2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

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 consist 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[1], 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 [**?**].

[1]Source code available at http://qelme.googlecode.com

II. Preliminaries

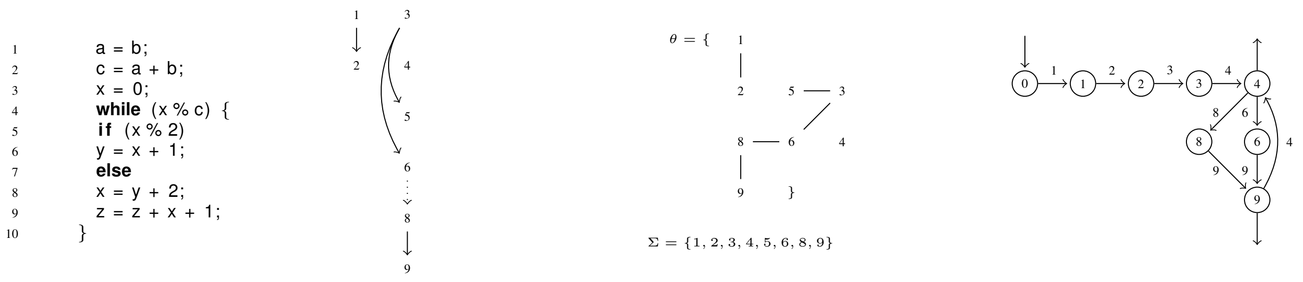
We use the conventional concept of *language* (or *string language*) L: a (sub)set of *strings* generated by a *free monoid* $\Sigma^* \supseteq L$, where Σ is the *alphabet*. Strings are denoted by smallcase letters: $u = a_1 a_2 \cdots a_n$, $v = b_1 b_2 \cdots b_m$, where $n = |u|, m = |v|$ are the length $|\cdot|$ 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* $\mathbb{F}(\Sigma, \mathcal{I})$, also called *partially commutative free monoid*. More formally, $\mathbb{F}(\Sigma, \mathcal{I}) = \Sigma^*/_{\equiv_\mathcal{I}}$, where $\mathcal{I} \subseteq \Sigma \times \Sigma$ is the *independence relation*. $\mathcal{I}$ is a symmetric and reflexive equivalence relation and $\equiv_\mathcal{I}$ is its *minimum congruence* over Σ^*. The *dependence* relation θ is also used as the complement of $\mathcal{I}$: $\theta = \mathcal{I}^c = \Sigma \times \Sigma \backslash \mathcal{I}$. Since $(a, a) \in \mathcal{I}, \forall a \in \Sigma$, the reflexive arcs are omitted if not strictly necessary.

A *trace language* T is a (sub)set of *traces* generated by $\mathbb{F}(\Sigma, \mathcal{I}) \supseteq T$, defined over the *commutative alphabet* $\langle \Sigma, \mathcal{I} \rangle$. More formally, $T = [L]_{\equiv_\mathcal{I}} = \{t \in \mathbb{F}(\Sigma, \mathcal{I}) \mid \exists u \in L : t = [u]\}$. The family of *rational trace languages* $Rat(\Sigma, \mathcal{I})$ is the focus of this work. $Rat(\Sigma, \mathcal{I})$ is proven to be generated by regular string languages [5]; i.e., $[L] = T \in Rat(\Sigma, \mathcal{I})$ 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 $\mathrm{Pref}_l(t)$ 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 '$\cdot$' of $\mathbb{F}(\Sigma, \mathcal{I})$ is s.t. $\forall t_1, t_2 \in \mathbb{F}(\Sigma, \mathcal{I})$ and $t_1 \cdot t_1 = t_1 t_1 = [uv]$, where $t_1 = [u]$ and $t_2 = [v]$. On languages, if $T_1 = [L_1], T_2 = [L_2]$, then $T_1 \cdot T_2 = \{t \in \mathbb{F}(\Sigma, \mathcal{I}) \mid t = t_1 \cdot t_2, t_1 \in T_1, t_2 \in T_2\}$. 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 $\mathcal{I}$ and θ can be represented with undirected graphs. Formally, in case of $\mathcal{I}$, $G = \langle V, E \rangle = \langle \Sigma, \mathcal{I} \rangle$. The notion of *clique covering* and *maximal clique* of a graph (i.e., of a relation) will be used. A *clique* $V_i \subseteq$ of 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 $\mathbb{M}_E(V) = \{V_1, \ldots, V_i, \ldots, V_k\}$ containing all


```
1   a = b;
2   c = a + b;
3   x = 0;
4   while (x % c) {
5   if (x % 2)
6   y = x + 1;
7   else
8   x = y + 2;
9   z = z + x + 1;
10  }
```

