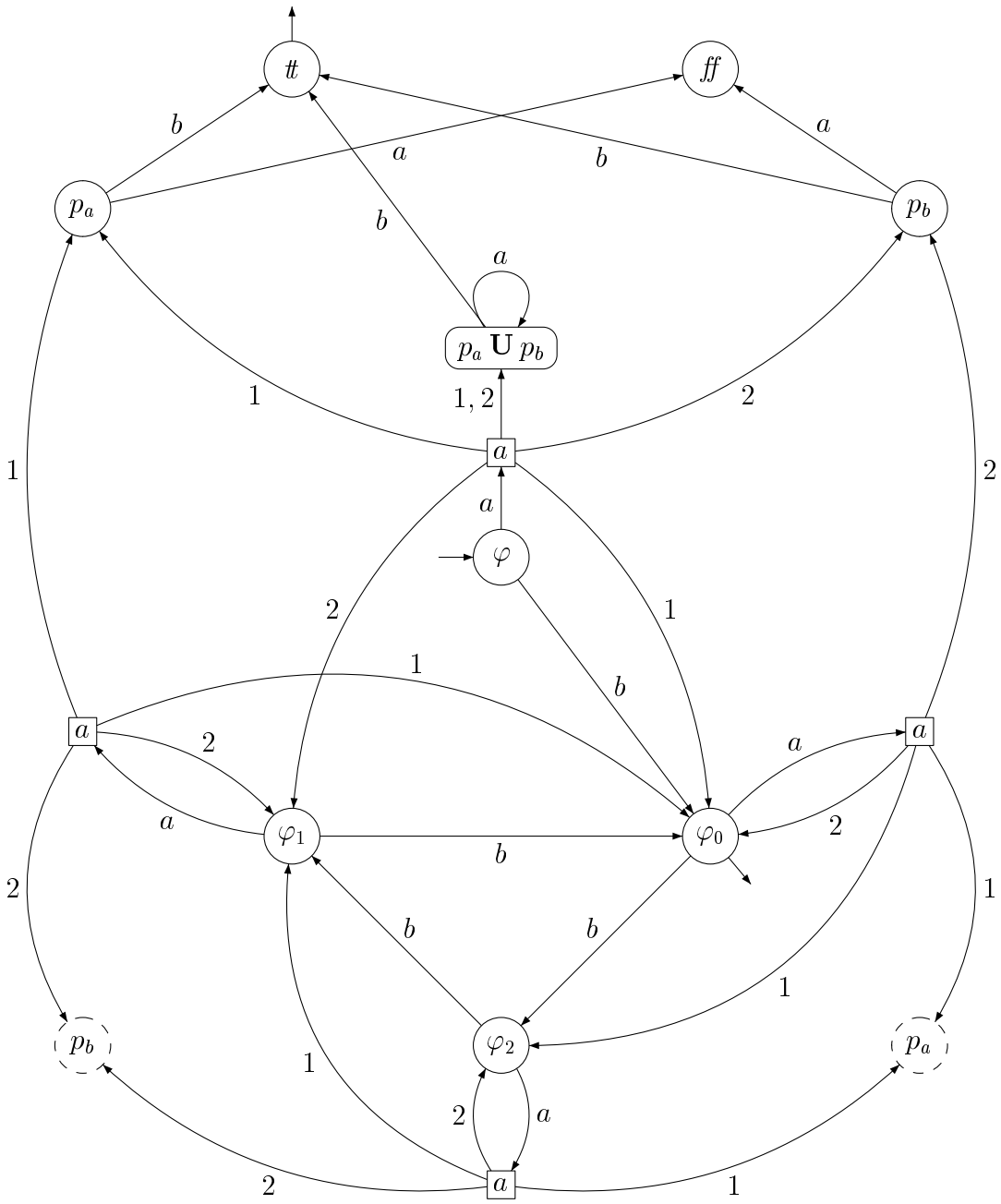
In Sections 8.3 and 8.4 we have shown how to associate with a GLTL-formula φ an alternating automaton $\mathcal{A}_\varphi$ recognizing exactly the models of φ. Therefore to decide satisfiability for GLTL-formulas (or for GTL-formulas) it suffices to know how to decide emptiness for alternating automata. We have the following result:

Theorem 16 *Let $\mathcal{A}$ be an alternating automaton, then testing whether $L(\mathcal{A}) = \emptyset$ can be realized in polynomial space.*

Proof:

