INFORMATIQUE THÉORIQUE ET APPLICATIONS

ROMAN R. REDZIEJOWSKI

Construction of a deterministic ω-automaton using derivatives

Informatique théorique et applications, tome 33, n° 2 (1999), p. 133-158

<http://www.numdam.org/item?id=ITA_1999__33_2_133_0>

Theoretical Informatics and Applications
Theoret. Informatics Appl. **33** (1999) 133–158

CONSTRUCTION OF A DETERMINISTIC ω-AUTOMATON USING DERIVATIVES

ROMAN R. REDZIEJOWSKI[1]

Abstract. A deterministic automaton recognizing a given ω-regular language is constructed from an ω-regular expression with the help of derivatives. The construction is related to Safra's algorithm, in about the same way as the classical derivative method is related to the subset construction.

AMS Subject Classification. 68Q45, 68R15.

1. INTRODUCTION

In 1964, Brzozowski [2] presented an elegant construction leading from a regular expression directly to a deterministic automaton recognizing the language denoted by that expression. The construction, based on the notion of a *derivative*, became one of the standard tools in the study of finite-state automata. A number of its applications and improvements is listed in the introduction to a recent paper [1].

The definition of a derivative is easily extended to ω-languages. But, the construction based on derivatives does not work for these languages. The automaton constructed from derivatives has one state for each distinct derivative. These states are usually too few for an acceptance condition expressed in terms of infinite repetition of states. For example, all derivatives of the language $X = (a \cup b)^* a^\omega$ are equal to X. The resulting automaton has only one state, allowing only two distinct acceptance conditions, that recognize only the languages $\emptyset$ and A^ω. This fact has been noticed in [5] and [9]; both papers end with an almost identical remark,

Keywords and phrases: Infinite word, omega-language, regular language, regular expression, rational language, rational expression, derivative, deterministic omega-automaton, Muller automaton.

[1] Ericsson Hewlett-Packard Telecommunications AB, Västberga Allé 9, S-12625 Stockholm, Sweden; e-mail: roman.redziejowski@ehpt.com

suggesting an acceptance condition in terms of transitions, rather than states. But this does not work either; the automaton constructed from derivatives has, in general, too few transitions, as well as too few states. The fact that there exist non-regular ω-languages with only finitely many distinct derivatives (see [11,12]) seems to further discourage this approach.

However, when we compute the derivatives of $X = (a \cup b)^* a^\omega$ in a formal way, we obtain the expression $(a \cup b)^* a^\omega \cup a^\omega$ for the derivative with respect to a. If we fail to notice that this is equal to X, we have two derivatives, and the resulting automaton has just enough states to formulate the required acceptance condition. Apparently, if we fail to notice equality of sufficiently many derivatives, we should always be able to obtain sufficiently many states. The construction presented here originates from this line of thought, but the result is not the expected automaton with one state per derivative. The states correspond instead to certain combinations of derivatives.

We start, in Section 2, by recalling the necessary notions about languages and automata. In Section 3, we establish the terminology concerning ordered binary trees. In Section 4, we reduce the problem of deciding whether a given infinite word belongs to a given language to the problem of deciding whether an ordered binary tree contains what we call a "live path". In Section 5, we develop an algorithm to solve the live-path problem and use it, in Section 6, as a base for constructing an automaton to recognize a given ω-regular language. The construction of the automaton is recapitulated in Section 7. Section 8 illustrates the construction on three examples, and Section 9 contains final remarks.

2. Languages and automata

We assume the reader to be familiar with the material reviewed in this section; the purpose is mainly to establish the terminology and notation. For a more thorough treatment, and an extensive bibliography, the reader is referred to the survey articles [6, 13, 14]. There is also a monograph in preparation, available as report [7].

An *alphabet* A is a finite nonempty set of *letters*. A sequence of letters from A is called a *word* (over A). A word can be finite or infinite, meaning a finite or infinite sequence of letters. The sequence of 0 letters is called the *empty word* and is denoted by λ. The set of all words is denoted by A^∞, the set of all finite words by A^*, the set of all finite words other than λ by A^+, and the set of all infinite words by A^ω. The concatenation of words $x \in A^*$ and $y \in A^\infty$ is denoted by xy. The concatenation of a sequence of words $x_1, x_2, x_3, \ldots$ where $x_i \in A^+$ for $i \geq 1$ is denoted by $x_1 x_2 x_3 \ldots$ The concatenation of infinitely many copies of a word $x \in A^+$ is denoted by x^ω.

Any subset of A^∞ is called a *language*.

We use these operations on languages:

union	$L_1 \cup L_2$	for $L_1 \subseteq A^\infty, L_2 \subseteq A^\infty$,
product	$L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$	for $L_1 \subseteq A^*, L_2 \subseteq A^\infty$,
star	$L^* = \lambda \cup L \cup L^2 \cup L^3 \cup \ldots$	for $L \subseteq A^*$,
omega	$L^\omega = \{x_1 x_2 x_3 \ldots \mid x_i \in L \text{ for } i \geq 1\}$	for $L \subseteq A^+$.

A *regular expression* (over alphabet A) is defined recursively as follows:

(1) each of symbols $\emptyset$, λ, and $a \in A$ is a regular expression.
(2) If X and Y are regular expressions, so are $X \cup Y$, XY, X^*, and (X).
(3) Nothing else is a regular expression unless its being so follows from a finite number of applications of (1) and (2).

(Many publications use the term "rational" instead of "regular" in this and the following definitions.)

A regular expression denotes a language: $\emptyset$ denotes the empty set, λ and $a \in A$ denote the singleton sets $\{\lambda\}$ and $\{a\}$; $X \cup Y$, XY, and X^* denote, respectively, the union, product, and star of languages denoted by X and Y; (X) denotes the same language as X. In the absence of parentheses, the operations are applied in the order: star, product, union.

An *ω-regular expression* is any expression of the form

$$\bigcup_{i=1}^{n} X_i Y_i^\omega = X_1 Y_1^\omega \cup X_2 Y_2^\omega \cup \ldots \cup X_n Y_n^\omega,$$

where $n \geq 1$, X_i is a regular expression, and Y_i is a regular expression denoting a language not containing λ, for $1 \leq i \leq n$. An ω-regular expression denotes the language obtained by applying the omega, product, and union (in this order) to the languages denoted by X_i and Y_i.

In the following, we write $|X|$ to mean the language denoted by a regular or ω-regular expression X. An *(ω-)regular language* is any language that can be denoted by an (ω-)regular expression.

The same language can usually be denoted by many different expressions. We say that two (ω-)regular expressions are *similar* if they can be transformed into each other using these rules:

$$X \cup X = X, \qquad X \cup \emptyset = \emptyset \cup X = X,$$
$$X \cup Y = Y \cup X, \qquad X\emptyset = \emptyset X = \emptyset,$$
$$(X \cup Y) \cup Z = X \cup (Y \cup Z), \qquad X\lambda = \lambda X = X.$$

One can easily see that similar expressions denote the same language. The point is that similarity of two expressions can always be effectively decided. Expressions that are not similar are called *dissimilar*. Dissimilar expressions may still denote the same language.

A *derivative* of an (ω-)regular expression X with respect to a word $w \in A^*$, denoted by $w^{-1}X$, is defined by these recursive rules:

$$a^{-1}\emptyset = a^{-1}\lambda = \emptyset, \qquad a^{-1}(XY) = (a^{-1}X)Y \cup o(X)a^{-1}Y,$$
$$a^{-1}a = \lambda, \qquad a^{-1}X^* = (a^{-1}X)X^*,$$
$$a^{-1}b = \emptyset, \qquad a^{-1}X^\omega = (a^{-1}X)X^\omega,$$
$$a^{-1}(X \cup Y) = a^{-1}X \cup a^{-1}Y, \qquad (wa)^{-1}X = a^{-1}(w^{-1}X),$$

where $a, b \in A$, $a \neq b$, and $o(X)$ denotes the language $|X| \cap \{\lambda\}$.

The expression $o(X)$ is obtained as follows:

$$o(a) = o(\emptyset) = \emptyset, \qquad o(XY) = o(X)o(Y),$$
$$o(\lambda) = \lambda, \qquad o(X^*) = \lambda,$$
$$o(X \cup Y) = o(X) \cup o(Y), \qquad o(X^\omega) = \emptyset.$$

A derivative of an (ω-)regular expression is also an (ω-)regular expression, and one can verify that $|w^{-1}X| = \{z \in A^\infty \mid wz \in |X|\}$. (In some publications, the derivative is referred to as the "left quotient", "left residual", or "state of X derived (or generated) by the word w". The notation is also varying; for example: D_wX in [2], $w\backslash X$ in [3], $\partial X/\partial w$, $\partial_w[X]$ or X_w in [4], X/w in [11, 12]. These terms and the notation are sometimes applied to the expression, and sometimes to the language denoted by it. The notation adopted here comes from [1, 6–8].)