θ = { 1 — 2, 5 — 3, 8 — 6, 4, 8 — 9 }
Σ = {1, 2, 3, 4, 5, 6, 8, 9}

(a) Code (b) DDG (c) Dependence relation (d) Automaton

Figure 1: The trace representation of a snippet of code.

and only the maximal cliques.

Being $\mathbb{F}(\Sigma, \theta)$ represented by $G = \langle V = \Sigma, E = \theta\rangle$, then $\mathbb{M}_\theta(\Sigma)$ is such that $\forall i, j \in [1, |\mathbb{M}_\theta(\Sigma)|] \mid i \neq j \wedge V_i, V_j \in \mathbb{M}_\theta(\Sigma) \Rightarrow V_i \not\subseteq V_j$. The (maximal) cliques are subsets of Σ, $\Sigma_1, \Sigma_2, \ldots, \Sigma_k \subseteq \Sigma$, with $k \geq 1$. Similarly, $\mathbb{M}(\Sigma, \mathcal{I})$ can be defined for $\mathcal{I}$.

A *projection* of a trace on Σ' is a morphism $\pi_{\Sigma'}(\cdot)$ defined as $\pi_{\Sigma'} : \Sigma^* \mapsto \Sigma'^*$: given $u = wa$, $\pi_{\Sigma'}(u) = \pi_{\Sigma'}(w)a$ if $a \in \Sigma'$; otherwise $\pi_{\Sigma'}(u) = \pi_{\Sigma'}(w)$, or $\pi_{\Sigma'}(\varepsilon) = \varepsilon$. Thus, $\pi_{\Sigma'}(u) = u' \in \Sigma'^*$. Since $\Sigma' \in \mathbb{M}(\Sigma)$, then Σ' are cliques. $\pi_{\Sigma'}(t)$ maintains the order of the symbols of t; that is, $\forall a_i, a_j \in \pi_{\Sigma'}(t) \wedge i < j \Rightarrow \exists i' < j' \mid a_{i'} = a_i \wedge a_{j'} = a_j \wedge a_{i'}, a_{j'} \in u$.

III. The Membership Problem for Trace Languages

We describe the MP, motivate why it is important in the context of traces, and detail the specification of the machine to solve it. Due to space limitations, proofs are omitted.

The MP is the problem of deciding whether or not a given string $u \in \Sigma^*$ is s.t. $u \in [t]$ where $[t] \in T$ is a given trace language. Despite the simplicity of its formulation, the MP plays a key role in real-world applications where classical dynamic programming techniques are used. Problems that can be described by means of a set of binary and symmetric constraints among a finite set of symbols (e.g., events, actions) also find an *effective* formalization in the framework of traces. For instance, consider the problem of determining whether or not a certain sequence of program instructions is an acceptable schedule. Each instruction is represented by a symbol in Σ while the dependency (e.g., *Data Dependency Graph* (DDG), *Concrete Dependency Graph* (CDG)) relations among them are approximated with θ. Figure 1 gives a simplified (i.e., no nested loops) example of the trace language associated to a program. *Direct* and *loop-carried* [6] data dependencies are represented by *solid* and *dotted* arcs, respectively. The resulting θ is the reflexive symmetric closure of the DDG with *both* the types of arcs are taken into account.

However, it must be noted that θ "encodes" less constrains than the DDG since $i — j \in \theta$ implies that the instruction i depends on j and vice-versa. In other words, *Read After Write* (RAW) dependencies of the DDG are transformed into *Write After Write* (WAW) dependencies of θ; WAW dependencies-remain WAW.

