Recognizing Regular Expressions by means of Dataflow Networks

Pascal Raymond
email: Pascal.Raymond@imag.fr

VERIMAG *
Miniparc ZIRST, 38330 Montbonnot-S$^{\mathrm{t}}$ Martin, France

Abstract. This paper addresses the problem of building a Boolean dataflow network (sequential circuit) recognizing the language described by a regular expression. The main result is that both the construction time and the size of the resulting network are linear with respect to the size of the regular expression.

Introduction

"Grep" machine: Let Σ be a vocabulary, L be a regular language on Σ. A "grep" machine is a machine receiving a sequence $s_0, s_1, \ldots, s_n, \ldots$ of symbols ($s_i \in \Sigma$) and computing a sequence $b_0, b_1, \ldots, b_n, \ldots$ of Booleans, such that b_n is true if and only if the word $s_0 s_1 \ldots s_n$ belongs to L[2].

This paper addresses the problem of building a "grep" machine for languages described by regular expressions. This problem is rather classical [4, 11, 10, 3, 1, 2]. We propose a solution which, to our knowledge, is new: Informally, it consists of building, from a regular expression E, a "circuit" (or Boolean data-flow network) exploring all the branches of a non-deterministic automaton recognizing $L(E)$. The relations between regular languages and sequential circuits are also studied [5, 7]. But the important point here, is that the automaton is never explicit (it is represented implicitly by a system of Boolean equations) nor deterministic. As a consequence, both the construction time and the size of the circuit are linear with respect to the size of the expression E.

Our method is presented in a general framework based on linear systems of equations. We first review the classical methods (deterministic finite automaton, non-deterministic finite automaton without or with empty-labelled transitions), by expressing them in this framework. We show that this classification coincides with the classes of complexity (exponential, quadratic and linear).

---

* Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université J. Fourier and VERILOG S.A. associated with IMAG.

[2] The "grep L" unix command is slightly different: in this case, b_n is true if and only if it exists some i such that $s_i s_{i+1} \ldots s_n$ belongs to L, but this problem is equivalent to the initial one with $L' = \Sigma^* L$

1 Regular expressions

1.1 Regular languages

Let Σ be a finite set of symbols; Σ^* denotes the set of all finite strings of symbols in Σ. The empty string (string of length 0) in Σ^* is denoted by ε. A language over Σ is any subset of Σ^*. The empty language is denoted by $\emptyset$. The set of regular languages over Σ is the least class of languages containing all the finite languages over Σ, and closed by union, concatenation (denoted by $\cdot$) and Kleene star operator (denoted by $*$). Any regular language can be denoted by a regular expression:

$$E ::= \emptyset \mid a \mid \; E \cdot E \mid E + E \mid E^*$$

The semantics of regular expressions is given by a function L which associates to each expression E the language described by E[3]:

$$\begin{array}{c} L(\emptyset) \; = \; \emptyset \quad L(a) \; = \; \{a\} \quad L(E+F) \; = \; L(E) \cup L(F) \\ L(E \cdot F) \; = \; L(E) \cdot L(F) \quad L(E^*) \; = \; \mu\ell.\{\varepsilon\} \cup (\ell \cdot L(E)) \end{array}$$

We note $\simeq$ the semantic equivalence over regular expressions, i.e.:

$$E \simeq F \Leftrightarrow L(E) = L(F)$$

1.2 More "useful" regular expressions

We are interested in "non trivial" regular languages, i.e. any language $\mathcal{L}$ which is neither empty ($\mathcal{L} \neq \emptyset$) nor reduced to the empty string ($\mathcal{L} \neq \{\varepsilon\}$). We will then use a more useful syntax, which forbids the representation of trivial languages:

$$E ::= a \mid \; E \cdot E \mid E + E \mid E^* \mid E^{\varepsilon}$$

The semantic of the power-ε operator is given by: $L(E^{\varepsilon}) \; = \; \varepsilon \cup L(E)$.

