A Recognizer of Rational Trace Languages

Federico Maggi

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano
fmaggi@elet.polimi.it

Abstract—The relevance of instruction parallelization and optimal event scheduling is currently increasing. In particular, because of the high amount of computational power available today, the industrial interest on automatic code parallelization is raising notably. In the last years, several contributions have arisen in these fields, exploiting the theory of traces that provides a powerful mathematical formalism that can be effectively used to model and study concurrent executions of events. However, there is a quite large amount of open problems that need to be further investigated in this area.

In this paper, we present a one-pass recognition algorithm to solve the membership problem for rational trace languages, that is the problem of deciding whether or not a certain string belongs (i.e., is member of) a trace, or a trace language. Solving this problem is fundamental for designing efficient parsers. Our solution is detailed through the formal specification of the Buffer Machine, a non-deterministic, finite-state automaton with multiple buffers that can solve the membership problem in polynomial time.

I. INTRODUCTION

Traces [1], [2], [3] can be exploited to model concurrency. Trace languages can be considered as an extension of string languages. For instance, while string languages can be used to define the syntax of a certain computer program, trace languages can also capture the dependencies among the program instructions.

In this work, we focus on the Membership Problem (MP) in the case of trace languages. This problem plays a key role in real-world applications where classical and expensive dynamic programming techniques are used. Our contribution consists in a one-pass version of a two-pass recognition algorithm [4] we analyzed in details. The original algorithm requires a pre-processing step before parsing. Although it consists in a linear scan, a two phases recognition may be unpractical for some real cases. We avoid this issue by incorporating the first phase in the parsing algorithm itself. Our algorithm is defined by means of an abstract machine we named Buffer Machine (BM), a non-deterministic recognizer that solves the MP. In addition, we evaluate the performances and characteristics of the proposed solution using a testbed implementation we released, which also includes a pseudo-random generator of strings, automata and dependency relations, which can be used to experiment with the prototype, as detailed in [2].

II. PRELIMINARIES

We use the conventional concept of language (or string language) $L$: a (sub)set of strings generated by a free monoid $\Sigma^* \supseteq L$, where $\Sigma$ is the alphabet. Strings are denoted by smallcase letters: $u = a_1a_2\cdots a_n$, $v = b_1b_2\cdots b_m$, where $n = |u|$, $m = |v|$ are the length $|.|$ of $u$ and $v$, respectively.

A trace is indicated as an equivalence class $[t] = \{t_1, t_2, \ldots, t_k\}$ represented by $t$. $[t]$ contains strings drawn from the trace monoid $F(\Sigma, I)$, also called partially commutative free monoid. More formally, $F(\Sigma, I) = \Sigma^*/\equiv_I$, where $I \subseteq \Sigma \times \Sigma$ is the independence relation. $I$ is a symmetric and reflexive equivalence relation and $\equiv_I$ is its minimum congruence over $\Sigma^*$. The dependence relation $\theta$ is also used as the complement of $I$: $t = \theta = \Sigma \times \Sigma \setminus I$. Since $(a, a) \in I$, $\forall a \in \Sigma$, the reflexive arcs are omitted if not strictly necessary.

A trace language $T$ is a (subset) of traces generated by $F(\Sigma, I) \supseteq T$, defined over the commutative alphabet $(\Sigma, I)$. More formally, $T = [L]_{\equiv_I} = \{t \in F(\Sigma, I) \mid \exists u \in L : t = [u]\}$. The family of rational trace languages $Rat(\Sigma, I)$ is the focus of this work. $Rat(\Sigma, I)$ is proven to be generated by regular string languages [5]; i.e., $[L] = T \in Rat(\Sigma, I)$ iif $L \in Reg(\Sigma)$. It is the smallest class of trace languages containing all finite sets and closed w.r.t. union, product and star.

Prefixes $Pref_i(t)$ of length $l$ of a trace $t$ are the set of words $t_i$ s.t. $t = t_i \cdot v$ for some trace $v$. The product operator ‘.’ of $F(\Sigma, I)$ is s.t. $\forall t_1, t_2 \in F(\Sigma, I)$ and $t_1 \cdot t_2 = t_1t_2 = [uv]$, where $t_1 = [u]$ and $t_2 = [v]$. On languages, if $T_1 = [L_1], T_2 = [L_2]$, then $T_1 \cdot T_2 = \{t \in F(\Sigma, I) \mid t = t_1 \cdot t_2 \in T_1, t_2 \in T_2\}$. The Kleene star on traces is $t^* = \cup_{n=0}^{\infty} t^n$ where $t^0 = \epsilon = [\epsilon]$ is the empty trace, and $t^n = t \cdot t^{n-1}$. On languages: $T^* = \cup_{n=0}^{\infty} T^n$, $T^0 = \{\epsilon\}$, and $T^n = T \cdot T^{n-1}$.