A. Solving the MP for Rational Trace Languages

The BM (Figure 2) has one input tape, a finite set of *First-In First-Out* (FIFO) *buffers*, and a finite-state *control device*. Each buffer is an infinite sequence of adjacent cells while the input tape is left-bounded. The BM moves by consuming one symbol at a time from the leftmost position of the input tape (*pop*), dispatches it onto an appropriate buffer (*enqueue*) and updates the state of the control device, accordingly. More formally, a buffer is defined as follows.

Definition 1 (Buffer): Given an alphabet Σ_i, called the *buffer alphabet*, a *buffer* b_i is a FIFO queue containing only symbols in Σ_i:$b_i := \rangle a_I \cdots a_j \cdots a_O\rangle$ where "$\rangle a_I \cdots$" is the input side while "$\cdots a_O\rangle$" is the output side. A buffer with no symbols is *empty* and is indicated as $b_i = \rangle\varepsilon\rangle = \varnothing$.

Each buffer b_i, short b_{Σ_i}, is associated to a subset $\Sigma_i \subset \Sigma$ 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 $\Sigma_i = \{a, c\} \subset \Sigma = \{a, b, c, d, e\}$, then $b_i = b_{\Sigma_i = \{a,c\}}$ holds symbols drawn from Σ_i only. Buffers are associated to the following functions.

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

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

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

- It is straightforward that if $b_{\Sigma_{i=\{a,c\}}} = \rangle ab\rangle$ and $b_{\Sigma_{j=\{d,e\}}} = \rangle\varepsilon\rangle$, then $\mathrm{Empty}(b_{\Sigma_i}) = \bot$ while $\mathrm{Empty}(b_{\Sigma_j}) = \top$.
- If $b_{\Sigma_{i=\{a,c\}}} = \rangle aaca\rangle$, $\mathrm{Dequeue}(b_{\Sigma_i})$ returns $\{a\}$ and, as a result, $b_{\Sigma_i} = \rangle aac\varepsilon\rangle$.
- Given $b_{\Sigma_i} = \rangle aac\rangle$, $\mathrm{Enqueue}(b_{\Sigma_i}, a)$ would result in $b_{\Sigma_i} = \rangle \underline{a}aac\rangle$. This function is such that if $b_{\Sigma_i} = \rangle aac\rangle$, invoking $\mathrm{Enqueue}(b_{\Sigma_i}, \varepsilon)$ results in $b_{\Sigma_i} = \rangle\varepsilon aac\rangle = \rangle aac\rangle$.

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* $\mathscr{B}_a$

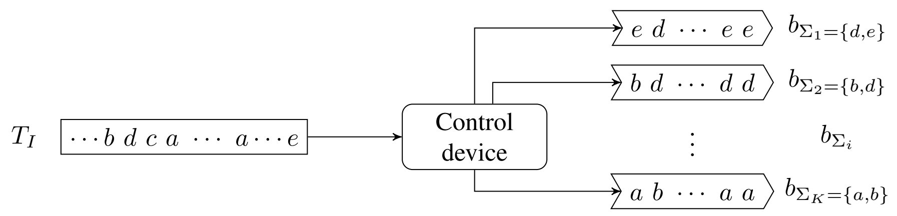


Figure 2: A visual representation of a BM. A sample trace language having K cliques in its dependence relation, ϑ, is used.

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: $\mathscr{B}_a = \{b_{\Sigma_i} \mid a \in \Sigma_i\}$.

For instance, consider $\mathbb{M}_\theta(\Sigma) = \{\Sigma_1 = \{a\}, \Sigma_2 = \{b,c\}, \Sigma_3 = \{c,d\}\}$ and its associated buffers $\{b_{\Sigma_1}, b_{\Sigma_2}, b_{\Sigma_3}\}$. The family of buffers specific to c is $\mathscr{B}_c = \{b_{\Sigma_1}, b_{\Sigma_3}\}$, 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 $b_{\Sigma_1}, \dots, b_{\Sigma_k}$ be a set of buffers and $\Gamma = \{\Sigma_i\}_N$ a set of N alphabets. A *family of buffers* $\mathscr{B}_\Gamma$ *specific to a set of alphabets* Γ is defined as the set of all the buffers b_{Σ_i} whose alphabet Σ_i is in Γ: $\mathscr{B}_\Gamma = \{b_{\Sigma_i} \mid \Sigma_i \in \Gamma\}$.