The number of distinct derivatives of an (ω-)regular expression is potentially infinite: one for each word $w \in A^*$. But, the number of dissimilar derivatives is always finite. This result was proved in [2] for regular expressions. (It was shown there to hold even for a weaker definition of similarity, excluding the rules about $\emptyset$ and λ.) A rather obvious extension to ω-regular expressions was noted in [9].

Any finite initial portion of a word $x \in A^\infty$ is called a *prefix* of x. The set of all prefixes of words in a language $L \subseteq A^\infty$ is denoted by $\mathrm{pref}(L)$. One can easily see that $|w^{-1}X| \neq \emptyset$ if and only if $w \in \mathrm{pref}(|X|)$. A set of equalities, similar to those for $o(X)$, can be used to decide whether a derivative is empty.

We define a (deterministic) *finite-state automaton* informally, as a machine that can assume a finite number of distinct *states*. One state is identified as the *initial state*. The machine is started in the initial state and reads an infinite word $w \in A^\omega$, letter by letter. Each letter causes a *transition* to another, or possibly the same, state. The resulting state is determined uniquely by the current state and the letter. This is usually represented by a graph where nodes represent the states and directed edges represent the transitions, as shown in Figure 1. Each transition is labeled by the letters that cause it. The initial state is pointed to by an arrow.

As the automaton processes an infinite word w, it must visit one or more of its states infinitely many times. Let the set of such states be $\mathrm{In}(w)$. The automaton is said to *accept* the word w if $\mathrm{In}(w)$ satisfies a certain condition. One such condition, known as the *Muller acceptance condition*, is that $\mathrm{In}(w)$ is one of a given family

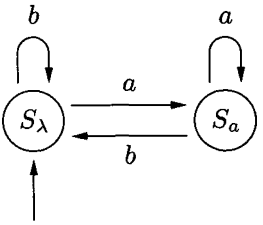

FIGURE 1.

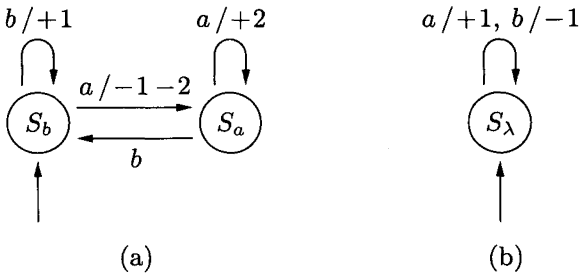

FIGURE 2.

T of sets of states. The automaton *recognizes* a language $L \subseteq A^\omega$ if it accepts exactly the words belonging to L.

The construction presented in this paper leads naturally to an alternative notion of acceptance, expressed in terms of transitions. To specify such acceptance, we place additional labels on the transitions; in Figure 2, they are shown following a slash.

The labels have the form $-n$ or $+n$, where n is a natural number. There may be several labels associated with the same letter.

When the automaton executes a transition in response to an input letter, it emits as output all the labels associated with that letter. For example, the automaton in Figure 2a emits -1 and -2 when going from S_b to S_a in response to input letter a. The automaton accepts w if, for some n, it emits $+n$ infinitely may times, and $-n$ only finitely many times. Thus, the automaton in Figure 2a accepts w if, after a finite number of letters, it keeps emitting $+1$, but not -1, or it keeps emitting $+2$, but not -2. Because of the topology of the transition graph, this happens to be equivalent to a Muller acceptance condition $\mathbf{T} = \{\{S_a\}, \{S_b\}\}$. But, such an equivalent condition does not exist in the general case. For example, the automaton in Figure 2b accepts w if, after a finite number of letters, it keeps emitting $+1$, but not -1. This is not equivalent to any Muller condition possible for this automaton.

Automata with Muller acceptance condition (shortly: Muller automata) are known to recognize exactly the class of ω-regular languages, in the sense that each language recognized by a Muller automaton is ω-regular, and for each ω-regular language there exists a Muller automaton recognizing that language. The same is true for automata with acceptance defined in terms of transitions (shortly: transition automata). Each Muller automaton can be transformed into a transition automaton recognizing the same language. To do this, assign distinct numbers to the members of $\mathbf{T}$. For a set $T \in \mathbf{T}$ numbered n, label transitions into each state $s \in T$ with $+n$, and transitions into each state $s \notin T$ with $-n$. Conversely, each transition automaton can be transformed into Muller automaton recognizing the same language. To do this, split each state so that there is a separate state for each combination of outputs for the incoming transitions. The members of $\mathbf{T}$ are sets of states containing at least one state with output $+n$, but none with $-n$, for some n.

An advantage of transition automata, as compared to Muller automata, is that they usually require fewer states to recognize a given language. Besides, one can transform the automaton (for instance, minimize the number of states), as long as its input-output relationship remains unchanged. The Muller automata, on the other hand, are simpler and well investigated. In Section 7, we indicate how to modify the construction to obtain a Muller automaton instead of a transition automaton.

3. Ordered binary trees

The reader should be familiar with the notion of a *tree* consisting of *nodes* and directed *edges*; we use without definition the related notions such as that of a *root* or *path*. To describe relations between nodes, we use the terms *parent*, *child*, *ancestor*, and *descendant*.

For a node x, the *subtree* with root x consists of the node x and all its descendants. A *partial tree* of a tree T is obtained from T by removing edges and nodes other than the root, in such a way that the result is still a tree.

All trees considered here are ordered binary trees, in the sense that all edges are classified into *left* and *right* edges. A node can be the origin of at most one left and at most one right edge. A node may be the origin of only one edge, either left or right, or of no edges at all. The node at the end of a left (or right) edge is called a left (respectively right) node, and a left (respectively right) child of its parent. A tree is *complete* if every node has both, left and right child. The trees considered here are normally infinite. By König's lemma, each such tree contains an infinite path from the root.

For $k \geq 1$, we define the k-th *level* of the tree as the set of nodes reached from the root by a path consisting of $k-1$ edges. If the number of nodes on the k-th level of a tree T has a highest value for $k = 1, 2, 3, \ldots$, we say that T has a *bounded width*. If no such highest value exists, we say that T has an *unbounded width*.

We define a *live path* to be any path from the root that contains infinitely many right edges. The reason for this definition will be clear in the next section.

In the usual graphical representation of an ordered binary tree, the left child with all its descendants is shown below and to the left of the parent, while the right child with all its descendants is below and to the right. This establishes a left-to-right ordering of nodes and paths that we shall exploit here; for this purpose, we define it more formally.

A path from the root is uniquely described by a sequence such as $lrllr$ or $rlrl\ldots$, where each letter corresponds to one edge, l indicating a left edge and r a right edge. We define the ordering $\prec$ of the paths as identical to the lexicographic order of these sequences, where $l \prec r$. To define this ordering for paths of different lengths, we assume the shorter path extended on the right with a suitable number of letters o, where $l \prec o \prec r$. We have thus, for example, $ll\ldots \prec ll \prec l \prec lr$. If $p \prec q$ holds for paths p, q, we say that p is *to the left of* q, and q is *to the right of* p; we also write $q \succ p$. We extend the ordering to nodes, defining $x \prec y$ (and $y \succ x$) if the path from the root to x is to the left of the path from the root to y.

The following result will be needed in the next section:

Lemma 3.1. *If a tree contains an infinite sequence of paths* $p_1 \prec p_2 \prec p_3 \prec \cdots$ *from the root, it contains a live path.*

(Proof is in the Appendix.)

4. Recognizing an ω-word

Let $X = \bigcup_{i=1}^{n} P_i Q_i^{\omega}$ be any ω-regular expression over an alphabet A. Suppose we are given a word $w \in A^{\omega}$. We are looking for a procedure to "decide" (in an infinite number of steps) if w belongs to the language $|X|$.

Let \$, called the *marker*, be an arbitrary symbol not in A. Define $X' = \bigcup_{i=1}^{n} P_i(\$Q_i)^{\omega}$. This is an ω-regular expression over the alphabet $A \cup \{\$\}$.