Both $I$ and $\theta$ can be represented with undirected graphs. Formally, in case of $I$, $G = \langle V, E \rangle = (\Sigma, I)$. The notion of clique covering and maximal clique of a graph (i.e., of a relation) will be used. A clique $V_i \subseteq G$ is any complete subgraph $G_i = \langle V_i, E_i \rangle$, i.e., $(a, b) \in E_i \forall a, b \in V_i$ with $a \neq b$. A clique $V_i$ is also maximal (w.r.t. the inclusion relation) if $G_i$ is the maximal complete super-graph of $G$; or, in other words, if there is no super-set that is a clique itself: $\forall V_j \supset V_i \Rightarrow i = j$. The maximal clique covering of $G$ with respect to $E$ is the set $\mathcal{M}_E(V) = \{V_1, \ldots, V_i, \ldots, V_k\}$ containing all
and only the maximal cliques.

Being $F(\Sigma, \theta)$ represented by $G = (V, \Sigma; E = \theta)$, then $M_\theta(\Sigma)$ is such that $\forall i,j \in [1,|M_\theta(\Sigma)|] \mid i \neq j \land V_i \subseteq V_j$. The (maximal) cliques are subsets of $\Sigma$, $\Sigma_1, \Sigma_2, \ldots, \Sigma_k \subseteq \Sigma$, with $k \geq 1$. Similarly, $M(\Sigma, z)$ can be defined for $z$.

A projection of a trace on $\Sigma'$ is a morphism $\pi_{\Sigma'}(\cdot)$ defined as $\pi_{\Sigma'} : \Sigma^* \rightarrow \Sigma'^*$: given $u = w a$, $\pi_{\Sigma'}(u) = \pi_{\Sigma'}(w)a$ if $a \in \Sigma'$; otherwise $\pi_{\Sigma'}(u) = \pi_{\Sigma'}(u)$, or $\pi_{\Sigma'}(\varepsilon) = \varepsilon$. Thus, $\pi_{\Sigma'}(u) = u' \in \Sigma'^*$. Since $\Sigma' \in M(\Sigma)$, then $\Sigma'$ are cliques. $\pi_{\Sigma'}(t)$ maintains the order of the symbols of $t$; that is, $\forall a_i, a_j \in \pi_{\Sigma'}(t) \land i < j \Rightarrow \exists i' < j' \mid a_{i'} = a_i \land a_{j'} = a_j \
\forall d,e \in \Sigma \setminus \{a\}$, is associated to a subset $\Sigma_i \subseteq \Sigma$ of symbols and thus uniquely identified by their own alphabet, i.e. $\forall b_i, b_j : \Sigma_i = \Sigma_j \Rightarrow b_i = b_j$. Furthermore, we define the concept of a buffer alphabet, a buffer $b_i$ is a FIFO queue containing only symbols in $\Sigma_i ; b_i := \langle a_i \cdots a_j \cdots a_o \rangle$ where “$a_i \cdots$” is the input side while “$\cdots a_o$” is the output side. A buffer with no symbols is empty and is indicated as $b_i = \langle \varepsilon \rangle = \emptyset$.

Each buffer, short $b_{\Sigma_i}$, is associated to a subset $\Sigma_i \in \Sigma$ of symbols and thus uniquely identified by their own alphabet, i.e. $\forall b_i, b_j : \Sigma_i = \Sigma_j \Rightarrow b_i = b_j$. For instance, if $\Sigma_i = \{a, c \} \subseteq \Sigma = \{a, b, c, d, e\}$, then $b_i = b_{\Sigma_i = \{a,c\}}$ holds symbols drawn from $\Sigma_i$ only. Buffers are associated to the following functions.

**Definition 2 (Buffer functions):** Let $b_{\Sigma_i}$ be a buffer. The functions $\text{Empty}(b_{\Sigma_i})$, $\text{Dequeue}(b_{\Sigma_i})$, and $\text{Enqueue}(b_{\Sigma_i}, a)$ are said to be buffer functions.

- $\text{Empty}(b_{\Sigma_i}) = T$ iff $b_{\Sigma_i} = \langle \varepsilon \rangle$ and $T$ otherwise.
- $\text{Dequeue}(b_{\Sigma_i})$ is defined only if $\text{Empty}(b_{\Sigma_i}) = T$. It removes the rightmost symbol $a_o$ from $b_{\Sigma_i} = \langle a_1 \cdots a_o \rangle$, and returns it as $\{a_o\}$.
- $\text{Enqueue}(b_{\Sigma_i}, a)$ inserts the symbol $a \in \Sigma_i$ into $b_{\Sigma_i}$.

**Example 1:** This example illustrates how each buffer function works.

- It is straightforward that if $b_{\Sigma_i = \{a,c\}} = \langle \varepsilon \rangle$ and $b_{\Sigma_i = \{a,c\}} = \langle \varepsilon \rangle$, then $\text{Empty}(b_{\Sigma_i}) = T$ while $\text{Empty}(b_{\Sigma_i}) = F$.
- If $b_{\Sigma_i = \{a,c\}} = \langle a a a a c \rangle$, $\text{Dequeue}(b_{\Sigma_i})$ returns $\{a\}$ and, as a result, $b_{\Sigma_i} = \langle a a a c \rangle$.
- Given $b_{\Sigma_i} = \langle a a c \rangle$, $\text{Enqueue}(b_{\Sigma_i}, a)$ would result in $b_{\Sigma_i} = \langle a a a c \rangle$. This function is such that if $b_{\Sigma_i} = \langle a a c \rangle$, invoking $\text{Enqueue}(b_{\Sigma_i}, c)$ results in $b_{\Sigma_i} = \langle a a a c \rangle = \langle a a c \rangle$.