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

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

- $\mathrm{Empty}(\mathscr{B}') \iff \bigwedge_{b_{\Sigma_i} \in \mathscr{B}'} \mathrm{Empty}(b_{\Sigma_i})$.
- $\mathrm{Dequeue}(\mathscr{B}')$ returns the set of symbols $\Sigma' = \{a_i \mid \{a_i\} = \mathrm{Dequeue}(b_{\Sigma_i}) \wedge b_{\Sigma_i} \in \mathscr{B}'\}$.
- $\mathrm{Enqueue}(\mathscr{B}', a)$ is such that $\mathrm{Enqueue}(b_{\Sigma_i}, a)$ is executed $\forall b_{\Sigma_i} \in \mathscr{B}'$.

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

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

The input tape is denoted as $T_I = a_1 a_2 \cdots a_n = |a_n \cdots a_2 a_1\rangle$ and can hold any symbol drawn from Σ. The $\mathrm{Enqueue}(T_I, a)$ 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 $Rat(\Sigma, \theta) \ni T = [L] \subseteq \mathbb{F}(\Sigma, \theta)$ is formally defined as follows. Note that $A : L = L(A)$ is known.

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

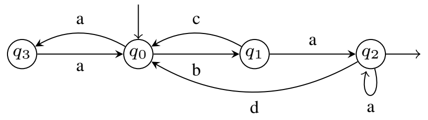
- $\mathscr{B} = \mathscr{B}_{\mathbb{M}_\theta(\Sigma)} = \{b_1, b_2, \dots, b_i, \dots, b_K\} = \{b_{\Sigma_1}, b_{\Sigma_2}, \dots, b_{\Sigma_i}, \dots, b_{\Sigma_K}\} \in \mathbb{B}$ is the *family of buffers* associated to $\mathbb{M}_\theta(\Sigma)$.
- $A = \langle \mathbb{Q}, \Sigma, \delta, q_0, \mathbb{Q}_F\rangle$ is the *deterministic recognizer* of $L = L(A)$, a finite state automaton: $\mathbb{Q}$ and $\mathbb{Q}_F \subseteq \mathbb{Q}$ are the finite set of states and acceptance states, respectively; and δ is the transition function.
- $\tau : \mathbb{Q} \times \Sigma \times \mathbb{B} \mapsto \wp(\mathbb{Q} \times \mathbb{B})$ 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: fires if $a \neq \varepsilon$ (i.e., $\mathrm{Empty}(T_I) = \bot$) and is s.t. $\tau(q, a, \mathbb{B}) = \langle q, \mathbb{B}\rangle$. It is always followed by a BufferWrite.
  - BufferWrite: is s.t. $\mathbb{B}' : \mathrm{Enqueue}(\mathscr{B}_a, a)$, $\tau(q, a, \mathbb{B}) = \langle q, \mathbb{B}'\rangle$.
  - BufferRead: fires if $\mathrm{Dequeue}(\mathscr{B}_a) = \{a'\}$ and it is s.t. $\tau(q, \varepsilon, \mathbb{B}) = \langle q' = \delta(q, a'), \mathrm{Dequeue}(\mathscr{B}_{a'})\rangle$

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

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

The Read transition reads one symbol from T_I; if it cannot be performed because $a = \varepsilon$ then a BufferRead is performed. Each Read is followed by a BufferWrite which enqueues the read symbol a on all the buffers in $\mathscr{B}_a$ (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 $M_1 = \langle A_1, \tau, \mathscr{B}, T_I\rangle$ where A_1 is as follows.