Proposition 4.1. *A word* $w \in A^{\omega}$ *belongs to* $|X|$ *if and only if one can insert into it infinitely many markers in such a way that each prefix of the resulting word* w' *belongs to pref*$(|X'|)$.

Proof. (1) The condition is sufficient: suppose the required word w' exists. Let $w_0, w_1, w_2, \ldots$ be words not containing \$, such that $w' = w_0\$w_1\$w_2\$\ldots$. It is easy to see that any word in pref$(|X'|)$ ending with \$ must belong to $|P_i(\$Q_i)^*\$|$ for some i. Because w' has infinitely many prefixes ending with \$, and the number of indices i is finite, there exists an index j, $1 \leq j \leq n$, such that infinitely many among these prefixes are in $|P_j(\$Q_j)^*\$|$. Given any $k \geq 1$, we can always choose a prefix $w_0\$w_1\$\ldots\$w_m\$ \in |P_j(\$Q_j)^*\$|$ with $m \geq k$. Because none of $w_0, w_1, \ldots, w_m$ contains \$, must be $w_0 \in |P_j|$ and $w_p \in |Q_j|$ for $1 \leq p \leq m$. Choosing k sufficiently large, one can show in this way that $w_p \in |Q_j|$ for all $p \geq 1$. Because $w = w_0 w_1 w_2 \ldots$, we have $w \in |P_j Q_j^{\omega}| \subseteq |X|$.

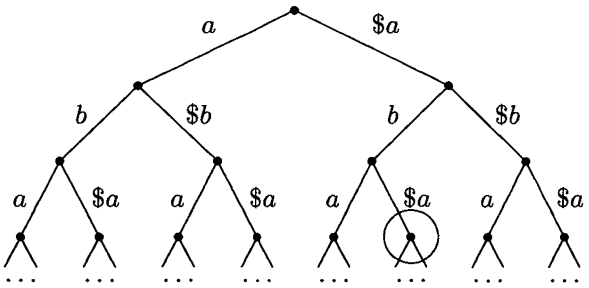

FIGURE 3.

(2) The condition is necessary: suppose $w \in |X|$. This means $w = pq_1q_2\ldots$ where $p \in |P_i|$ and $q_m \in |Q_i|$ for some i and all $m \geq 1$. The word $w' = p\$q_1\$q_2\$\ldots$ belongs to $|X'|$, and thus has the required property. □

Because $\lambda \notin |Q_i|$ for $1 \leq i \leq n$, the required word w' cannot have more than one marker before each letter. All possible ways of inserting markers before the letters of w can be represented by an ordered binary tree such as in Figure 3, which represents insertions of markers into the word $w = aba^\omega$. Each level of the tree represents a point in the word where we can insert a marker. A left edge represents the decision not to insert a marker at that point, and a right edge represents the decision to insert a marker.

Each node on a level $k \geq 1$ represents the result of $k-1$ such decisions: a prefix of a possible word w'. In the figure, that prefix is the word spelled out by symbols along the path from the root to the node. For example, the encircled node represents the prefix $\$ab\$a$.

Each infinite path from the root of the tree represents a possible word w'; in the figure, it is the word spelled out by symbols along the path. Each live path represents a possible word w' with infinitely many markers.

In order to exploit Proposition 4.1, we delete from the tree all nodes representing the words that are not in $\mathrm{pref}(|X'|)$. For this purpose, we label each node with the derivative of X' with respect to the word represented by that node. Taking as an example $X = (a \cup b)^*a^\omega$, we have $X' = (a \cup b)^*(\$a)^\omega$; the resulting labels for the tree of Figure 3 are shown in Figure 4.

The prefixes of w' that are not in $\mathrm{pref}(|X'|)$ are represented by nodes labeled with derivatives that denote $\emptyset$. We delete from the tree all such nodes, together with the adjacent edges. (Recall that we can always effectively decide which derivatives denote the empty language.) One can easily see that the result is still a tree, finite or not. We denote this tree by $T(w, X)$. The tree $T(w, X)$ for $w = aba^\omega$ and $X = (a \cup b)^*a^\omega$ is shown in Figure 5.

Proposition 4.2. *A word $w \in A^\omega$ belongs to $|X|$ if and only if $T(w, X)$ contains a live path.*

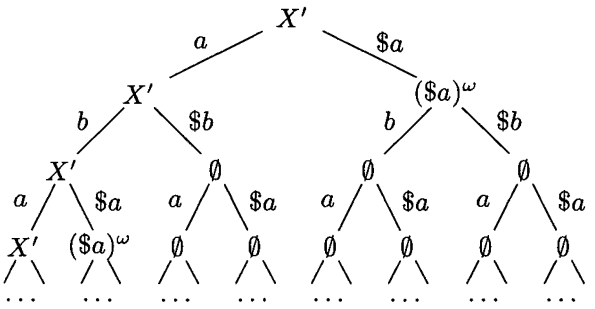

FIGURE 4.

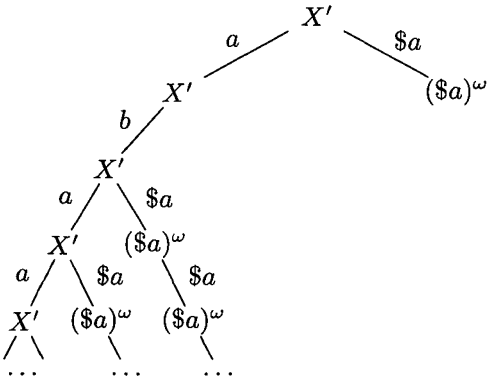

FIGURE 5.

Proof. (1) The condition is sufficient: suppose $T(w, X)$ contains a live path p. It represents a word w' obtained by inserting infinitely many markers into w. Each prefix v of w' is represented by a node along p. The fact that the node belongs to $T(w, X)$ means that $|v^{-1}X'| \neq \emptyset$ and thus $v \in \text{pref}(|X'|)$. By Proposition 4.1, we have $w \in |X|$.
(2) The condition is necessary: Suppose $w \in |X|$. By Proposition 4.1, one can insert into w infinitely many markers so that each prefix of the resulting word w' belongs to $\text{pref}(|X'|)$. The word w' is represented by a live path p in the complete tree representing all possible insertions of \$. As each prefix of w' belongs to $\text{pref}(|X'|)$, each node of p is labeled with a non-empty derivative, and is included in $T(w, X)$. □

If two nodes, x and y, on the same level of $T(w, X)$ are labeled with similar derivatives, the entire subtrees with roots x and y are isomorphic. This is so because similar labels denote the same language. The children of x and y in

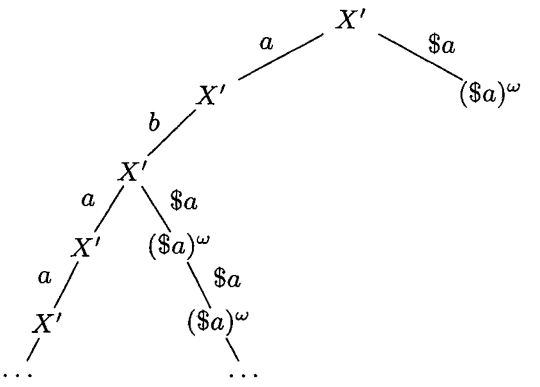

FIGURE 6.

the original complete tree are labeled with derivatives of the same language with respect to the same letter. These labels denote, respectively, the same languages (even if they are dissimilar). The same applies to children of the children, and so on. As we can always decide if a label denotes the empty language, the corresponding nodes are either both deleted, or both retained, in the construction of $T(w, X)$.

We can eliminate such redundant subtrees, and preserve the property stated by Proposition 4.2. Let us modify the tree $T(w, X)$ as follows. If two or more nodes on the same level are labeled with similar derivatives, remove all these nodes except the rightmost. Together with each node, remove the adjacent edges. The result is a partial tree of $T(w, X)$, in the following denoted by $\widehat{T}(w, X)$. The tree $\widehat{T}(w, X)$ for the preceding example is shown in Figure 6.