As non-deterministic polynomial space is equal to deterministic polynomial space, we give a non-deterministic algorithm. To prove the non-emptiness of the language recognized by $\mathcal{A}$ we only have to construct an accepting run of $\mathcal{A}_\varphi$. The algorithm starts with the initial state φ and guesses a letter a_1 and a minimal model for $\delta(\varphi, a_1)$ (seen as a boolean positive formula). Then, it guesses the next letter a_2 and for any state appearing in the minimal model, it guesses a minimal model for its image by δ reading a_2 and therefore computes

Fig. 1. $\mathcal{A}_\varphi$ for $\varphi = [\mathrm{Mod}_{1,3}(\mathbf{X}p_a \vee p_b)] \wedge [p_a \,\mathbf{U}\, p_b]$



a set of states modeling all the preceding formulas (the algorithm works with a set of states and therefore it only needs a linear space to recall it) and so on. Finally, it decides to stop and accepts if all the actual states are final states.

To guess a minimal model of a boolean positive formula it suffices to explore all the possible valuations what gives the size of a minimal model and then to guess one of the minimal models. This can be made in polynomial space

and therefore the entire algorithm only needs polynomial space. In fact, the algorithm could just guess a model without verifying it is a minimal one, as by non determinism there exists a run of the algorithm where all guessed models are minimal.

■

We therefore have the following corollary:

Corollary 2 *Deciding whether an GLTL-formula (or a GTL-formula) is satisfiable is a PSPACE-complete problem.*

Proof:

The PSPACE membership is a consequence of Theorem 15 and Theorem 16. The PSPACE-hardness is a consequence of the PSPACE-hardness for the same problem restricted to PTL-formulas [12].

9 Conclusion

Using vectorial algorithms we have given new characterizations of star-free languages (as the class of PTL-vectorial languages), of solvable languages (as the class of MTL-vectorial languages) and of regular languages (as the class of GTL-vectorial languages). However, even in the easiest case, that is for star-free languages, there is no general efficient method to compute an algorithm associated with a given language. Nevertheless, since vectorial languages are closely related with temporal logic this is not that surprising at all, as the computation of an algorithm associated with an automaton is at least as difficult as finding a temporal logic formula associated with a given language, which is exponential with regard to the automaton.

We have characterized subsets of vectorial operations by equivalent sets of temporal logic operators.

It is interesting to note that vectorial algorithms provide a more detailed information about an automaton than logical formulas without any loss in computational complexity and in the complexity of the operators used in both models.

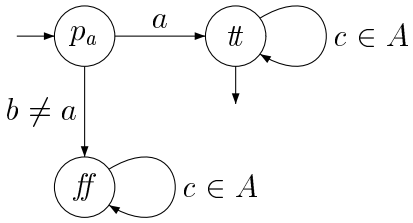
Finally, we have shown that deciding the validity of a GTL-vectorial algorithm is PSPACE-complete. As a byproduct we have obtained that the extension of LTL with group operators does not change the complexity of the satisfiability problem, which is still PSPACE-complete, and we have given an effective algorithm deciding this question.

10 Appendix: proof of Theorem 14

The aim is to prove that if φ is a positive LTL-formula, then we have that $L_\varphi = L(\mathcal{A}_\varphi)$, where $\mathcal{A}_\varphi$ is the automaton associated with φ.

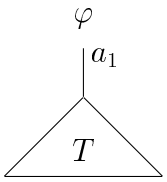
For this we reason by induction on the formula φ:

(1) If $\varphi = p_a$ then $\mathcal{A}_\varphi$ is the following alternating automaton:

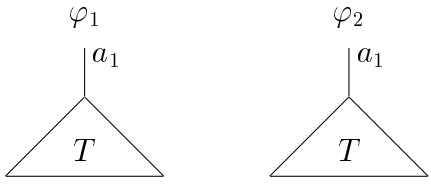


and therefore it easily seen that $\mathcal{A}_\varphi$ recognizes the language: $L(\mathcal{A}_\varphi) = \{aw \mid w \in A^*\} = L_\varphi$.

(2) If $\varphi = \varphi_1 \vee \varphi_2$ then any run of $\mathcal{A}_\varphi$ has the following form:

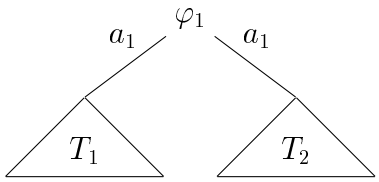


where a_1 is the first letter of the word and where one of the following runs is a run of $\mathcal{A}_{\varphi_1}$ for the first one and of $\mathcal{A}_{\varphi_2}$ for the second one:

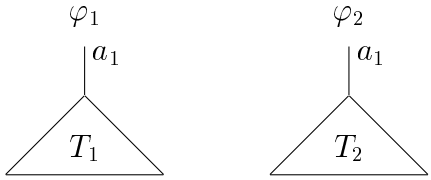


Therefore, we have that $L(\mathcal{A}_\varphi) = L(\mathcal{A}_{\varphi_1}) \cup L(\mathcal{A}_{\varphi_2})$. By induction hypothesis we thus have that $L_\varphi = L(\mathcal{A}_\varphi)$.