Moreover, we define the concept of family of buffers as either one of the following definitions.

**Definition 3 (Family of buffers specific to a symbol):** Let $b_{\Sigma_1}, \ldots, b_{\Sigma_k}$ be a set of buffers. A family of buffers $B_a$
specific to the symbol \( a \in \Sigma \) is defined as the set of all buffers whose alphabet contains the symbol \( a \in \Sigma \). That is: \( \mathcal{B}_a = \{ b_{k} \mid a \in \Sigma_i \} \).

For instance, consider \( \mathcal{M}_a(\Sigma) = \{ \Sigma_1 = \{ a \}, \Sigma_2 = \{ b, c \}, \Sigma_3 = \{ c, d \} \} \) and its associated buffers \( \{ b_{\Sigma_1}, b_{\Sigma_2}, b_{\Sigma_3} \} \). The family of buffers specific to \( c \) is \( \mathcal{B}_c = \{ b_{\Sigma_1}, b_{\Sigma_3} \} \), i.e., a set of those buffers belonging to the subset of the clique covering induced by the symbol \( c \). A more general definition is the following.

**Definition 4 (Family of buffers specific to a set of alphabets):** Let \( b_{\Sigma_1}, \ldots, b_{\Sigma_K} \) be a set of buffers and \( \Gamma = \{ \Sigma_i \}_{i=1}^N \) a set of \( N \) alphabets. A family of buffers \( \mathcal{B}_\Gamma \) specific to a set of alphabets \( \Gamma \) is defined as the set of all the buffers \( b_{\Sigma_i} \) whose alphabet \( \Sigma_i \) is in \( \Gamma \): \( \mathcal{B}_\Gamma = \{ b_{\Sigma_i} \mid \Sigma_i \in \Gamma \} \).

The buffer functions are extended to families of buffers as follows.

**Definition 5 (Extended buffer functions):** Let \( \mathcal{B}' \) be a family of buffers. The extended buffer functions are:

- **Empty(\( \mathcal{B}' \))** \( \iff \bigwedge_{b_{\Sigma_i} \in \mathcal{B}'} \text{Empty}(b_{\Sigma_i}) \).
- **Dequeue(\( \mathcal{B}', a \))** returns the set of symbols \( \Sigma' = \{ a_i \mid \{ a_i \} = \text{Dequeue}(b_{\Sigma_1}) \land b_{\Sigma_i} \in \mathcal{B}' \} \).
- **Enqueue(\( \mathcal{B}' \), \( \{ a \} \))** is such that Enqueue(\( b_{\Sigma_2}, \{ a \} \)) is executed \( \forall b_{\Sigma_2} \in \mathcal{B}' \).

Given the above definitions, it is straightforward to define the input tape.

**Definition 6 (Input Tape):** Let \( \Sigma \) be an alphabet. The input tape is \( T_I := b_{\Sigma_2} \).

The input tape is denoted as \( T_I = a_1 a_2 \ldots a_n = [a_n \ldots a_2 a_1] \) and can hold any symbol drawn from \( \Sigma \). The Enqueue(\( T_I, a \)) function is undefined for \( T_I \) while the other operations are as in Definition 2. Without loss of generality, we assume that the input string is already on \( T_I \) and its symbols are consumed in the same order of placement. No further enqueues are allowed.

The BM \( M \) that solves the MP for a given trace language \( \text{Rat}(\Sigma, \theta) \supseteq T = [L] \subseteq \mathbb{F}(\Sigma, \theta) \) is formally defined as follows. Note that \( A : L = L(A) \) is known.

**Definition 7 (Buffer Machine):** A Buffer Machine (BM) is a 4-tuple \( M := (A, \tau, \mathcal{B}, T_I) \), where \( T_I \) is the input tape, and:

- \( \mathcal{B} = \mathcal{B}_{M_0}(\Sigma) = \{ b_1, b_2, \ldots, b_{i_1}, \ldots, b_K \} = \{ b_{\Sigma_1}, b_{\Sigma_2}, \ldots, b_{\Sigma_2}, \ldots, b_{\Sigma_K} \} \in \mathcal{B} \) is the family of buffers associated to \( M_0(\Sigma) \).
- \( A = \langle Q, \Sigma, \delta, q_0, Q_F \rangle \) is the deterministic recognizer of \( L = L(A) \), a finite state automaton: \( Q \) and \( Q_F \subseteq Q \) are the finite set of states and acceptance states, respectively; \( \delta \) is the transition function.