Proposition 4.3. *A word $w \in A^\omega$ belongs to $|X|$ if and only if $\widehat{T}(w, X)$ contains a live path.*

Proof. It is enough to show that $T(w, X)$ contains a live path if and only if $\widehat{T}(w, X)$ does.
(1) The condition is sufficient: any path in $\widehat{T}(w, X)$ is a path in $T(w, X)$.
(2) The condition is necessary: suppose $T(w, X)$ contains a live path p. Suppose p is not contained in $\widehat{T}(w, X)$ (for otherwise we are ready). Because the root is in $\widehat{T}(w, X)$, the initial portion of p must be in $\widehat{T}(w, X)$. Let this initial portion be called p_1. Let x be the last node of p_1. Let y be the next node in p, already outside $\widehat{T}(w, X)$. If y is not in $\widehat{T}(w, X)$, $\widehat{T}(w, X)$ must contain a node $z \succ y$, on the same level as y, and labeled with a similar derivative. Let $T(y)$ and $T(z)$ denote, respectively, the subtrees with roots y and z. The continuation of p from y is a live path in $T(y)$. As remarked before, $T(z)$ is isomorphic with $T(y)$. The corresponding path in $T(z)$ is also live; extended backwards to the root of $T(w, X)$, it gives a live path p' in $T(w, X)$. Suppose p' is not contained in $\widehat{T}(w, X)$

(for otherwise we are ready). Again, an initial portion of p' must be in $\widehat{T}(w, X)$. Call this initial portion p_2; it obviously contains z. Using the relationship between the nodes x, y, and z, one can verify that $p_1 \prec p_2$.

By repeating this procedure, we either arrive at a live path that is contained in $\widehat{T}(w, X)$, or continue indefinitely, obtaining a sequence of paths $p_1 \prec p_2 \prec p_3 \prec \ldots$, all contained in $\widehat{T}(w, X)$. In that case, $\widehat{T}(w, X)$ contains a live path by Lemma 3.1. □

Notice that the rule of always leaving the *rightmost* among the nodes with similar derivatives is essential. For example, leaving the leftmost nodes in the tree of Figure 5 instead of the rightmost produces a tree without a live path.

By our construction, the labels on each level of $\widehat{T}(w, X)$ are dissimilar. All of them are derivatives of the ω-regular expression X'. As X' has only finitely many dissimilar derivatives, $\widehat{T}(w, X)$ has a bounded width.

5. Detecting a live path

In the preceding section, we reduced the problem of deciding if $w \in |X|$ to the problem of deciding if the bounded-width tree $\widehat{T}(w, X)$ contains a live path. This is a general problem concerning ordered binary trees, so let us consider an arbitrary bounded-width tree T, and forget for a while the expressions at its nodes.

We need a method for detecting if one of many branching paths visits certain checkpoints infinitely often. Such a method is a central part of another construction of a deterministic automaton, invented by Safra [10]. The paths in that case are paths in an automaton. We borrow the idea from [10] and apply it to our tree.

The idea is to identify certain "green levels" as we proceed down a tree. The first green level is level 1. The next green level occurs as soon as every path from the last green level contains at least one right edge.

We present the algorithm as a process of marking up a graphical representation of T. This is only a way to explain and verify the algorithm; in the next section, we shall eliminate the tree altogether. We use a representation of T where the nodes on each level appear in the order $\prec$. The descendants of a contiguous sequence of nodes on one level form then a contiguous sequence on the next level. To remember which nodes are already reached *via* a right edge, we enclose each encountered right node in a pair of brackets, and then copy the brackets to the next level so that they enclose all descendants of the bracketed nodes. When every node is enclosed in brackets, we have reached a green level; we remove all brackets and repeat the process. This process is illustrated in Figure 7. The first encountered right node is x; we insert brackets around x, and then around its descendants on levels 3 and 4. After we have added brackets around the right node y on level 4, all nodes are in brackets, and we have reached a green level. We remove the brackets and start again. The next encountered right nodes are v and v; we have another green level when we reach t.

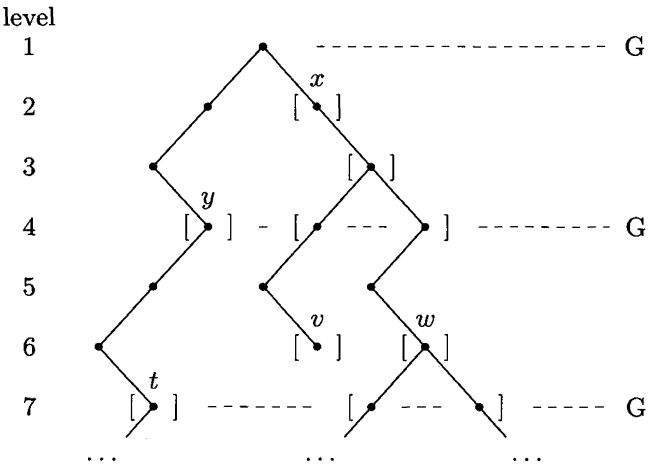

FIGURE 7.

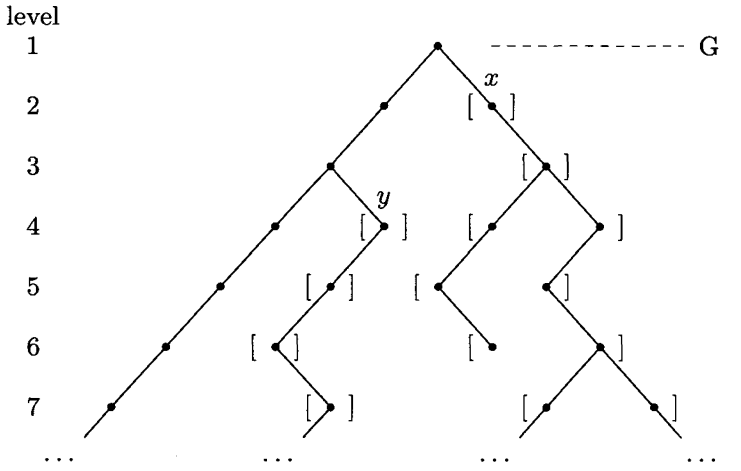

FIGURE 8.

Obviously, the tree has a live path if we can identify in this way infinitely many green levels. But the converse is not necessarily true. A tree has infinitely many green levels only if all infinite paths from the root are live. Thus, our process will never detect a live path in the tree of Figure 8 if the leftmost path consists entirely of left edges. We can go on inserting brackets around the subtrees with roots x and y, but the leftmost node on each level will never have brackets.

The solution is to repeat the process recursively for each bracketed subtree. A bounded-width tree can contain only finitely many infinite paths from the root,

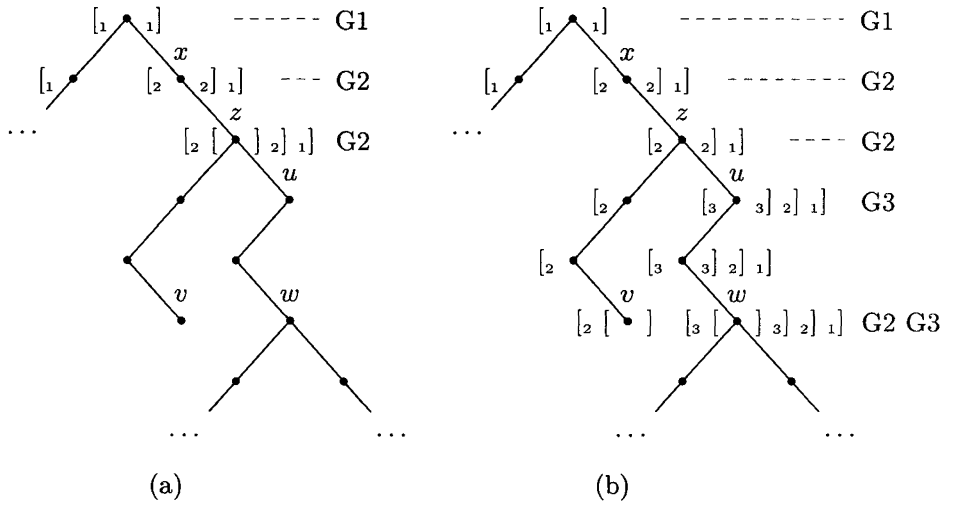

FIGURE 9.

and thus only finitely many infinite paths that are not live. If we proceed down a live path, we must eventually arrive at a subtree that does not contain any of them.

Because we want to identify the green levels separately for each subtree, we need to identify the brackets that enclose each of them. We do this by numbering the brackets. To have a uniform process, we also enclose the entire tree in brackets numbered 1.

The process for a part of the tree of Figure 8 is illustrated in Figure 9. Green levels of a subtree with brackets numbered n are annotated with Gn.