Also, $T_I = |bdca\rangle$, $\mathscr{B} = \{b_{\Sigma_1}, b_{\Sigma_2}\}$ and $b_{\Sigma_1} = b_{\{a,b,c\}} = \rangle abb\rangle$, $b_{\Sigma_2} = b_{\{b,d,e\}} = \rangle dbbed\rangle$. In this configuration:

- A BufferRead cannot be performed from any of the buffers since $\nexists b_{\Sigma_i} \in \mathscr{B} : \mathrm{Dequeue}(b_{\Sigma_i}) = \{a'\} \wedge q' = \delta(q_0, a')$ for some q'.
- A Read is performed and the current symbol a is consumed from T_I.
- Thus, a BufferWrite is performed: the symbol a is enqueued to all the buffers b_{Σ_i} such that $a \in \Sigma_i$. In this case b_{Σ_1} becomes $b'_{\Sigma_1} = a\rangle abb\rangle$ hence: $\mathscr{B} = \{\rangle aabb\rangle, \rangle dbbed\rangle\}$.

On the other hand, if buffers are such that: $b_{\Sigma_1} = b_{\{a,b,c\}} = \rangle abcc\rangle$, $b_{\Sigma_2} = b_{\{b,d,e\}} = \rangle dbed\rangle$; then a BufferRead can be performed on $b_{\Sigma_1} = \rangle abc\rangle c$ and the configuration of M_1

changes: $\mathscr{B} = \{\rangle abc\rangle, \rangle dbbed\rangle\}$, and the current state of the machine becomes $q' = q_0$.

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

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

Proposition 1: $\mathscr{B}$ is a family of buffers specific to the set Γ of alphabets, s.t. $\bigcup_{\Sigma_i \in \Gamma} \Sigma_i = \Sigma$.

Proof: $\mathbb{M}_\theta(\Sigma)$ is a covering of *all* the alphabet: $\cup_{\Sigma_i \in \mathbb{M}_\theta(\Sigma)} = \Sigma$. Note that, $\Gamma = \mathbb{M}_\theta(\Sigma)$. By definition, one buffer b_{Σ_i} exists for each $\Sigma_i \in \mathbb{M}_\theta(\Sigma) = \Gamma$. Thus, each $b_{\Sigma_i} \in \mathscr{B}$ is s.t. $a \in \Sigma_i \Leftrightarrow a \in \Sigma$. If $\exists a' \notin \Sigma$ then a' does not belong to any of the cliques $\Sigma_i \in \mathbb{M}_\theta(\Sigma)$ otherwise it would belong to Σ as well, raising a contradiction. However, it could be that $\Sigma_i \cap \Sigma_j \neq \varnothing$ for some i, j. ■

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

Proof: Let us assume that $u \in [t]$ but $\hat{\tau}(q_0, u, \{\varnothing, \ldots, \varnothing\}) \cap M_F = \emptyset$. Thus, 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 $M^{C'} = \langle q, \{\varnothing, \ldots, \varnothing\}\rangle$ where $q \notin \mathbb{Q}_F$;
2) M ends up at configuration $M^{C''} = \langle q_F, \mathscr{B}\rangle$ s.t. $\exists b_{\Sigma'} \in \mathscr{B}'' \mid \text{Empty}(b_{\Sigma'}) = \bot \wedge q \in \mathbb{Q}_F$;
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 $L \neq L(A)$. $\lightning$
2) $\Rightarrow u$ is accepted by A since $q_F \in \mathbb{Q}_F$. Also, $\exists_{\geq 1} a' \in b_{\Sigma'}$ that has not 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 τ (Definition 7). $\lightning$
   b) a' cannot be dequeued, but this must be that, either:
      i) $\nexists q' \mid \delta(q', a')$, but this implies that $q \notin \mathbb{Q}_F$.
      ii) $a' \notin \Sigma'$, but this implies that Σ' is not the alphabet of $b_{\Sigma'}$, otherwise a' would not be in $b_{\Sigma'}$.

      Thus, i. and ii. contradict the definition of Buffer-Read mode of τ (Definition 7).
   c) (straightforward) the Read preceding the Buffer-Write has read a wrong symbol, that contradicts the definition of Read and also implies that $a' \notin u$.