- \( \tau : Q \times \Sigma \times \mathbb{F} \rightarrow \nu(Q \times \mathbb{F}) \) is the transition function of the control device. The current symbol on \( T_I \) is denoted by \( a \in \Sigma \). Three transition modes are defined:
  - **Read** reads if \( a \neq \epsilon \) (i.e., \( \text{Empty}(T_I) \neq \perp \)) and is s.t. \( \tau(q, a, \mathcal{B}) = (q', \mathcal{B}'') \). It is always followed by a **BufferWrite**.
  - **BufferWrite** reads s.t. \( \mathcal{B}' : \text{Enqueue}(\mathcal{B}_a, a) \), \( q, \mathcal{B} \rightarrow (q', \mathcal{B}'') \).
  - **BufferRead** reads if \( \text{Dequeue}(\mathcal{B}_a) = \{ a' \} \) and is s.t. \( \tau(q, a, \mathcal{B}''') = \{ q' = \delta(q, a'), \text{Dequeue}(\mathcal{B}_a) \} \).

The choice among the buffer(s) \( b_{\Sigma_i} \), and among the three modes is non-deterministic.

**Definition 8 (BM configuration):** A configuration of a BM, \( M \), is a tuple \( M^C = \langle q, \mathcal{B} \rangle \in \nu(Q \times \mathbb{F}) \) where \( q \) is the current state of \( A \). \( M^I = \langle q_0, \{ \varnothing, \ldots, \varnothing \} \rangle \) is the initial configuration while \( M^F = \langle q, \epsilon, \mathcal{B} \rangle \) is the acceptance configuration, where \( q_F \in Q_F \).

The **Read** transition reads one symbol from \( T_I \) if it cannot be performed because \( a = \epsilon \) then a **BufferRead** is performed. Each **Read** is followed by a **BufferWrite** which enqueues the read symbol \( a \) on all the buffers in \( \mathcal{B}_a \) (i.e., the current symbol is pushed onto all the buffers associated to the clique the symbol \( a \) belongs to). The **BufferRead** consumes an \( a \) from all the buffers having \( a \) on the output side.

**Example 2 (BM transition modes):** Given \( M_1 = \langle A_1, \tau, \mathcal{B}, T_I \rangle \) where \( A_1 \) is as follows.

- **Read** transition reads one symbol from \( T_I \); if it cannot be performed because \( a \neq \epsilon \) then a **BufferRead** is performed. Each **Read** is followed by a **BufferWrite** which enqueues the read symbol \( a \) on all the buffers in \( \mathcal{B}_a \) (i.e., the current symbol is pushed onto all the buffers associated to the clique the symbol \( a \) belongs to). The **BufferRead** consumes an \( a \) from all the buffers having \( a \) on the output side.

- **BufferWrite** reads s.t. \( \mathcal{B}' : \text{Enqueue}(\mathcal{B}_a, a) \), \( q, \mathcal{B} \rightarrow (q', \mathcal{B}'') \).

- **BufferRead** reads if \( \text{Dequeue}(\mathcal{B}_a) = \{ a' \} \) and is s.t. \( \tau(q, a, \mathcal{B}''') = \{ q' = \delta(q, a'), \text{Dequeue}(\mathcal{B}_a) \} \).

The choice among the buffer(s) \( b_{\Sigma_i} \), and among the three modes is non-deterministic.

**Figure 2:** A visual representation of a BM. A sample trace language having \( K \) cliques in its dependence relation, \( \theta \), is used.
changes: \( \mathcal{B} = \{ \langle ab\rangle, \langle d\text{bbed} \rangle \} \), and the current state of the machine becomes \( q' = q_0 \).

Definition 9 (Extension of \( \tau \)): The extension of \( \tau \) to strings is \( \hat{\tau} : \mathcal{Q} \times \Sigma^* \times \mathbb{B} \rightarrow \nu(\mathcal{Q} \times \mathbb{B}) \). \( \hat{\tau}(q, \varepsilon, \mathcal{B}) := \langle q, \mathcal{B} \rangle \), and \( \hat{\tau}(q, u, \mathcal{B}) := \bigcup_{r \in \hat{\tau}(q, u, \mathcal{B})} \tau(r, a, \mathcal{B}''') \), where \( \mathcal{B}', \mathcal{B}'' \) are determined according to the transition modes.

Note that \( |\mathcal{B}| \) is determined by the (size of the clique covering of the) dependence relation, \( K = |\mathcal{M}_0(\Sigma)| \), since one buffer is instantiated for each clique of the dependence relation. Also note the following.

Proposition 1: \( \mathcal{B} \) is a family of buffers specific to the set \( \Gamma \) of alphabets, s.t. \( \bigcup_{\Sigma_i \in \mathcal{F}} \Sigma_i = \Sigma \).

Proof: \( \mathcal{M}_0(\Sigma) \) is a covering of all the alphabet: \( \bigcup_{\Sigma_i \in \mathcal{M}_0(\Sigma)} \Sigma_i = \Sigma \). Note that, \( \Gamma = \mathcal{M}_0(\Sigma) \). By definition, one buffer \( b_{\Sigma_i} \) exists for each \( \Sigma_i \in \mathcal{M}_0(\Sigma) = \Gamma \). Thus, each \( b_{\Sigma_i} \in \mathcal{B} \) is s.t. \( a \in \Sigma_i \rightarrow a \in \Sigma \). If \( \exists a' \notin \Sigma \) then \( a' \) does not belong to any of the cliques \( \Sigma_i \in \mathcal{M}_0(\Sigma) \) otherwise it would belong to \( \Sigma \) as well, raising a contradiction. However, it could be that \( \Sigma_i \cap \Sigma_j \neq \emptyset \) for some \( i, j \).