The first few steps are shown in Figure 9a. As soon as we encounter the right node x, we start looking for green levels in the subtree formed by its descendants. The subtree is enclosed in brackets numbered 2. Its first green level is that containing x itself. On the next level, the subtree has one right node, z. We enclose it in (yet unnumbered) brackets. Since this is the only node within the brackets numbered 2, the subtree 2 has here another green level. This is the situation in Figure 9a.

The subsequent steps are shown in Figure 9b. We remove the brackets between $[_2$ and $_2]$ and proceed to look for another green level of subtree 2. On the next level, the subtree has two nodes. One of them, u, is a right node, so we insert a pair of brackets around it. Since the other node is a left node, this is not a green level of subtree 2, and we have to start tracking the subtree of descendants of u. We assign number 3 to the brackets just inserted. We have here the first green level of subtree 3.

The situation within the subtrees 2 and 3 remains unchanged on the next level. On the level below it, we have two right nodes, v and w, and insert a pair of brackets around each. We have now a green level for both subtrees, 2 and 3. This is the situation in Figure 9b. The next step is to remove all brackets between $[_2$ and $_2]$; this removes all memory of the subtree 3, but there is no need to track that subtree any more.

If we use a new number for each subtree, the numbers will grow indefinitely. It will be practical to reuse the numbers "freed" by finished or abandoned subtrees. If we reuse the numbers, we have to distinguish green levels of different subtrees using the same number. To do this, we annotate the first green level of a subtree numbered n by $-n$, and the subsequent green levels by $+n$. The presence of a subtree with infinitely many green levels is then indicated by $-n$ appearing finitely many times, and $+n$ infinitely many times, for some number n.

In the following, we write $B(n,k)$ to denote the pair of brackets numbered n on level k. If a node x inside $B(n,k)$ is not enclosed in another pair nested in $B(n,k)$, we say that x *visible* in $B(n,k)$; otherwise we say that x is *hidden*. We say that a pair of brackets is *saturated* if all nodes enclosed in it are hidden. By *resetting* a pair of brackets we mean removing all brackets within it. The exact procedure for marking up the tree is:

Algorithm 5.1. *Enclose the root in brackets numbered 1, and annotate level 1 with -1. For $k = 1, 2, 3, \ldots$, mark up level $k+1$ in these four steps:*

(A1) enclose each right node on level $k+1$ in a pair of unnumbered brackets.

(A2) For each pair of brackets $B(n,k)$ on level k, enclosing at least one node with a child: insert a pair of brackets $B(n,k+1)$ on level $k+1$ enclosing exactly the children of nodes in $B(n,k)$, together with any brackets inserted around them by (A1). Insert the brackets so that they are nested in the same way as identically numbered brackets on level k.

(A3) Reset each saturated pair of brackets $B(n,k+1)$ on level $k+1$ that is not contained in another saturated pair. Annotate level $k+1$ with $+n$ for each pair thus reset.

(A4) Assign a distinct number to each pair of unnumbered brackets. Use the lowest natural numbers that did not appear on level $k+1$ after (A3). Annotate level $k+1$ with $-n$ for each number n thus assigned.

The result for the tree of Figure 8 is shown in Figure 10. Notice that the Algorithm did not produce annotation $+3$ on level 6 corresponding to green level G3 of Figure 9; this is the result of ignoring nested saturated pairs in (A3). Notice the reuse of number 3 at node w.

One can easily see that the unnumbered brackets inserted by (A1) cannot be saturated at (A3), so (A3) always resets numbered brackets. From (A3) and (A4) follows that each level annotated with n (meaning $+n$ or $-n$) contains brackets numbered n. Starting from a level k annotated with $-n$, brackets numbered n are propagated by (A2) to consecutive levels, enclosing a subtree with root at level k. This can terminate in one of two ways: either the subtree ends, or the

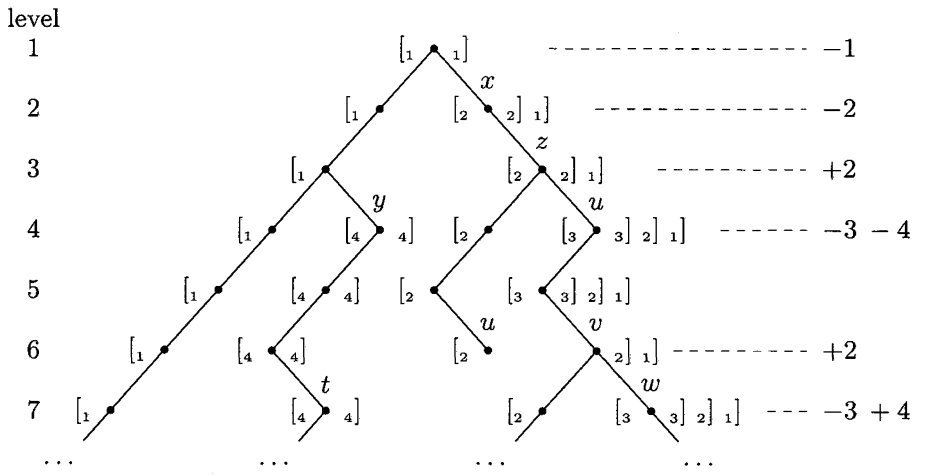

FIGURE 10.

brackets are removed by (A3). We note other facts about the marking produced by Algorithm 5.1:

Lemma 5.2. *Let k_1 be any level annotated with n. Let $k_2 > k_1$ be any level containing a pair of brackets numbered n, and such that none of the levels k, $k_1 < k \leq k_2$ is annotated with n. The path from level k_1 to any node x in $B(n, k_2)$ contains a right edge if and only if x is hidden in $B(n, k_2)$.*

Lemma 5.3. *Let k_1 be any level annotated with n. Let $k_2 > k_1$ be any level annotated with $+n$, such that none of the levels k, $k_1 < k < k_2$ is annotated with n. Each path from level k_1 to a node in $B(n, k_2)$ contains a right edge.*

Lemma 5.4. *The number of brackets on one level never exceeds the number of nodes on that level.*

The proofs are in the Appendix. We are now ready for the main result:

Proposition 5.5. *T contains a live path if and only if there exists a number n such that Algorithm 5.1 annotates infinitely many levels with $+n$, and only finitely many with $-n$.*

Proof. (1) The condition is sufficient: suppose the required number n exists. Let k be the last level annotated with $-n$. Let $T(k, n)$ be the subtree with root on level k, enclosed in brackets numbered n. Infinitely many levels after the k-th are annotated with $+n$, showing that the tree $T(k, n)$ is infinite. Let p be an infinite path from the root of $T(k, n)$. This path crosses infinitely many levels annotated with $+n$. By Lemma 5.3, p contains at least one right edge between each pair of consecutive levels annotated with $+n$. Thus, p contains infinitely many right edges. Extended back to the root of T, it is a live path in T.

(2) The condition is necessary: suppose T contains a live path p, but the required number n does not exist. That means, for each n, if only finitely many levels are annotated with $-n$, only finitely many levels are annotated with $+n$, and there exists the last level annotated with n.

The brackets numbered 1 are never removed, so only one level (the first) is annotated with -1. By our assumption, there exists the last level, k_1, annotated with 1. Let x_1 be any right node on p on a level $k > k_1$. By Lemma 5.2, x_1 is enclosed by one or more pairs of brackets within the pair $B(1, k)$. Let the number on the outermost such pair be n_1. The brackets numbered n_1 appear on each level below k: they are copied by (A2) because x_1 has a descendant on each level below k, and they are not removed by (A3) because no level below k is annotated with $+1$.

Because the brackets numbered n_1 are present on every level below k, (A4) can not annotate any of these levels with $-n_1$, so only finitely many levels are so annotated. By our assumption, there exists the last level, k_2, annotated with n_1. Let x_2 be any right node on p on a level $k > k_2$. The reasoning can be repeated to show that another pair of brackets, numbered n_2, is nested within the pair numbered n_1 on all subsequent levels.

This can be repeated indefinitely, showing that the number of brackets appearing on the same level increases without a bound as we proceed down the tree. Because T has a bounded width, the number of brackets on some level must exceed the number of nodes on that level. But, this is not possible according to Lemma 5.4. □

6. Back to recognizing an ω-word

We return now to the tree $\widehat{T}(w, X)$. The tree $\widehat{T}(w, X)$ of Figure 6, marked up according to Algorithm 5.1, is shown in Figure 11.