We note $\mathcal{R}exp$ the set of regular expressions, and we also define a function $\Lambda : \mathcal{R}exp \rightarrow \mathbb{B} = \{\mathit{true}, \mathit{false}\}$, such that $\Lambda(E)$ is true if and only if the empty string belongs to $L(E)$.

$$\begin{array}{c} \Lambda(a) \; = \; \mathit{false} \quad \Lambda(E^{\varepsilon}) \; = \; \mathit{true} \quad \Lambda(E^*) \; = \; \mathit{true} \\ \Lambda(E+F) \; = \; \Lambda(E) \vee \Lambda(F) \quad \Lambda(E \cdot F) \; = \; \Lambda(E) \wedge \Lambda(F) \end{array}$$

[3] $\mu\ell.f(\ell)$ denotes the least fixpoint of the equation $\ell = f(\ell)$

2 Linear systems of equations

In this section we show that the problem of building a grep machine from a regular expression can be expressed as follows: build a system of language equations of the form "$X = t_1 + ... + t_n$", where each t_i is a language term of a "special" form. The various known methods for building grep machines can be related to the form of the terms. This classification is largely inspired by [5, 6].

In this paper, we use the following notations: let θ be a system of equations, we note $\mathcal{V}(\theta)$ the set of language identifiers defined in θ, $\mathcal{A}(\theta) \in \mathcal{V}(\theta)$ the main language (the axiom), and for each $X \in \mathcal{V}(\theta)$, $\theta(X)$ the set of terms t_i defining X in θ. We note $\mathcal{L}(\theta)$ the solution (if it exists) of θ, i.e. the language denoted by the axiom $\mathcal{A}(\theta)$.

2.1 Suffix-linear systems

In such systems, a term is either ε, or of the form $a \cdot Y$, where a is a symbol and Y is a language identifier. More precisely, each equation is of the form:

$$X = a_1 \cdot Y_1 + ... + a_n \cdot Y_n \ [+ \ \varepsilon]$$

Such a system θ is equivalent to a finite automaton (Q, I, F, T_Σ), where:

- $Q = \mathcal{V}(\theta)$ is the set of states,
- $I = \mathcal{A}(\theta)$ is the (unique) initial state,
- $F = \{X \in \mathcal{V}(\theta) \mid \varepsilon \in \theta(X)\}$ is the set of final states,
- $T_\Sigma \in Q \times \Sigma \times Q$ is the set of symbol-labelled transitions, with: $(X, a, Y) \in T_\Sigma \Leftrightarrow (a \cdot Y) \in \theta(X)$.

Deterministic automaton (DFA): Moreover, if in each equation $X = a_1 \cdot Y_1 + ... + a_n \cdot Y_n \ [+ \ \varepsilon]$, all the symbols a_i are different, the automaton is deterministic, and each Y_i is said to be the a_i-derivative of X. Such a deterministic system can be built by computing derivatives of regular expressions. Brzozowski has formally defined this construction [4]. The size of the resulting deterministic automaton (and thus the cost of the construction) is, in the worst case, exponential with respect to the size of the regular expression (number of operators in the regular expression).

Non-deterministic automaton (NFA): If in each equation $X = a_1 \cdot Y_1 + ... + a_n \cdot Y_n \ [+ \ \varepsilon]$, some a_i are equal, the resulting automaton is non-deterministic. McNaughton and Yamada defined the construction of such an automaton [10]. Berry and Sethi gave an efficient algorithm to build such an automaton from a regular expression [3]. Antimirov defined a notion of partial derivatives, which generalizes the notion of derivative for non-deterministic automata [2]. For those algorithms, the size of the resulting non-deterministic automaton (and thus the cost of the construction) is, in the worst case, quadratic with respect to the size of the regular expression. More precisely, the number of states is linear, but the number of transitions can be quadratic.

Non-deterministic automaton with ε-transitions: The previous definition can be extended with terms of the form Y. Such systems are equivalent to a non-deterministic automaton with ε-transitions: $(Q, I, F, T_\Sigma, T_\varepsilon)$, where:

- $T_\varepsilon \in Q \times Q = \{(X, Y) \mid Y \in \theta(X)\}$

Thompson has defined the construction of such machines from regular expressions [11]. The main result is that both the size of the automaton and the cost of its construction are linear with respect to the size of the regular expression.