Proposition 2 (Correctness): Let \( M \) be a BM. \[ [t] \in T = [L] \in \mathcal{R}(\mathcal{Q}, \theta) \] a trace, and \( u \in L \). If \( u \in [t] \) then \( \hat{\tau}(q_0, u, \{ \varepsilon, \ldots, \varnothing \}) \cap M_F = \emptyset \).

Proof: Let us assume that \( u \in [t] \) but \( \hat{\tau}(q_0, u, \{ \varepsilon, \ldots, \varnothing \}) \cap M_F = \emptyset \). Then, at the end of all the runs of the non-deterministic machine, one of the following conditions must hold:

1) \( M \) ends up at configuration \( M_{C'} = \langle q, \emptyset, \ldots, \varnothing \rangle \) where \( q \notin Q_F \);
2) \( M \) ends up at configuration \( M_{C''} = \langle q_F, \mathcal{B} \rangle \) s.t. \( \exists b_{\Sigma'_{j}} \in \mathcal{B} \) that has not been dequeued.
3) the reading head of \( T_I \) is not pointing to the last symbol of \( u \).

Let us analyze the implications of each case.

1) \( \Rightarrow u \) is not accepted by \( A \), i.e. \( u \notin L \), but this means that \( L \neq L(A) \).
2) \( \Rightarrow u \) is accepted by \( A \) since \( q_F \in Q_F \). Also, \( \exists a' \in b_{\Sigma'} \) that has not been dequeued from \( b_{\Sigma'} \). If \( \exists 1 a' \), then the \( a' \) also prevents other symbols to be dequeued. If \( \exists 1 a' \), then \( a' \) is the only symbol on the buffer. In both of the cases, either:

a) \( a' \) has been enqueued in the wrong buffer, but this contradicts the definition of the BufferWrite mode of \( \tau \) (Definition 7).

b) \( a' \) cannot be dequeued, but this must be that, either:

i) \( \exists q' \in \delta(q', a') \), but this implies that \( q \notin Q_F \).

ii) \( a' \notin \Sigma' \), but this implies that \( \Sigma' \) is not the alphabet of \( b_{\Sigma'} \), otherwise \( a' \) would not be in \( b_{\Sigma'} \).

Thus, i. and ii. contradict the definition of the BufferRead mode of \( \tau \) (Definition 7).

3) (straightforward) \( \Rightarrow \) the machine is not at the end of the computation.

Note 1: An alternative way to proof Proposition 2 consists in showing that it implements Algorithm 1, which can be shown to be equivalent to the algorithm presented in [4, Section 3.2].

IV. IMPLEMENTATION DETAILS

This section describes the technical aspects of Quick Earley-Like Membership Evaluator (QELME), the tool we released to experiment with one of the possible deterministic algorithms that implement the non-deterministic BM.

A. Implementing the Buffer Machine Through a Deterministic Algorithm

Before going into the details of the QELME architecture, a description of the deterministic algorithm used is given. Algorithm 1 requires the clique covering \( C = \mathcal{M}_0(\Sigma) \) of the dependence relation, the automaton \( A \) that defines the (string) language and the string \( u \) to be tested. The procedure needs some variables to perform intermediate computations, namely:

- \( E \) is a set that contains structured elements \( e \) in the same form described in Section V and used in [4]. We recall that \( e.cursors \) is a tuple of \( m \) elements, thus \( e.cursors_j \in \{ 0, \ldots, |\pi_{\Sigma_j}(u)| \} \) indicates the \( j \)-th element (i.e., cursor), where \( |\pi_{\Sigma_j}(u)| \) is the length of the projection of the string \( u \) on the \( j \)-th clique.

- \( \Pi \) is the set of projection on cliques, used to emulate the buffers (see Definition 1 and 7).

- \( M \) is a matrix of size \( |C| \times |Q| \) initialized to \( \perp \). As in [4, Section 3.2], it is used to keep track of the elements actually existing in \( E \), in order to perform the union in constant time. As a shorthand, we will use the function \( M : E \rightarrow \{ \top, \bot \} \) defined as \( M(e) = \begin{cases} \top & e \in e.cursors \ \\ \bot & \text{otherwise} \end{cases} \).

- \( R \) is a vector of cursors, one per clique. Each element of the vector is \( R_j \in \{ 0, \ldots, |\pi_{\Sigma_j}(u)| \} \) and points to the last symbol read from the \( j \)-th projection. Along with each element in \( E \), the vector \( R \) is used to represent a buffer. More precisely, each buffer \( b_j \in \mathbb{B} \) of the BM is emulated by (1) one projection \( \pi_{\Sigma_j}(u) \), (2) one pointer \( R_j \) to the head of the buffer (i.e., beginning of the projection) and (3) one pointer \( e.cursors_j \) to its tail (i.e., end of the projection).