Let us list the symbols appearing on consecutive levels. For the tree of Figure 11, this gives:

level 1	$[_1\, X'\, {}_1]$	-1
2	$[_1\, X'\, [_2\, (\$a)^\omega\, {}_2]\, {}_1]$	-2
3	$[_1\, X'\, {}_1]$	
4	$[_1\, X'\, [_2\, (\$a)^\omega\, {}_2]\, {}_1]$	-2
5	$[_1\, X'\, [_2\, (\$a)^\omega\, {}_2]\, {}_1]$	$+2$
6	$[_1\, X'\, [_2\, (\$a)^\omega\, {}_2]\, {}_1]$	$+2$
	...	

These rows can be written without actually constructing the tree $\widehat{T}(w, X)$ and marking it up. Suppose first that $\widehat{T}(w, X)$ is the complete tree, such as in Figure 4. The first row is $[_1\, X'\, {}_1] - 1$. Each node on level $k \geq 1$ has two children, labeled,

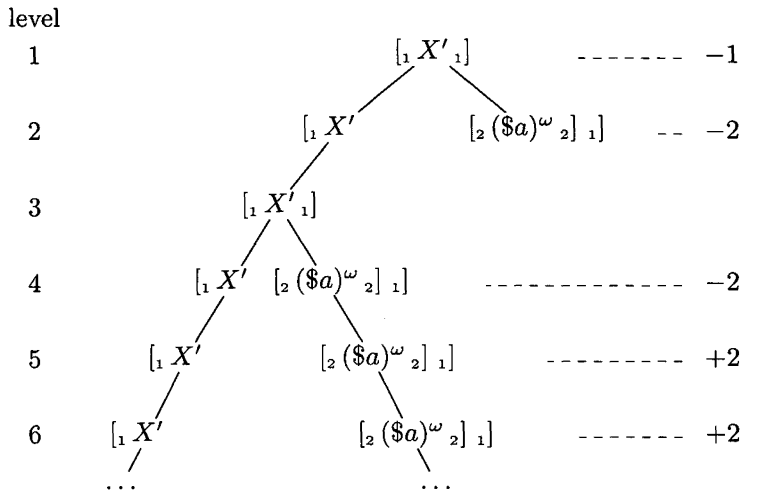

FIGURE 11.

respectively, with $w_k^{-1}d$ and $(\$w_k)^{-1}d$, where d is the label of the node and w_k is the k-th letter of w. A row representing level $k+1$ with markings produced by (A1) and (A2) is obtained by replacing each derivative d in row k by $w_k^{-1}d\ [\ (\$w_k)^{-1}d\ ]$.

In the construction of $T(w, X)$, we remove from the complete tree all nodes labeled with empty derivatives. This corresponds to deleting from the row all derivatives d denoting $\emptyset$, together with any enclosing brackets. In the construction of $\widehat{T}(w, X)$, we remove from $T(w, X)$ each derivative that has a similar derivative to the right of it. This corresponds to deleting from the row all such derivatives and any brackets that enclose them.

The reader may verify that the following algorithm faithfully reproduces the rows obtained for $\widehat{T}(w, X)$ marked up using Algorithm 5.1:

Algorithm 6.1. *Start with* $[_1 X'_1]\ -1$ *as row 1. For each* $k \geq 1$, *construct row* $k+1$ *by transforming row* k *in these four steps:*

(B1) replace each derivative d *in the row by* $w_k^{-1}d\ [\ (\$w_k)^{-1}d\ ]$, *where* w_k *is the* k*-th letter of* w.

(B2) Remove each derivative that denotes $\emptyset$ *or has a similar derivative to the right of it. Then, remove any empty pairs of brackets.*

(B3) Reset each saturated pair of brackets that is not contained in another saturated pair. Annotate the row with $+n$ *for each pair thus reset, where* n *is the number of the pair.*

(B4) Assign a distinct number to each pair of unnumbered brackets. Use the lowest natural numbers that did not appear in the row after (B3). Annotate the row with $-n$ *for each number* n *thus assigned.*

From Propositions 4.3 and 5.5 follows:

Proposition 6.2. *A word $w \in A^\omega$ belongs to $|X|$ if and only if there exists a number n such that Algorithm 6.1 produces infinitely many rows annotated with $+n$, and only finitely many annotated with $-n$.*

One can imagine a machine that computes consecutive rows according to Algorithm 6.1 as it reads the word w. By watching the annotations $-n$ and $+n$, one can decide if $w \in |X|$.

But, the machine does not need to perform any computations. Define the "state" to be the part of a row between, and including, the brackets $[_1 \ _1]$. Define the "output" to be the part consisting of the annotations $-n$ and $+n$. At each stage, the next state and the output are determined by the current state and the next letter of w.

Each state is a sequence of dissimilar derivatives and numbered brackets. Let the number of dissimilar derivatives of X' be k. By Lemma 5.4, the state can contain at most k pairs of brackets. Hence, (B4) can never assign a number higher than k to a pair of brackets. The number of possible distinct states for given X and all possible words $w \in A^\omega$ is thus finite. Given an expression X, one can compute the next state and the output once for all possible states and input letters $a \in A$, and program the machine to use these results. Such a machine is nothing else but a finite-state automaton recognizing the language $|X|$. In the next section, we recapitulate the construction of that automaton.

To simplify the presentation, we have ignored so far the case when the tree $\widehat{T}(w, X)$ is finite. The tree being finite means that w has a prefix not belonging to $\mathrm{pref}(|X|)$, and thus itself cannot belong to $|X|$. Algorithm 6.1 applied to a finite tree produces, at some stage, an empty row. Applied to such an empty row, the Algorithm produces another empty row.

7. The construction

Let $X = \bigcup_{i=1}^{n} P_i Q_i^\omega$ be a given ω-regular expression over alphabet A. To construct a finite-state automaton recognizing the language $|X|$, construct the expression $X' = \bigcup_{i=1}^{n} P_i(\$Q_i)^\omega$, where \$ is a new letter not in A. Compute all dissimilar derivatives of X' and identify those equal to $\emptyset$.

States

The states of the automaton are named by distinct sequences consisting of derivatives of X' and brackets numbered with integers between 1 and k, where k is the number of nonempty dissimilar derivatives. We do not define the well-formed state names because the procedure for construction of all accessible states will produce only well-formed names. Notice that empty sequence is a well-formed state name.

Initial state

The initial state is $[_1\, \lambda^{-1}X'_1] = [_1\, X'_1]$.

Next state

For a state s and an input letter $a \in A$, the next state is obtained as follows:

(N1) replace each derivative d appearing in s by $a^{-1}d$ [$(\$a)^{-1}d$].

(N2) Remove each derivative that denotes $\emptyset$ or has a similar derivative to the right of it. Then, remove any empty pairs of brackets.

(N3) Reset each saturated pair of brackets that is not contained in another saturated pair. (A pair of brackets is saturated if every derivative within it is enclosed in additional brackets. Resetting a pair means removing all brackets within it.)

(N4) Assign a distinct number to each pair of unnumbered brackets. Use the lowest natural numbers that did not appear in the state after (N3).

Output

The output from the transition defined by (N1-N4) is a (possibly empty) collection of positive or negative integers defined as follows:

(O1) produce output $+n$ for each saturated pair of brackets reset according to (N3), where n is the number of that pair.

(O2) Produce output $-n$ for each number n assigned by (N4).

Accessible states

To construct all accessible states, start with initial state and construct states reached after all sequences of 1, 2, 3, ..., etc. input letters until no new states are obtained.

Acceptance condition

One can easily see that the automaton thus constructed implements the machine described in the preceding section. (The only difference is that we omitted output -1 at the first line.) From Proposition 6.2 follows that a word $w \in A^\omega$ belongs to $|X|$ if and only if it causes the automaton to output $+n$ infinitely many times, and $-n$ only finitely many times, for some number n.

Constructing a Muller automaton

If you insist on constructing a Muller automaton rather than a transition automaton, remove (O2), consider the whole row, together with the $+n$ annotations, as a state name, and change (N4) to:

(M4) Assign a distinct number to each pair of unnumbered brackets. Use the lowest natural numbers that did not appear in the state before (N1).

The result of this change is that brackets numbered n disappear from the state before n is reused. The acceptance condition $\mathbf{T}$ consists of subsets of states T

such that for some n, all states in T include brackets numbered n, and at least one state includes $+n$.

8. Examples

We illustrate the construction of the automaton on three examples. The first is the same as in Sections 4 and 6. The other two are borrowed from [7], Section I.8.