Our goal is to define a linear algorithm that builds a grep machine from a regular expression, so we choose the third kind of linear systems (i.e. non-deterministic automaton with ε-transitions).

2.2 Prefix-linear systems

A completely dual definition can be given by using terms of the form $Y \cdot a$ instead of $a \cdot Y$. Let us see the example of a non-deterministic prefix-linear system with ε-transitions. Such a system θ is equivalent to a finite automaton $(Q, I, F, T_\Sigma, T_\varepsilon)$, where:

- $Q = \mathcal{V}(\theta)$ is the set of states,
- $I = \{X \in \mathcal{V}(\theta) \mid \varepsilon \in \theta(X)\}$ is the set of initial states,
- $F = \mathcal{A}(\theta)$ is the (unique) final state,
- $T_\Sigma \in Q \times \Sigma \times Q$ is the set of symbol-labelled transitions, with: $(Y, a, X) \in T_\Sigma \Leftrightarrow (Y \cdot a) \in \theta(X)$,
- $T_\varepsilon \in Q \times Q = \{(X, Y) \mid Y \in \theta(X)\}$ is the set of ε-labelled transitions.

In this paper, we chose to build prefix-linear systems. Indeed, the problems of building prefix- or suffix-linear systems are completely equivalent, but, as we will show in a following section, prefix-linear systems are "equivalent" to sequential circuits, and then, in some sense, directly "executable".

3 From regular expressions to prefix-linear systems

3.1 Abstract syntax for left-linear systems

We propose here a "functionnal-like" syntax which allows a system of equations to be represented by a single equation:

$$\begin{array}{rcl}
\textit{system} & ::= & X = \textit{terms} \\
\textit{terms} & ::= & \varepsilon \;\mid\; X \;\mid\; X \cdot a \;\mid\; \textit{terms} + \textit{terms} \;\mid \\
 & & \mathsf{let}\, X = \textit{terms}\ \mathsf{in}\ \textit{terms} \;\mid\; \mathsf{rec}\, X = \textit{terms}
\end{array}$$

Note that $\mathsf{rec}\, X = \textit{terms}$ is a macro-notation for $\mathsf{let}\, X = \textit{terms}\ \mathsf{in}\, X$.

The semantic function $\mathcal{L}$ is naturally extended to this new syntax: $\mathcal{L}(\sigma)$ is the language denoted by the *system* σ. We also extend the semantic function for *terms*, which are, in some sense, "partially built" systems: $\mathcal{L}(\tau)$ is (if it exists) the language denoted by the system $X = \tau$, where X is any identifier not appearing in τ.

3.2 The basic algorithm

The idea is to define a function Θ which transforms a regular expression into a prefix-linear system, i.e. which verifies:

$$L(E) = \mathcal{L}(\Theta(E))$$

Moreover, we want the cost of the translation to be linear with respect to the size of the regular expression; the translation should also introduce few variables.

For that purpose, we introduce a recursive function $\Gamma(X, E)$ whose parameters are a regular expression (E) and a language identifier (X) representing the "prefixes" of E. The result of Γ is a partially built system of equations (i.e. a "*terms*" according to the previously defined abstract syntax). This will allow us to introduce variables only when needed.

The meaning of Γ is quite simple: intuitively, $\Gamma(X, E)$ "denotes" the same language as "$X \cdot E$". More formally, let L_X be the language denoted by X:

$$\mathcal{L}(\Gamma(X, E)) = L_X \cdot L(E)$$

The definition of Γ is given by induction on the structure of regular expressions:

$$\Gamma(X, a) = X \cdot a \tag{1}$$
$$\Gamma(X, E^{\varepsilon}) = X + \Gamma(X, E) \tag{2}$$
$$\Gamma(X, E + F) = \Gamma(X, E) + \Gamma(X, F) \tag{3}$$

Let Y be a new identifier:

$$\Gamma(X, E \cdot F) = \mathsf{let}\, Y = \Gamma(X, E)\ \mathsf{in}\ \Gamma(Y, F) \tag{4}$$
$$\Gamma(X, E^*) = \mathsf{rec}\, Y = X + \Gamma(Y, E) \tag{5}$$