The buffer functions (see Definition 2 and 5) are emulated as shown in Table I

<table>
<thead>
<tr>
<th>BUFFER FUNCTION</th>
<th>IMPLEMENTED EMULATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enqueue( (b_{\Sigma_j}, a) )</td>
<td>Enqueue( (\pi_{\Sigma_j}(u), a) ) &amp; ( R_j \leftarrow R_j + 1 )</td>
</tr>
<tr>
<td>Dequeue( (b_{\Sigma_j}) )</td>
<td>( e.cursors_j \leftarrow e.cursors_j + 1, \forall e \in E )</td>
</tr>
<tr>
<td>Empty( (b_{\Sigma_j}) )</td>
<td>( \forall e \in E \Rightarrow e.cursors_j = R_j )</td>
</tr>
</tbody>
</table>

Table I: Emulation of buffer functions.
\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input}
\State $A \leftarrow \langle Q, \Sigma, \delta, q_0, Q_F \rangle$ /* The automaton */
\State $C \leftarrow \Sigma_1, \ldots, \Sigma_m$ /* Clique covering */
\State $u \leftarrow u_1 \cdots u_n$ /* The string */
\State \textbf{Variables}
\State $\text{decision} \leftarrow \bot$
\State $E \leftarrow \{\}$ /* The working list */
\State $\Pi(u) \leftarrow \{\pi_{\Sigma_1}(u), \ldots, \pi_{\Sigma_m}(u)\} = \{\varepsilon, \ldots, \varepsilon\}$ /* The projections */
\State $M : C \times Q \mapsto \{\top, \bot\}$ /* Initialized to $\bot$ */
\State $R \leftarrow \langle 0, \ldots, 0 \rangle \cdots m$ /* Right cursors */
\While{$i \leq 2n \land \neg \text{decision}$}
\State $E' \leftarrow \{\}$ /* Move the virtual cursor one step ahead */
\If{$i \leq n$}
\State \textbf{forall} $\Sigma' \in I(u_i)$ \textbf{do}
\Comment{Perform a BufferWrite}
\State \textbf{Enqueue($\pi_{\Sigma'}(u), u_i$)}
\EndForAll
\EndIf
\ForAll{$e \in E$}
\If{$\exists j \mid e.\text{cursors}_j = R_j$ then $E' \leftarrow E' \cup e$}
\Comment{For all the cursors}
\For{$j \leftarrow 0; j < m; j \leftarrow j + 1$}
\If{$e.\text{cursors}_j < R_j$ then}
\State $u_s \leftarrow (\pi_{\Sigma_j}(u))_{e.\text{cursors}_j}$
\If{$\exists (e.\text{state}, u_s)$ then}
\State \textbf{Update($u_s, e, E', C, R, M, A$)}
\Comment{Check readability}
\If{$\forall \Sigma_k \in I(u_s) \Rightarrow (\pi_{\Sigma_k}(u))_{e.\text{cursors}_k} = u_s$ then}
\Comment{Perform a BufferRead}
\EndIf
\EndIf
\EndIf
\EndFor
\EndForAll
\State $i \leftarrow i + 1$
\Comment{Update the next working list}
\State $E \leftarrow E'$
\EndWhile
\end{algorithmic}
\caption{Deterministic algorithm implementing the BM.}
\end{algorithm}

Furthermore, Algorithm 1 relies on the shorthand procedure $\text{Update}(\cdot)$ (Algorithm 2), which is performed only if the current character $u_s$ is on the tail of all the buffers whose alphabet contains $u_s$. The tail of the buffer, according to the current element $e$, is the symbol at position $e.\text{cursors}_k$ of the $k$-th clique, that is the symbol $(\pi(u))_{e.\text{cursors}_k}$.

The main cycle is executed $2n$ times in the worst case ($O(n)$). If the string is accepted before $2n$ iterations (but after $i$ iterations, at least), then the cycle exits with a positive decision. Note that the condition $i < 2n$ is required since, in the worst case, the string $u$ is fully permuted w.r.t. all the commutations allowed by $\theta$ and thus all the symbols must be buffered before being read. This is equivalent to a double scan of the whole string. Due to this, the buffering (i.e., $\text{BufferWrite}$) phase is executed only if there are symbols to be read, $i < n$. For this reason, we will call $i$ a “virtual cursor”: it emulates the real cursor on the input tape $T_I$ if $0 \leq i \leq n$, while if $n < i \leq 2n$ it is used to count the iterations of the outmost cycle. The \textbf{forall} cycle iterates over the existing element into the current working list $E$. For all the left cursors hold by the current element $e$, if there are symbols to read in the buffer (i.e., if $e.\text{cursors}_j < R_j$), then the symbol on the top of the buffer (i.e., projection) is read and $\text{Update}$ is invoked.

\textbf{B. Prototype implementation}

The abstract BM has been implemented into a highly configurable, parametric, and scalable testbed application, written in the Python language. The open-source code is available for download at http://qelme.googlecode.com. The application can be decomposed into the following components.