(3) If $\varphi = \varphi_1 \wedge \varphi_2$ then any run of $\mathcal{A}_\varphi$ has the following form:

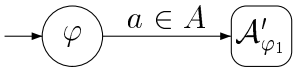


where a_1 is the first letter of the word and where the following runs are runs of $\mathcal{A}_{\varphi_1}$ for the first one and of $\mathcal{A}_{\varphi_2}$ for the second one:



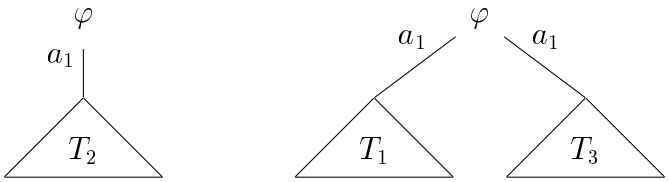
Therefore, we have that $L(\mathcal{A}_\varphi) = L(\mathcal{A}_{\varphi_1}) \cap L(\mathcal{A}_{\varphi_2})$. By induction hypothesis we thus have that $L_\varphi = L(\mathcal{A}_\varphi)$.

(4) If $\varphi = \mathbf{X}\varphi_1$ then $\mathcal{A}_\varphi$ has the following form:

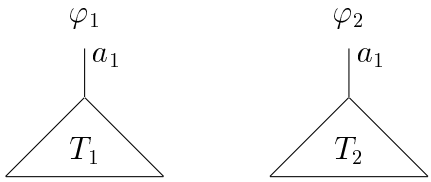


where $\mathcal{A}'_{\varphi_1}$ is equal to $\mathcal{A}_{\varphi_1}$ except that φ_1 is not an initial state. The outgoing transitions from φ to $\mathcal{A}'_{\varphi_1}$ go to the state φ_1.
Therefore, we have that $L(\mathcal{A}_\varphi) = \bigcup_{a \in A} aL(\mathcal{A}_{\varphi_1})$, and the induction hypothesis concludes this case: $L_\varphi = L(\mathcal{A}_\varphi)$.

(5) If $\varphi = \varphi_1 \,\mathbf{U}\, \varphi_2$. A run for $\mathcal{A}_\varphi$ on a non empty word u whose first letter is a_1 can have two different forms:

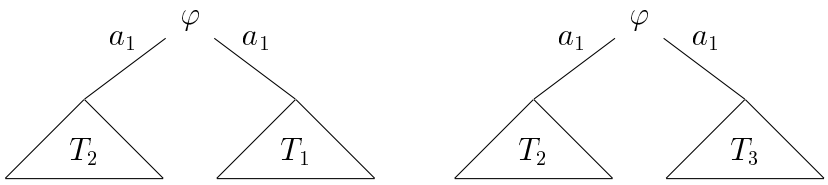


where T_3 is a run of $\mathcal{A}_\varphi$, and where the following runs are respectively a run of $\mathcal{A}_{\varphi_1}$ for the first one and a run of $\mathcal{A}_{\varphi_2}$ for the second one:

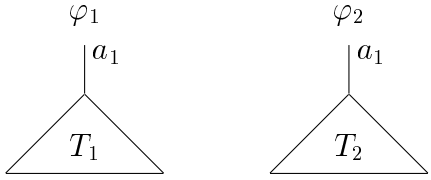


By induction hypothesis, the first run is an accepting run for u if and only if $(u,1) \models \varphi_2$ and the second one is accepting if and only if $(u,1) \models \varphi_1$ and T_3 is an accepting run of $\mathcal{A}_\varphi$. As the root of T_3 is $\varphi = \varphi_1 \mathbf{U} \varphi_2$ and as φ is not a final state this implies that T_3 cannot be reduced to its root. Therefore, by an easy induction on the length of u, it follows that u is recognized by $\mathcal{A}_\varphi$ if and only if there exists i, $1 \leq i \leq |u|$ such that for all j, $1 \leq j < i$ we have $(u,j) \models \varphi_1$ and $(u,i) \models \varphi_2$ (This means that we cannot have always the second kind of run). Therefore, u is recognized by $\mathcal{A}_\varphi$ if and only if $(u,1) \models \varphi$. This implies that $L_\varphi = L(\mathcal{A}_\varphi)$.