The system corresponding to the regular expression E is obtained by computing $\Gamma(Z, E)$ with $Z = \varepsilon$, and then, by naming the result; let X be a new identifier:

$$\Theta(E) = X = (\mathsf{let}\, Z = \varepsilon\, \mathsf{in}\, \Gamma(Z, E))$$

3.3 Example

Let us consider the expression $(a^* + b)^* \cdot a$:

$$\begin{aligned}
\Gamma(Z, (a^* + b)^* \cdot a) &= \mathsf{let}\ \ Y = \Gamma(Z, (a^* + b)^*)\ \ \mathsf{in}\ \ \Gamma(Y, a) \\
&= \mathsf{let}\ \ Y = \Gamma(Z, (a^* + b)^*)\ \ \mathsf{in}\ \ Y \cdot a \\
\Gamma(Z, (a^* + b)^*) &= \mathsf{rec}\ \ T = Z + \Gamma(T, a^* + b) \\
\Gamma(T, a^* + b) &= \Gamma(T, a^*) + \Gamma(T, b) \\
&= (\mathsf{rec}\ \ W = T + \Gamma(W, a)) + T \cdot b \\
&= (\mathsf{rec}\ \ W = T + W \cdot a) + T \cdot b
\end{aligned}$$

A classical system of equations is obtained by extracting the sub-equations and naming the axiom with X:

$$X \;=\; Y \cdot a \quad Y \;=\; T \quad T \;=\; Z + W + T \cdot b \quad W \;=\; T + W \cdot a \quad Z \;=\; \varepsilon$$

3.4 Automaton size and comparison

The size of $\Theta(E)$ (or of the equivalent NFA $(Q, I, F, T_\Sigma, T_\varepsilon)$) can be easily related to the size of E: let o be the number of symbol occurrences, d be the number of dots, e be the number of power-εoperators, and k the number of Kleene stars, we have:

$$|Q| = d + k + 2 \quad |T_\Sigma| = o \quad |T_\varepsilon| = k + e$$

The two additional states are the initial state introduced by the starting rule ("$\mathsf{let}\ Z = \varepsilon\ \mathsf{in}$..."), and a final state corresponding to the axiom.

It is interesting to compare this solution where the states correspond to the dot and star operators, to the McNaughton and Yamada automata (or *normalized NFA's*) where the states correspond to the symbol occurences (in a normalized NFA, we have $|Q| = o + 1$) [10, 3].

The same notion of states (attached to dots and stars) exists in the Antimirov's work: the algorithm can then be viewed as a simple way to compute Antimirov's partial derivatives [2]. However, Antimirov's algorithm does not produce ε-transitions, so the resulting automaton is, in the worst case, quadratic with respect of the size of the regular expression.

The use of ε-transitions makes this algorithm close to the Thompson's one. But the basic Thompson NFA is, in general, bigger: between r and $2r$ states, and between r and $4r$ transitions, where r is the length of the regular expression. Our algorithm can be viewed as a way to directly produce an optimized Thompson NFA.

4 Boolean networks

We present here a simple model for sequential circuits. It consists of a Boolean operator network, together with a simple data-flow semantics. The networks are built with classical Boolean operators (true, false, not, and, or) and a delay binary operator fby, whose first argument is the initial value, and the second the Boolean value to be delayed.

4.1 Syntax

We define a syntax that allows the representation of a network (a cyclic graph in general) by a simple syntactic tree. This is done by using a "functional-like" syntax (X stands for any identifier):

$$\begin{aligned} net ::= &\ X \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{not}\ net \mid net\ \mathsf{and}\ net \mid net\ \mathsf{or}\ net \mid net\ \mathsf{fby}\ net \\ \mid &\ \mathsf{let}\ X = net\ \mathsf{in}\ net \mid \mathsf{rec}\ X = net \end{aligned}$$