\textit{Sigma:} This module implements $\Sigma$. The internal representation is a Python \texttt{List}. Methods to generate random strings
of given length and random coverings of the alphabet are provided. The generation of the coverings can be controlled through the following parameter.

Definition 10 (Clique covering density): Let $M(\Sigma)$ be a clique covering of the alphabet $\Sigma$. The density of $M(\Sigma) = \{\Sigma_1, \ldots, \Sigma_m\}$ is

$$d(M(\Sigma)) = \frac{|M(\Sigma)|}{|\Sigma|}.$$

The density not only captures the size of the covering, but also the degree of overlapping among cliques, which is an important measure for evaluation. For instance, if a covering of size $m$ contains a symbol $a$ that belongs to all $m$ cliques, each operation (e.g., BufferWrite, BufferRead) regarding $a$ must be iterated $O(m)$. On the other hand if the degree of overlapping is zero, then each operation regarding $a$ must be iterated $O(1)$ times. $d$ is such that:

$$d = \left\{ \begin{array}{ll} 1 & M(\Sigma) = \{\{a\} \mid a \in \Sigma \} \\ |\Sigma| & M(\Sigma) = \{\Sigma' \mid \Sigma' = \Sigma \} \end{array} \right.$$

Covering: This module implements $M(\Sigma)$ and requires $\Sigma$. It has been implemented with an indexed mapping exploiting Python hash tables (i.e., the Dict data type). This allows to access in $O(1)$ time any cliques that contain the symbol $a$. Instead of representing a clique covering with a direct hash table only, such as $\{0: [a, b, c], 1: [c, d], \ldots\}$, we also store an index $\{a: [0], b: [0], c: [0, 1], d: [1]\}$. The first of the two data structures is accessed less often w.r.t. the latter. A debugging method is also provided to generate a visual representation of the clique covering using the Graphviz library.

FSM: This module implements $A$. It holds an instance of $\Sigma$ to represent the alphabet, while (final) states are stored as a List of strings, i.e., $q_0, q_1, \ldots$. The transition function is a Dict that implements the $Q \times \Sigma \mapsto Q$ mapping. The Cartesian product is a Python Tuple; for instance, $(q_0, a): (q_3, (q_3, a): q_0, \ldots)$. The module also provides shorthand methods to test the readability of a character and the membership of a string.

The most important methods are those used to generate random testing data: strings, traces, automaton (of local type).

- The method $\text{rndString}(avglen:\text{int})$ generates an accepted string which length is approximately $\text{avglen}$. The randomization is obtained by assigning all the transitions the same probability to be trigger (uniform distribution over all the possible $\delta(q, a)$, for each $q$). At each state, one of the transitions is chosen at random and the corresponding symbol $a$ is appended to the string being constructed. Since the string must be accepted, then it is not known in advance whether the last symbol is appended to the final string exactly at the given length; thus, when the target length is reached, the string is returned at the next transition to a final state.

- The output of $\text{rndString}$ is used by $\text{rndTrace}(\text{c:Covering, swaps:int})$ to generate a random trace. Given a random accepted string and a clique covering of the dependence relation, the method performs $\text{swaps}$ permutations on randomly chosen digrams. The $\text{swap}$ parameter allows to control a measure of “distance” between the random trace and the original string.

- Last but not least, $\text{rndLocal}(\text{alphabet:Sigma, bSize:int})$ is
used to generate random automata of local type given the alphabet — which determines the number of states \(|Q| = |\Sigma| + 1\) — and the parameter \(b\text{Size}\). For each state the \(b\text{Size}\) parameter indicates the average number of arcs pointing to preceding states. If \(b\text{Size} = 0\) then the automaton is a chain, recognizing the string \(a_1a_2 \cdots a_{|\Sigma|}\) where all \(a_i \in \Sigma\) and \(\delta : \exists \forall (a_i, a_{i+1}) = a_{i+1}, \forall a_i \in \Sigma\). In general, \(\forall a_i \in Q \Rightarrow \exists b\text{Size} \cdot j : j \leq i, \delta(a_i, a_j) = a_j\). Obviously, the \(b\text{Size}\) parameter influences the lengths of the strings generated by traversing the automaton at random.

**Element:** This module implements elements of \(E\). It stores the current state as a string (e.g., \(q_0\), \(q_1\)) and the left cursors as a List of integers. Given an instance of this class, the method \(\text{inc}(a:\text{Char}, c:\text{Covering})\) increments of one unit all the left cursors the symbol \(a\) belongs to. Given another List of cursors \(R\) (i.e., the right cursors) the methods \(\text{existsEquals}\) and \(\text{equals}\) implements \(\exists j \mid e.\text{cursors}_i = R_i\) and \(\forall j \in e.\text{cursors} \mid e_j^{-1} = R_j\), where left implements \(e.\text{cursors}\).

**Decisor:** This module implements Algorithm 1 in two fashions: with and without code profiling. Code profiling records detailed information regarding the amount of time consumed by each single function point, to the granularity of one line of code. Also, the module provides the method \(\text{onAllHeads}(a:\text{Char})\) which implements the innermost if, with \(u_i = a\).