Example 1

$X = (a \cup b)^* a^\omega$. In this case, $X' = (a \cup b)^*(\$a)^\omega$. We start by computing the derivatives of X' that will be used in the construction. Denoting the derivative $w^{-1}X'$ by D_w, we have:

$$\begin{array}{lll} D_\lambda = X', & D_{\$a} = (\$a)^\omega, & D_{\$ab} = \emptyset, \\ D_a = D_\lambda, & D_{\$b} = \emptyset, & D_{\$a\$a} = D_{\$a}, \\ D_b = D_\lambda, & D_{\$aa} = \emptyset, & D_{\$a\$b} = \emptyset. \end{array}$$

Denote by S_w the state reached after an input word $w \in A^*$. The initial state is $S_\lambda = [_1\, X'\, _1] = [_1\, D_\lambda\, _1]$. We proceed to construct transitions (the omitted steps are void).

- Letter a applied to state S_λ:
  - (N1) replace D_λ by $D_a\,[\,D_{\$a}\,]$, obtaining $[_1\, D_\lambda\,[\,D_{\$a}\,]\, _1]$.
  - (N4) Assign number 2 to brackets, obtaining $S_a = [_1\, D_\lambda\, [_2\, D_{\$a}\, 2]\, _1]$.
  - (O2) Output -2.
- Letter b applied to state S_λ:
  - (N1) replace D_λ by $D_b\,[\,D_{\$b}\,]$, obtaining $[_1\, D_\lambda\,[\,\emptyset\,]\, _1]$.
  - (N2) Remove $[\,\emptyset\,]$, obtaining $[_1\, D_\lambda\, _1] = S_\lambda$.
- Letter a applied to state S_a:
  - (N1) replace D_λ and $D_{\$a}$ by $D_a\,[\,D_{\$a}\,]$ and $D_{\$aa}\,[\,D_{\$a\$a}\,]$, obtaining $[_1\, D_\lambda\,[\,D_{\$a}\,]\,[_2\,\emptyset\,[\,D_{\$a}\,]\, _2]\, _1]$.
  - (N2) Remove the first $[\,D_{\$a}\,]$ and $\emptyset$, obtaining $[_1\, D_\lambda\, [_2\, [\,D_{\$a}\,]\, _2]\, _1]$.
  - (N3) Remove brackets within $[_2\ _2]$, obtaining $[_1\, D_\lambda\, [_2\, D_{\$a}\, 2]\, _1] = S_a$.
  - (O1) Output $+2$.
- Letter b applied to state S_a:
  - (N1) replace D_λ and $D_{\$a}$ by $D_b\,[\,D_{\$b}\,]$ and $D_{\$ab}\,[\,D_{\$a\$b}\,]$, obtaining $[_1\, D_\lambda\,[\,\emptyset\,]\,[_2\,\emptyset\,[\,\emptyset\,]\, _2]\, _1]$.
  - (N2) Remove $[\,\emptyset\,]$ and $[_2\,\emptyset\,[\,\emptyset\,]\, _2]$, obtaining $[_1\, D_\lambda\, _1] = S_\lambda$.

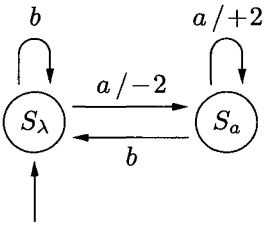

FIGURE 12.

As no new states were reached from S_a, we have two states, S_λ and S_a, and four transitions:

state +	letter	⇒ state		output
S_λ	a	$[_1\, D_\lambda\, [_2\, D_{\$a}\, {}_2]\, {}_1]$	$= S_a$	-2
S_λ	b	$[_1\, D_\lambda\, {}_1]$	$= S_\lambda$	
S_a	a	$[_1\, D_\lambda\, [_2\, D_{\$a}\, {}_2]\, {}_1]$	$= S_a$	$+2$
S_a	b	$[_1\, D_\lambda\, {}_1]$	$= S_\lambda$.	

The automaton recognizing $|(a \cup b)^* a^\omega|$ is shown in Figure 12. Notice that this is not the minimal automaton recognizing this language: the automaton of Figure 2b recognizes the same language using only one state.

EXAMPLE 2

$X = (a \cup b)^* ba^\omega$. We have $X' = (a \cup b)^* b(\$a)^\omega$. The derivatives needed for the construction are:

$$
\begin{aligned}
&D_\lambda = X', && D_{ba} = D_\lambda, && D_{b\$aa} = \emptyset,\\
&D_a = D_\lambda, && D_{bb} = D_b, && D_{b\$ab} = \emptyset,\\
&D_b = X' \cup (\$a)^\omega, && D_{b\$a} = (\$a)^\omega, && D_{b\$a\$a} = D_{b\$a},\\
&D_{\$a} = \emptyset, && D_{b\$b} = \emptyset, && D_{b\$a\$b} = \emptyset,\\
&D_{\$b} = \emptyset, &&&&
\end{aligned}
$$

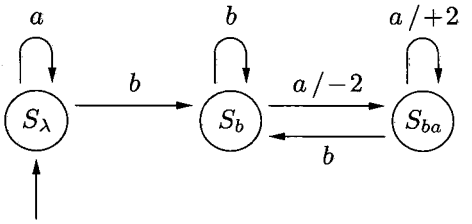

FIGURE 13.

where, as before, D_w stands for the derivative $w^{-1}X'$. The transitions are:

state	+	letter	⇒	state		output
S_λ		a		$[_1 D_{\lambda\,1}]$	$= S_\lambda$	
S_λ		b		$[_1 D_{b\,1}]$	$= S_b$	
S_b		a		$[_1 D_\lambda [_2 D_{b\$a\,2}]_{\,1}]$	$= S_{ba}$	-2
S_b		b		$[_1 D_{b\,1}]$	$= S_b$	
S_{ba}		a		$[_1 D_\lambda [_2 D_{b\$a\,2}]_{\,1}]$	$= S_{ba}$	$+2$
S_{ba}		b		$[_1 D_{b\,1}]$	$= S_b$.	

The automaton recognizing $|(a \cup b)^* ba^\omega|$ is shown in Figure 13.

EXAMPLE 3

$X = ((b \cup c)^* a \cup b)^\omega$. We have $X' = (\$((b \cup c)^* a \cup b))^\omega$. The transitions are:

state	+	letter	⇒	state		output
S_λ		a		$[_1 D_{\lambda\,1}]$	$= S_\lambda$	$+1$
S_λ		b		$[_1 D_{\$b\,1}]$	$= S_b$	$+1$
S_λ		c		$[_1 D_{\$c\,1}]$	$= S_c$	$+1$
S_b		a		$[_1 D_{\lambda\,1}]$	$= S_\lambda$	$+1$
S_b		b		$[_1 D_{\$b\,1}]$	$= S_b$	$+1$
S_b		c		$[_1 D_{\$c\,1}]$	$= S_c$	$+1$
S_c		a		$[_1 D_{\lambda\,1}]$	$= S_\lambda$	
S_c		b		$[_1 D_{\$c\,1}]$	$= S_c$	
S_c		c		$[_1 D_{\$c\,1}]$	$= S_c$,	

where, as before, D_w stands for the derivative $w^{-1}X'$. The automaton recognizing $|((b \cup c)^* a \cup b)^\omega|$ is shown in Figure 14.

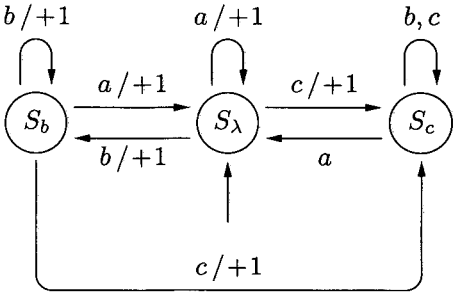

FIGURE 14.

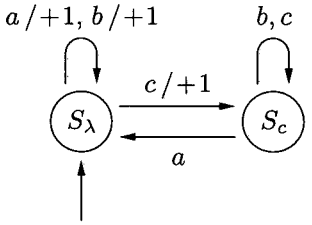

FIGURE 15.

The states S_b and S_λ are equivalent in the sense of the next state and output, so the above automaton is equivalent to that shown in Figure 15.