   Hence, 2. leads to contradictions. $\lightning$
3) (straightforward) $\Rightarrow$ the machine is not at the end of the computation. $\lightning$

■

Note 1: An alternative way to proof Proposition 2 consists in showing that it implements Algorithm 1, which can ben 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 $C = \mathbb{M}_\theta(\Sigma)$ 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.
- Π is the set of projection on cliques, used to emulate the buffers (see Definition 1 and 7).
- $\underline{\underline{M}}$ is a matrix of size $|C| \times |\mathbb{Q}|$ initialized to $\bot$. 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 $M : E \mapsto \{\top, \bot\}$ defined as $M(e) = \underline{\underline{M}}(e.state, e.cursors)$.
- $\underline{\underline{R}}$ is a vector of cursors, one per clique. Each element of the vector is $R_j \in \{0, ..., |\pi_{\Sigma_j}(u)|\}$ 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 $e.cursor_j$ to its tail (i.e., end of the projection).

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

Buffer Function	Implemented Emulation
$\text{Enqueue}(b_{\Sigma_j}, a)$	$\text{Enqueue}(\pi_{\Sigma_j}(u), a) \wedge R_j \leftarrow R_j + 1$
$\text{Dequeue}(b_{\Sigma_j})$	$e.cursors_j \leftarrow e.cursors_j + 1, \forall e \in E$
$\text{Empty}(b_{\Sigma_j})$	$\forall e \in E \Rightarrow e.cursors_j = R_j$

Table I: Emulation of buffer functions.

```
Input
$A \leftarrow \langle \mathbb{Q}, \Sigma, \delta, q_0, \mathbb{Q}_F \rangle$                /* The automaton */
$C \leftarrow \Sigma_1, \ldots, \Sigma_m$                /* Clique covering */
$u \leftarrow u_1 \cdots u_n$                /* The string */
Variables
$decision \leftarrow \bot$
$E \leftarrow \{\}$                /* The working list */
$\Pi(u) \leftarrow \{\pi_{\Sigma_1}(u), \ldots, \pi_{\Sigma_m}(u)\} = \{\varepsilon, \ldots, \varepsilon\}$                /* The projections */
$\underline{\underline{M}} : C \times \mathbb{Q} \mapsto \{\top, \bot\}$                /* Initialized to ⊥ */
$\overline{R} \leftarrow \langle \underbrace{0, \ldots, 0}_{m} \rangle$                /* Right cursors */

while $i \leq 2n \wedge \neg decision$ do
    $E' \leftarrow \{\}$
    if $i \leq n$ then
        forall $\Sigma' \in I(u_i)$ do
            Enqueue($\pi_{\Sigma'}(u), u_i$)                /* Perform a BufferWrite */
        end
    end
    forall $e \in E$ do
        if $\exists j \mid e.cursors_j = R_j$ then $E' \leftarrow E' \cup e$
        for $j \leftarrow 0; j < m; j \leftarrow j + 1$ do                /* For all the cursors */
            if $e.cursors_j < R_j$ then
                $u_s \leftarrow (\pi_{\Sigma_j}(u))_{e.cursors_j}$                /* Perform a BufferRead */
                if $\exists \delta(e.state, u_s)$ then                /* Check readability */
                    if $\forall \Sigma_k \in I(u_s) \Rightarrow (\pi_{\Sigma_k}(u))_{e.cursors_k} = u_s$ then
                        Update($u_s, e, E', C, R, M, A$)
                    end
                end
            end
        end
    end
    $i \leftarrow i + 1$                /* Move the virtual cursor one step ahead */
    $E \leftarrow E'$                /* Update the next working list */

end
```

Algorithm 1: Deterministic algorithm implementing the BM.

Furthermore, Algorithm 1 relies on the shorthand procedure $\mathrm{Update}(\cdot)$ (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 $e.cursors_k$ of the k-th clique, that is the symbol $(\pi(u)_{\Sigma_k})_{e.cursors_k}$.

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 θ 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., 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 **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 $e.cursors_j < R_j$), then the symbol on the top of the buffer (i.e., projection) is read and Update is invoked.

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.