Just like for the linear systems of equations (§ 3.1), a network can also be described by a set of equations. For instance, the system of Boolean equations:

$$s = x\ \mathsf{and}\ \mathsf{not}\ y \quad x = a\ \mathsf{or}\ b \quad y = \mathsf{false}\ \mathsf{fby}\ (\ x\ \mathsf{or}\ y\ )$$

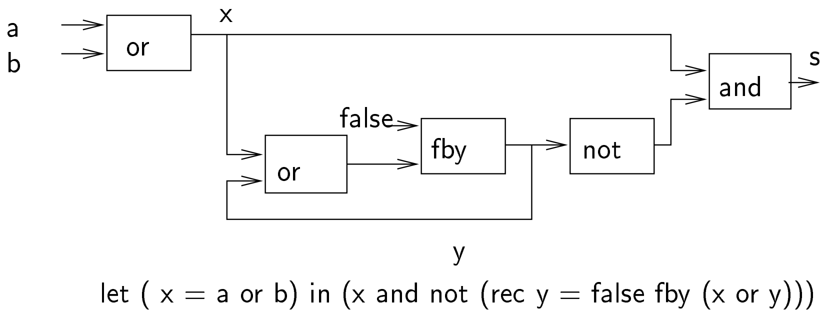

Fig. 1. A Boolean network and the corresponding expression

is equivalent to the network shown (in both syntactic and graphic form) in Fig. 1.

We only consider networks without combinational loop: recursive definitions can only appear under a fby operator. The set of correct networks is denoted by $\mathcal{N}$.

4.2 Synchronous semantics

The set of free variables (inputs) of a network n is denoted by $\mathit{Free}(n)$. A valuation of $\mathit{Free}(n)$ is a function which associates a Boolean value to each variable of $\mathit{Free}(n)$: $\rho : \mathit{Free}(n) \to \mathbb{B}$. A trace over $\mathit{Free}(n)$ is a finite sequence of valuation: $\nu = \rho_1, ..., \rho_k$.

The synchronous semantics models the behavior of idealized sequential circuits: given an input trace ν of length k, a network n produces a sequence of k Boolean values. More precisely, each sub-network (each node in the network) produces a sequence of k Boolean values. For instance, the constant true produces a sequence of k "*true*". Let $\nu = \rho_1, ..., \rho_k$ be the input trace, the input x produces the sequence $\rho_1(x), ..., \rho_k(x)$. The classical operators operate pointwise on sequences; for instance, if n produces the sequence $b_1, ..., b_k$ and n' the sequence $b'_1, ..., b'_k$, then n or n' produces $b_1 \vee b'_1, ..., b_k \vee b'_k$. At last, the fby operator delays its second operand, while its first operand defines the initial value: n fby n' produces $b_1, b'_1, ..., b'_{k-1}$.

The absence of combinational loops in the network n is a sufficient property for this semantics to be operational. In other terms, if n has no combinational loop, then for any input trace ρ, the corresponding output sequence is completely determined. We then consider that any network n without combinational loop is a function from traces to Boolean sequences, and we note $n(\nu)$ the output sequence corresponding to the trace ν.

4.3 Language recognized by a Boolean network

A trace is said to be recognized by the network n if the corresponding output Boolean sequence ends with "*true*". The set of traces recognized by n is called the behavior recognized by n, denoted by $\mathcal{B}(n)$:

$$\mathcal{B}(n) = \{\nu \mid n(\nu) = b_1, ..., \mathit{true}\}$$

In terms of languages, the set of valuation can be viewed as an alphabet Σ, a trace as a word in Σ^*, and a behavior as a language over Σ. Since a network is clearly a finite state machine (with at most 2^m states, where m is the number of fby operators in the network), the behavior recognized by a network is a regular language over Σ.

5 Prefix-linear systems and networks

In this section, we will map a prefix-linear system of equations θ to an Boolean network $n = \Omega(\theta)$.

5.1 Encoding

A prefix-linear system describes a language over some alphabet Σ, while the networks describe behaviors over a set of Boolean variables. So the initial alphabet has to be encoded using Boolean variables. We consider here a "trivial" one-to-one encoding (i.e. we identify the alphabet to a set of Boolean variables). Indeed, the method can easily be adapted for more powerful encoding such as logarithmic encoding.