9. FINAL REMARKS

Because of the same basic idea for detecting a live path, our construction is in many ways similar to Safra's. Expressions with nested brackets are essentially tree structures. Our states are thus trees of derivatives. The Safra's states are trees of subsets of states of a non-deterministic automaton. This relationship is similar to that between Brzozowski's derivative method (where states are derivatives) and the subset construction (where states are subsets of states of a non-deterministic automaton). But, there is no exact correspondence between our states and Safra's states. One difference is that all derivatives within a state are ordered. Using this order, we eliminate similar derivatives anywhere in the state, while Safra's algorithm eliminates duplicate states only within the same level of a tree-state. This seems to somewhat reduce the number of distinct states. (We could further improve on this by eliminating derivatives that denote a subset of a language denoted by any derivative on the right.) Some simplification comes also from the acceptance condition expressed in terms of transitions, rather than states. The difference may be illustrated by the fact that the automaton for the language of

Example 3, constructed in [7] by Safra's method, has originally five states. However, these minor improvements do not seem to significantly reduce the complexity $2^{O(n \log n)}$ of Safra's construction (which, according to a result quoted in [10], constitutes the lower bound).

An advantage of the described construction is that it goes directly from an expression to a deterministic automaton, without an intermediate step of constructing a non-deterministic automaton. Also, manipulating states in the form of strings of symbols is simpler than manipulating Safra's tree-states, especially if the computations are done by hand. This can be again seen by comparison with examples in [7].

Brzozowski's derivative method for regular expressions has two important properties. First, it can produce a unique, minimal, automaton for the given language, not depending on the expression used to denote that language. (This happens when all derivatives used in the construction denote distinct languages.) Second, it can be applied to extended regular expressions, that allow intersection and complementation in addition to union, product, and star. (It was, in fact, presented so in [2].)

Unfortunately, none of these properties is shared by the construction presented here. The resulting automaton depends strongly on the expression, and need not have the minimal number of states. (A unique minimal automaton does not in general exist for an ω-regular language.) Complementation and intersection can probably be used within the regular expressions P_i and Q_i in $X = \bigcup_{i=1}^{n} P_i Q_i^{\omega}$, but Proposition 4.1 breaks down if X does not have the stated form.

Appendix

Proof of Lemma 3.1

Suppose a tree T contains the required sequence of paths. Notice that $p \prec q$ implies $p \neq q$, so all paths p_i for $i \geq 1$ are distinct.

Suppose a node x belongs to infinitely many among the paths p_i. Denote the set of these paths by P. Suppose the left child of x does not belong to some path $p_j \in P$. That means p_j either ends at x or proceeds to the right child of x. In each case, p_j is to the right of any path that proceeds from x to the left child. The path p_j is to the right of exactly $j-1$ among the paths p_i, namely those with $i < j$; that means at most finitely many paths in P proceed from x to the left child. As all paths p_i are distinct, at most one can end at x; hence, all paths in P but finitely many proceed to the right child of x. In other words:

(*) if a node belongs to infinitely many among the paths p_i then either all of them proceed to the left child, or all but finitely many proceed to the right child.

To find a live path in T, start with the root. From each node, proceed to the child that belongs to infinitely many among the paths p_i. Call the resulting path p. Since the root belongs to infinitely many among the paths p_i, (*) guarantees that so does each node encountered on the way; hence p is infinite.

Suppose now that p contains only finitely many right edges. Then, from some node x on, all edges of p are left edges. But then, according to (*), every node on p after x belongs to exactly the same paths p_i as x, meaning all these paths are identical; this contradicts the assumption that all of them are distinct.

Proof of Lemma 5.2

Consider a fixed level k_1 annotated with n. The proof is by induction on k_2.
(1) Induction base: $k_2 = k_1 + 1$. Consider any node x in $B(n, k_2)$. Because level k_2 is not annotated with $-n$, the pair $B(n, k_2)$ was inserted by (A2) as a copy of pair $B(n, k_1)$ containing the parent y of x. Because level k_1 is annotated with n, y is visible in $B(n, k_1)$: the pair was either inserted by (A1) and numbered by (A4), or reset by (A3), in the process of marking the level k_1. If x is the left child of y, no brackets are inserted around it by (A1) in the process of marking the level k_2, and x is visible in $B(n, k_2)$. If x is the right child, (A1) inserts a pair B of unnumbered brackets around it. (A2) inserts $B(n, k_2)$ immediately around B. (A3) can remove B only by resetting $B(n, k_2)$ or some brackets around $B(n, k_2)$. It did not reset $B(n, k_2)$ because the level is not annotated with $+n$. It did not reset any containing brackets because this would remove $B(n, k_2)$. Hence, the brackets B (with a number added by (A4)) remain around x, and x is hidden in $B(n, k_2)$. In each case, the Lemma holds for x.
(2) Induction step: suppose the Lemma holds for some level $k_2 > k_1$. Suppose level $k = k_2 + 1$ is not annotated with n and contains the pair $B(n, k)$. Consider any node x in $B(n, k)$. As before, level k_2 must contain the parent y of x, enclosed in brackets $B(n, k_2)$. Two situations are possible:

- the path from level k_1 to y does not contain any right edge. Then, by inductive assumption, y is visible in $B(n, k_2)$. In the same way as before, we can verify that x becomes hidden in $B(n, k)$ if and only if the step from y to x adds a right edge to the path.
- The path from level k_1 to y contains at least one right edge. Then, by inductive assumption, y is hidden in $B(n, k_2)$. The outermost pair of brackets enclosing y within $B(n, k_2)$ is copied by (A2) to become the outermost pair enclosing x within $B(n, k)$, and is not removed by (A3) for the same reason as before.

Proof of Lemma 5.3

Let k_1 and k_2 be as stated. The reasoning in the proof of Lemma 5.2 shows that the situation stated by that Lemma holds before step (A3) for level k_2, even if level k_2 is annotated with $+n$. The fact that k_2 is annotated with $+n$ means that $B(n, k_2)$ was saturated before (A3), that is, every node within $B(n, k_2)$ was hidden. By the preceding remark, each path from level k_1 to such a node contains a right edge.

Proof of Lemma 5.4

Suppose there are more pairs of brackets than nodes. Each node is visible in exactly one pair of brackets. That means, at least one pair contains no visible nodes. By definition, such a pair is saturated. But, (A3) does not leave any saturated pair, and (A4) does not introduce any.

The author thanks two anonymous referees for a number of useful suggestions.

REFERENCES

[1] V. Antimirov, Partial derivatives of regular expressions and finite automata constructions. In *STACS 95*, E.W. Mayr and C. Puech, Eds., Springer-Verlag (1995) 455–466.
[2] J.A. Brzozowski, Derivatives of regular expressions. *J. Assoc. Comput. Mach.* **11** (1964) 481–494.
[3] J.A. Brzozowski and E. Leiss, On equations for regular languages, finite automata, and sequential networks. *Theoret. Comput. Sci.* **10** (1980) 19–35.
[4] J.H. Conway, *Regular Algebra and Finite Machines.* Chapman and Hall (1971).
[5] D. Park, Concurrency and automata on infinite sequences, in *Proc. 5th GI Conference, Karlsruhe*, Springer-Verlag, Lecture Notes in Computer Science **104** (1981) 167–183.
[6] D. Perrin, Finite automata, in *Handbook of Theoretical Computer Science*, J. van Leeuven, Ed., **B**, Elsevier Science Publishers (1990) 1–57.
[7] D. Perrin and J.-E. Pin, Mots infinis. Internal report LITP 93.40, Laboratoire Informatique Théorique et Programmation, Institut Blaise Pascal, 4 Place Jussieu, F-75252 Paris Cedex 05 (1993).
[8] J.-E. Pin, *Varieties of Formal Languages.* North Oxford Academic (1986).
[9] R.R. Redziejowski, The theory of general events and its application to parallel programming. Technical paper TP 18.220, IBM Nordic Laboratory, Lidingö, Sweden (1972).
[10] S. Safra, On the complexity of ω-automata, in *Proc. 29th Annual Symposium on Foundations of Computer Science IEEE* (1988) 319–327.
[11] L. Staiger, Finite-state ω-languages. *J. Comput. System Sci.* **27** (1983) 434–448.
[12] L. Staiger, The entropy of finite-state ω-languages. *Problems of Control and Information Theory* **14** (1985) 383–392.
[13] L. Staiger, ω-languages. In *Handbook of Formal Languages*, G. Rozenberg and A. Salomaa, Eds., **3**, Springer-Verlag (1997) 339–387.
[14] W. Thomas, Automata on infinite objects, in *Handbook of Theoretical Computer Science*, J. van Leeuven, Ed., **B**, Elsevier Science Publishers (1990) 133–191.

Communicated by J.-E. Pin.
Received October, 1997. Accepted January, 1999.