(6) If $\varphi = \varphi_1 \mathbf{R} \varphi_2$. A run for $\mathcal{A}_\varphi$ on a non empty word u whose first letter is a_1 can have two different forms:



where T_3 is a run of $\mathcal{A}_\varphi$, and where the following runs are respectively a run of $\mathcal{A}_{\varphi_1}$ for the first one and a run of $\mathcal{A}_{\varphi_2}$ for the second one:



By induction hypothesis, the first run is an accepting run for u if and only if $(u,1) \models \varphi_2$ and $(u,1) \models \varphi_1$. The second one is accepting if and only if $(u,1) \models \varphi_2$ and T_3 is an accepting run of $\mathcal{A}_\varphi$. As the root of T_3 is φ and as $\varphi = \varphi_1 \mathbf{R} \varphi_2$ is a final state this implies that T_3 can be reduced to its root. Therefore, by an easy induction on the length of u, it follows that u is recognized by $\mathcal{A}_\varphi$ if and only if one of the following cases is true

(i) For all i, $1 \leq i \leq |u|$, $(u,i) \models \varphi_2$.

(ii) There exists i, $1 \leq i \leq |u|$ such that for all j, $1 \leq j \leq i$ we have

$(u, j) \models \varphi_2$ and $(u, i) \models \varphi_1$ (The condition on φ_2 to be satisfied is released at position i as φ_1 is satisfied).

This exactly means that u is recognized by $\mathcal{A}_\varphi$ if and only if $(u, 1) \models \varphi$. This implies that $L_\varphi = L(\mathcal{A}_\varphi)$.

This induction proves that for any LTL-formula φ we have that $L_\varphi = L(\mathcal{A}_\varphi)$.

■

Acknowledgments

I gratefully acknowledge the many helpful suggestions of Anca Muscholl during the preparation of the paper. I also wish to express my thanks to Jean-Eric Pin for suggesting many stimulating ideas.

References

[1] M Arfi. Opérations polynomiales et hiérarchie de concaténation. *Theoretical Computer Science*, 91:71–84, 1991.

[2] A. Baziramwabo, P. McKenzie, and D. Therien. Modular temporal logic. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 344–351. IEEE, 1999.

[3] A. Bergeron and S. Hamel. Cascade decomposition are bit-vector algorithms. In *Implementation and Application of Automata, 6th International Conference, CIAA 2001, Pretoria, South Africa, July 23-25, 2001, Revised Papers*, volume 2494 of *Lecture Notes in Computer Science*, pages 13–26. Springer, 2002.

[4] A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–66, 2002.

[5] J. Cohen, D. Perrin, and J.-E. Pin. On the expressive power of temporal logic for finite words. *Journal of Computer and System Sciences*, 46:271–294, 1993.

[6] J.A. Kamp. *Tense Logic and the Theory of Linear Order.* Ph.d. thesis, University of California, Los Angeles, 1968.

[7] R. McNaughton and S. Papert. *Counter-free Automata.* MIT Press, 1971.

[8] J.-E. Pin. *Varieties of formal languages.* North Oxford, LondonPlenum, New-York, 1986. (Translation of Variétés de langages formels).

[9] J.-E. Pin. Syntactic semigroups. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*, volume 1, chapter 10, pages 679–746. Springer Verlag, 1997.

[10] Jean-Eric Pin and Pascal Weil. Polynomial closure and unambiguous product. *Theory Comput. Systems*, 30:1–39, 1997. version complète de [10].

[11] M.P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8:190–194, 1965.

[12] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.

[13] H. Straubing. Families or recognizable sets corresponding to certain varieties of finite monoids. *Journal of Pure and Applied Algebra*, 15:305–318, 1979.

[14] H. Straubing. *Finite automata, formal logic, and circuit complexity.* Birkhäuser, 1994.

[15] M. Y. Vardi. An automata-theoretic approach to linear-temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for concurrency*, number 1043 in Lecture Notes in Computer Science, pages 238–266. Springer, 1996.

[16] Th. Wilke. *Classifying discrete temporal properties.* Habilitation thesis, Kiel, Germany, 1998.

[17] Th. Wilke. Classifying discrete temporal properties. In *STACS 99*, number 1563 in Lecture Notes in Computer Science, pages 32–46, Berlin, 1999. Springer.