Sigma: This module implements Σ. The internal representation is a Python `List`. Methods to generate random strings

```
Input
$u_s$                                                  /* Current symbol */
$e'$                                                   /* Current element */
$E'$                                                   /* The working list */
$C$                                                    /* Clique covering */
$R$                                                    /* Right cursors */
$M$                                                    /* Marking of $E'$ */
$A \leftarrow \langle \mathbb{Q}, \Sigma, \delta, q_0, \mathbb{Q}_F \rangle$              /* The automaton */

procedure Update($u_s, e', E', C, R, M, A$) begin
    $e' \leftarrow e$                                       /* Temporary copy */
    $e'.state \leftarrow \delta(e'.state, u_s)$              /* Assign the new state to the element */
    forall $\Sigma_k \in I(u_s)$ do                          /* For all the cliques */
        $e.cursors_k \leftarrow e.cursors_k + 1$             /* Update the left cursor */
    end
    if $M(e') = \bot \wedge \exists j \mid e'.cursors_j = R_j$ then      /* If not existing */
        $E' \leftarrow E' \cup \{e'\}$                       /* Add it to the next working list */
        $M(e') \leftarrow \top$                              /* Mark it as existing */
        $decision \leftarrow e'.state = q \wedge q \in \mathbb{Q}_F \wedge \forall j \Rightarrow e'.cursors_j = R_j$
    end
end
```

Algorithm 2: The procedure Update used by Algorithm 1.

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 $\mathbb{M}(\Sigma)$ be a clique covering of the alphabet Σ. The *density* of $\mathbb{M}(\Sigma) = \{\Sigma_1, \ldots, \Sigma_m\}$ is

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

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* the 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:

$$d = \begin{cases} 1 & \mathbb{M}(\Sigma) = \{\{a\} \mid a \in \Sigma\} \\ \frac{1}{|\Sigma|} & \mathbb{M}(\Sigma) = \{\Sigma' \mid \Sigma' = \Sigma\} \end{cases}$$

Covering: This module implements $\mathbb{M}(\Sigma)$ 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], ...`}, 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 represents the alphabet, while (final) states are stored as a `List` of strings, i.e., `q0, q1`, etc. The transition function is a `Dict` that implements the $\mathbb{Q} \times \Sigma \mapsto \mathbb{Q}$ mapping. The Cartesian product is a Python `Tuple`; for instance, {`(q0, a): q3, (q3, a): q0, ...`}. 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 rndString(avglen:int) generates an *accepted* string which length is *approximately* 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 rndString is used by rndTrace(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 swaps permutations on randomly chosen digrams. The swap parameter allows to control a measure of "distance" between the random trace and the original string.
- Last but not least, rndLocal(alph:Sigma, bSize:int) is

used to generate random automata of local type given the alphabet — which determines the number of states ($|\mathbb{Q}| = |\Sigma| + 1$) — and the parameter bSize. For each state the bSize parameter indicates the average number of arcs pointing to preceding states. If bSize = 0 then the automaton is a chain, recognizing the string $a_1 a_2 \cdots a_{|\Sigma|}$ where all $a_i \in \Sigma$ and $\delta : \exists \delta(a_i, a_{i+1}) = a_{i+1}, \forall a_i \in \Sigma$. In general, $\forall a_i \in \mathbb{Q} \Rightarrow \exists_{=\mathsf{bSize}} a_j : j \leq i, \delta(a_i, a_j) = a_j$. Obviously, the bSize parameter influences the length 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., `q0, q1`) and the left cursors as a `List` of integers. Given an instance of this class, the method inc(a:Char, c: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 existsEquals and equals implements $\exists j \mid e.cursors_j = R_j$ and $\forall j \in e.cursors \mid e_j^{-1} = R_j$, where left implements $e.cursors$.

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 onAllHeads(a:Char) which implements the innermost **if**, with $u_s =$ a.