5.2 Empty trace

As they are defined, the networks cannot recognize the empty trace, so the resulting network cannot be "completely equivalent" to the source system. Let θ be a prefix-linear system, the resulting network $\Omega(\theta)$ simply verifies:

$$\mathcal{B}(\Omega(\theta)) = \mathcal{L}(\theta) \setminus \{\varepsilon\}$$

5.3 Initial states

Intuitively, each term in a system of equations will be replaced by a network recognizing its non-empty traces. In order to treat terms of the form $X \cdot a$, we need to know if the empty string belongs to X, since, if $\varepsilon \in \mathcal{L}(X)$:

$$\mathcal{L}(X \cdot a) \setminus \{\varepsilon\} = (\mathcal{L}(X) \setminus \{\varepsilon\}) \cdot a + a$$

The set of variables X such that $\varepsilon \in \mathcal{L}(X)$ is called the set of initial states (referring to the terminology of NFA's). This set is denoted by $\mathcal{I}(\theta)$ and is recursively defined by:

$$X \in \mathcal{I}(\theta) \Leftrightarrow (\varepsilon \in \theta(X)) \ \vee \ (\exists Y \in \mathcal{I}(\theta) \mid Y \in \theta(X))$$

In fact, this set can be computed during the construction of the system, using the function Λ (§ 1.2) and this computation has a linear cost (with respect to the size of the regular expression).

5.4 From prefix-linear systems to networks

The definition of the mapping function Ω is the following:

$$\Omega(\varepsilon) = \mathsf{false} \qquad \Omega(X) = X \qquad \Omega(\tau + \tau') = \Omega(\tau) \,\mathsf{or}\, \Omega(\tau')$$
$$\Omega(X \cdot a) = (\mathsf{true\,fby}\, X) \,\mathsf{and}\, a \quad \text{if } X \in \mathcal{I}(\theta)$$
$$\Omega(X \cdot a) = (\mathsf{false\,fby}\, X) \,\mathsf{and}\, a \quad \text{if } X \notin \mathcal{I}(\theta)$$
$$\Omega(\mathsf{let}\, X = \tau \,\mathsf{in}\, \tau') = (\mathsf{let}\, X = \Omega(\tau) \,\mathsf{in}\, \Omega(\tau'))$$

Let us come back to the example of § 3.3 ($\theta = \Theta((a^* + b)^* \cdot a)$). We first compute $\mathcal{I}(\theta) = \{Z, T, Y, W\}$, then we obtain the network (given as a set of equations):

$$\begin{aligned} X &= (\mathsf{false\,fby}\, Y) \,\mathsf{and}\, a \\ Y &= T \\ T &= Z \,\mathsf{or}\, W \,\mathsf{or}\, (\mathsf{true\,fby}\, T) \,\mathsf{and}\, b \\ W &= T \,\mathsf{or}\, (\mathsf{true\,fby}\, W) \,\mathsf{and}\, a \\ Z &= \mathsf{false} \end{aligned}$$

5.5 Combinational loops and normalization

We have defined a method for building a sequential circuit form a regular expression: $\Omega(\Theta(E))$.

As it is computed, the intermediate system $\Theta(E)$ (and then the resulting network too) may contain combinational loops. For instance, in the example of § 3.3 ($\theta = \Theta((a^* + b)^* \cdot a)$), we have $W \in \theta(T)$ and $T \in \theta(W)$.

Such combinational loops appear in $\Theta(E)$ if and only if there exists in E a sub-expression of the form (F^*) such that $\Lambda(F)$. In order to avoid combinational loops, we define a function *Norm* on regular expressions, which recursively replaces each sub-expression of the form (F^*) by an equivalent expression (G^*) such that $\neg\Lambda(G)$.