**V. RELATED WORK**

The contributions in this area are limited to a few, key approaches [4], [7], [8], [9]. Also, properties of traces focused on the MP are presented in [10], [3].

In [9] trace prefixes are exploited to solve the MP for trace languages in \(\mathit{Rot}(\Sigma)\). First, the algorithm inductively computes the prefixes \(\text{Pref}_t\) as \(\forall t' \in \text{Pref}_t(t) \Rightarrow \exists t'' \in \text{Pref}_{t-1}(t) \mid t' = t'' \cdot [a], a \in \Sigma\). The MP is reduced to checking whether the state \(\exists q \in Q \mid q \in Q_{\text{Pref}} \cap Q_\emptyset\). The set \(Q_{t=|t|} \subseteq Q\) is efficiently computed while constructing the prefixes: let \(Q_t\) be the set of states reachable by reading the trace prefixes \(\text{Pref}_{|t|-1}(t) = \{t_1, \ldots, t_i\} = \{t_j \mid t = t_j \cdot [a], a \in \Sigma, j = 1, \ldots, i\}\). Thus, \(Q_t = \bigcup_{j=1}^i \{q \in Q \mid q \in \delta(t', a_j), t' \in Q_{t_j}\}\). Prefixes are efficiently computed and stored as graph nodes \(V = \text{Pref}_t\); an edge exists for each pair of nodes \(t', t''\) s.t. \(t' = t'' \cdot [a]\). The time and space complexity are proven to be \(O(|t|^\alpha)\) and \(O(|t|^\alpha-1)\), respectively, with \(\alpha = \max_{\Sigma} |e.\sum| |\Sigma|\).

On the same direction, [8] assumes \(L \in CF(\Sigma)\) and proposes an algorithm having performances comparable w.r.t. the aforementioned approach. The worst case time complexity is still polynomial: \(O(|t|^{3\alpha})\). However, as underlined in [4] such a time complexity is unacceptable for practical purposes since the independence relation of common programs consisting of hundredths of instructions \((|t| \propto 10^2)\) may turn the complexity in an exponential function.

A recent work in [7], [4] focuses on both rational and local trace languages. An alternative prefix calculation technique is presented. An algorithm that solves the MP in \(O(|t|^\alpha)\) time is given. More precisely, our work is based on [4], which focuses on local languages. An algorithm based on the scheme of the Earley parser [11] is given to solve the MP. It assumes that \(T = [L] \in \mathit{Rot}(\Sigma)\) and requires a linear scan of the input to calculate the set of projections \(R(t)\) on maximal cliques. Based on \(R(t)\), an array of \(|t|+1 = n+1\) elements, \(E[0], E[1], \ldots, E[n+1]\) is constructed following a procedure driven by the automaton \(A : L = L(A)\). One symbol at a time is consumed on each projection according to the current state of \(A\). For instance, \(t\) is represented by \(R(t) = \{\pi_{\Sigma_1}(t) = ab, \pi_{\Sigma_2}(t) = bdd, \pi_{\Sigma_3}(t = cce\}\) and both \(a\) and \(c\) can be read on the current state of \(A\), then the procedure moves on both the first and the third projections. Formally, a cell \(E[i]\) is created at step \(i\). \(E[i], i = 1, \ldots, m\), where \(m = |M_\emptyset(\Sigma)|\) holds the data required by step \(i+1\). An element \(e_j \in E[i]\), stores \(1\) \(e_j.\text{state} \in Q\) the current state on \(A\), and \(2\) \(e_j.\text{cursors} \in \{0, 1, \ldots, \sigma\}\), the length of the prefix that has been read until step \(j\) on each of the \(m\) projections. For instance, if \(e_j \in E[2]\), \(e_j.\text{cursors} = \{1, 0, 1\}\) and \(e_j.\text{state} = q_s\) (shorten \(e_j = q_s(1, 0, 1)\)), at step \(2\) there exist a path on the automaton —reaching state \(q_3=\text{shorten}\) s.t. one symbol is consumed on the first and the third projection and no symbols are read on the second one. The algorithm requires \(O(|t|^\alpha)\) time in the worst case. Note that, this approach exploits \(\theta\) while [9], [8] utilizes \(T\).

**VI. CONCLUSIONS**

We defined the BM (Buffer Machine), a non-deterministic machine to solve the MP for rational trace languages. A tested implementation we released have been used to set up experiments on arbitrarily long inputs and complex commutative alphabets. As expected from theory the BM can solve the MP in polynomial time. In addition, we found that the size \(\sigma\) of the largest clique of the dependence relation influences the computation time only if the alphabet size is fixed. Otherwise, time is independent from \(\sigma\).

A deeper analysis of both time and space complexity is planned as future work, also in order to refine our results and to proof the existence of upper and lower bounds. Furthermore, parallelism among BM’s transition modes will be taken into account. In particular, we believe that some enqueque and dequeue operations can be executed concurrently without affecting the soundness of the machine. This improvement requires static analysis of the automaton and dependence relation.

**REFERENCES**