V. Related Work

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

In [9] trace prefixes are exploited to solve the MP for trace languages in $Rat(\Sigma)$. First, the algorithm inductively computes the prefixes $\mathrm{Pref}(t) = \{t_i\}$ as $\forall t' \in \mathrm{Pref}_l(t) \Rightarrow \exists t'' \in \mathrm{Pref}_{l-1}(t) \mid t' = t'' \cdot [a], a \in \Sigma$. The MP is reduced to checking whether the state $\exists q \in \mathbb{Q} \mid q \in \mathbb{Q}_F \cap \mathbb{Q}_t$. The set $\mathbb{Q}_{t=[u]} \subseteq \mathbb{Q}$ is efficiently computed while constructing the prefixes: let $\mathbb{Q}_{t_j}$ be the set of states reachable by reading the trace prefixes $\mathrm{Pref}_{|t|-1}(t) = \{t_1, \ldots, t_i\} = \{t_j \mid t = t_j \cdot [a_j], a_j \in \Sigma, j = 1, \ldots, i\}$. Thus, $\mathbb{Q}_t = \cup_{j=1}^{i} \{q \in \mathbb{Q} \mid q \in \delta(t', a_j), t' \in \mathbb{Q}_{t_j}\}$. Prefixes are efficiently computed and stored as graph nodes $V = \mathrm{Pref}(t)$; 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 in $O(|t|^{\alpha})$ and $O(|t|^{\alpha-1})$, respectively, with $\alpha = \max_{\Sigma' \in \mathbb{M}_\theta(\Sigma)} |\Sigma'|$.

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 $O(|t|^{\sigma})$ 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 $T = [L] \in Rat(\Sigma)$ 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 $R(t) = \{\pi_{\Sigma_1}(t) = abb, \pi_{\Sigma_2}(t) = bdd, \pi_{\Sigma_3}(t) = cec\}$ 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 $m = |\mathbb{M}_\theta(\Sigma)|$ holds the data required by step $i+1$. An element $e_j \in E[i]$, stores (1) $e_j.state \in \mathbb{Q}$ the current state on A, and (2) $e_j.cursors \in \{0, 1, \ldots, \sigma\}^m$, the length of the prefix that has been read untill step j on each of the m projections. For instance, if $e_j \in E[2]$, $e_j.cursors = \langle 1, 0, 1\rangle$ and $e_j.state = q_3$ (shorten $e_j = q_3\langle 1, 0, 1\rangle$), at step 2 there exist a path on the automaton —reaching state q_3— 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 $O(|t|^{\sigma})$ time in the worst case. Note that, this approach exploits θ while [9], [8] utilizes $\mathcal{I}$.

VI. Conclusions

We defined the BM (Buffer Machine), a non-deterministic machine to solve the MP for rational trace languages. A testbed 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 σ of the largest clique of the dependence relation influences the computation time only if the alphabet size is fixed. Otherwise, time is independent from σ.

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. Furtherermore, parallelism among BM's transition modes will be taken into account. In particular, we believe that some enqueue 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

[1] Mazurkiewicz, A.: Concurrent program schemes and their interpretations. DAIMI Rep. PB **78** (1977)

[2] Cartier, P., Foata, D.: Problèmes combinatoires de commutation et réarrangements. (1969)

[3] Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific (1995)

[4] Savelli, A.: Two contributions to automata theory on parallelization and data compression. PhD thesis, Politecnico di Milano and Université de Marne-la-Vallée (2007)

[5] Szijarto, M.: A classification and closure properties of languages for describing concurrent system behaviours. Fundam. Inform. **4**(3) (1981) 531–550

[6] Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**(4) (1994) 345–420

[7] Breveglieri, L., Crespi Reghizzi, S., Savelli, A.: Efficient Word Recognition of Certain Locally Defined Trace Languages. In: Proceedings of the 5th International Conference on WORDS, Montréal (QC) Canada, Université du Québec a Montréal (September 2005)

[8] Avellone, A., Goldwurm, M.: Analysis of algorithms for the recognition of rational and context-free trace languages. RAIRO Informatique théorique et Applications **32** (1998) 141–152

[9] Bertoni, A., Mauri, G., Sabadini, N.: Membership problems for regular and context-free trace languages. Information and Computation **82**(2) (1989) 135–150

[10] Rytter, W.: Some properties of trace languages. Fundamenta Informaticae **7** (1984) 117–127

[11] Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13**(2) (1970) 94–102