$$\begin{aligned} \mathit{Norm}(a) &= a \\ \mathit{Norm}(E + F) &= \mathit{Norm}(E) + \mathit{Norm}(F) \\ \mathit{Norm}(E \cdot F) &= \mathit{Norm}(E) \cdot \mathit{Norm}(F) \\ \mathit{Norm}(E^{\varepsilon}) &= \mathit{Norm}(E)^{\varepsilon} \\ \mathit{Norm}(E^*) &= \mathit{NormBis}(E)^* \end{aligned}$$

where:

$$\begin{aligned}
NormBis(E) &= Norm(E) \ \text{ if } \neg\Lambda(E) \\
NormBis(E+F) &= NormBis(E) + NormBis(F) \ \text{ if } \Lambda(E+F) \\
NormBis(E\cdot F) &= NormBis(E) + NormBis(F) \ \text{ if } \Lambda(E\cdot F) \\
NormBis(E^{\varepsilon}) &= NormBis(E) \\
NormBis(E^{*}) &= NormBis(E)
\end{aligned}$$

The function *Norm* is a simple parsing which applies the "interesting" function *NormBis* to the Kleene star operators.

Notice that this normalization does not replace the expression E^* by an expression F^* such that $L(F) = L(E) \setminus \varepsilon$. Such a transformation would have a quadratic cost since, for all regular languages ℓ and ℓ' containing the empty string: $(\ell \cdot \ell') \setminus \varepsilon = (\ell \setminus \varepsilon) \cdot \ell' + \ell \cdot (\ell' \setminus \varepsilon)$.

The normalization simply replaces the expression E^* by an expression F^* such that $L(F^*) = L(E^*)$ and $\neg\Lambda(F)$. The correctness of the algorithm is based on the following lemma: $(\varepsilon \in \ell \wedge \varepsilon \in \ell') \ \Rightarrow \ (\ell \cdot \ell')^* = (\ell + \ell')^*$.

Here are some examples of normalization:

$$Norm((a^* + b)^*) \ = \ (a+b)^* \quad Norm(((a^*)^*)^*) \ = \ a^*$$
$$Norm((a^{\varepsilon} \cdot b^* + c^*)^*) \ = \ (a+b+c)^*$$

6 From regular expressions to Boolean networks

Finally, let E be a regular expression over an alphabet Σ interpreted as a set of Boolean variables, we have defined a method which computes a correct "equivalent" network n, i.e. a network without combinational loop recognizing all the non-empty traces of the language $L(E)$:

$$\mathcal{B}(\Omega(\Theta(Norm(E)))) = L(E) \setminus \varepsilon$$

This method involves the computation of the function Λ on each sub-expression of E, the normalization of E, and also the computation of the initial states in the system $\Theta(Norm(E))$. But the main result is that the cost of all those treatments is linear with respect to the size of the source regular expression.

In order to outline this linear complexity, we define in this section a function Φ which directly produces a network from a regular expression. Like the function Θ, the function Φ is defined using a recursive function, Υ, which takes a special parameter representing the prefixes. This function takes two parameters to describe the prefixes:

$$\begin{aligned}
\Upsilon : \mathcal{N} \times \mathbb{B} \times \mathcal{R}exp &\rightarrow \mathcal{N} \\
(X, b, E) &\mapsto n
\end{aligned}$$

The first parameter (X) is a network which is supposed to recognize the non-empty traces of the prefixes, while the second one (b) is a Boolean value indicating whether the empty trace belongs to the prefixes. Intuitively, this new parameter b allows us to directly compute the set of "initial states" (§ 5.3). The normalization of the regular expression (§ 5.5) is also performed during the construction. For that purpose, a special recursive function Υ^* is applied to the Kleene star operands.

$$\Phi(E) = \mathsf{let}\, Z = \mathsf{false}\, \mathsf{in}\, \Upsilon(Z, \mathit{true}, E)$$

$$\begin{aligned}
\Upsilon(X, b, a) &= (b\, \mathsf{fby}\, X)\, \mathsf{and}\, a \\
\Upsilon(X, b, E + F) &= \Upsilon(X, b, E)\, \mathsf{or}\, \Upsilon(X, b, F) \\
\Upsilon(X, b, E^{\mathcal{E}}) &= X\, \mathsf{or}\, \Upsilon(X, b, E)
\end{aligned}$$

Let Y be a new identifier:

$$\begin{aligned}
\Upsilon(X, b, E.F) &= \mathsf{let}\, Y = \Upsilon(X, b, E)\, \mathsf{in}\, \Upsilon(y, b \wedge \Lambda(E), F) \\
\Upsilon(X, b, E^*) &= \mathsf{rec}\, Y = X\, \mathsf{or}\, \Upsilon^*(Y, b, E)
\end{aligned}$$

$$\begin{aligned}
\Upsilon^*(X, b, E) &= \Upsilon(X, b, E) \quad \text{if } \neg\Lambda(E) \\
\Upsilon^*(X, b, E + F) &= \Upsilon^*(X, b, E)\, \mathsf{or}\, \Upsilon^*(X, b, F) \quad \text{if } \Lambda(E + F) \\
\Upsilon^*(X, b, E \cdot F) &= \Upsilon^*(X, b, E)\, \mathsf{or}\, \Upsilon^*(X, b, F) \quad \text{if } \Lambda(E \cdot F) \\
\Upsilon^*(X, b, E^{\mathcal{E}}) &= \Upsilon^*(X, b, E) \\
\Upsilon^*(X, b, E^*) &= \Upsilon^*(X, b, E)
\end{aligned}$$

Figure 2 shows the result of $\Upsilon(\mathsf{false}, \mathit{true}, (a^* + b)^* \cdot a)$. One can see that the resulting network has the same structure as the regular expression.

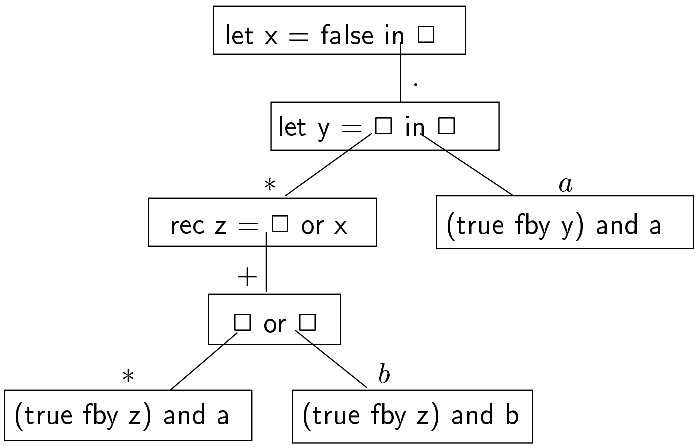

Fig. 2. The network of $(a^* + b)^* \cdot a$

7 Implementation and use

The origin of this work takes place in the domain of synchronous programming [8]. Synchronous programming offers an idealized framework for programming reactive systems. For instance, idealized sequential circuits are synchronous programs. This idealized framework also permits to perform formal verifications on programs. Critical properties are described by means of Boolean synchronous programs called *observers* [9]. Such observers can be expressed with any existing synchronous language (Lustre, Argos, Esterel), but it appears that the classical regular constructs (sequence, iteration) are also very useful to describe safety properties.

The idea was to translate, with a minimal cost, a safety property expressed with regular constructs into a suitable synchronous program, such as a Boolean network.

A tool called reglo has been designed for this purpose. This tool translates a set of regular expressions, describing traces over a set of Boolean variables (i.e. the input alphabet is supposed to be already Boolean encoded), into an equivalent Boolean dataflow network, expressed in the language Lustre.

References

1. A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, chapter 5, pages 257–300. Elsevier Science Publishers B. V., 1990.
2. V. Antimirov. Partial derivatives of regular expressions and finite automata constructions. In E. W. Mayr and C. Puech, editors, *Proceedings of STACS '95*, pages 455–466, Munich, Germany, March 1995. LNCS 900, Springer Verlag.
3. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.
4. J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.
5. J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.
6. C. H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. In Apostolico, Crochemore, Galil, and Manber, editors, *Combinatorial Pattern Matching. Proceedings*, pages 88–108. LNCS 644, Springer Verlag, 1992.
7. R. W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *Journal of the ACM*, 29(3):603–622, 1982.
8. N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.
9. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.
10. R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Trans. on Electronic Computers*, 9(1):39–47, 1960.
11. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–423, June 1968.