DEI, Politecnico di Milano Dottorato in Ing. dell'Informazione

Ricerca Minore

Specification and Evaluation of an Efficient Recognizer for Rational Trace Languages

Federico Maggi

XXII ciclo
Rapporto Interno TR-2008-23

Sommario

Proponiamo una versione ad una sola scansione di un algoritmo Earley-like per il problema dell'appartenenza nei linguaggi traccia razionali. L'algoritmo, che riconosce in tempo polimoniale, è inizialmente descritto attraverso la specifica formale della *Non Deterministic Buffer Machine* (NDBM); in secondo luogo, la procedura di riconoscimento è dettagliata attraverso un algoritmo deterministico, con alcuni esempi per chiarirne il funzionamento. Inoltre, descriviamo l'implementazione prototipale dell'algoritmo e dell'applicativo di test utilizzato per valutare empiricamente le performance e le caratteristiche della soluzione proposta. A questo scopo, abbiamo costruito, ed ivi descriviamo, anche dei generatori di dati (e.g., automi, stringhe, tracce) pseudo-casuali.

Minor Research

Specification and Evaluation of an Efficient Recognizer for Rational Trace Languages

Federico Maggi

Cycle XXII
Scientifical Report TR-2008-23

Abstract

An improved, one-pass version of a two-pass, Earley-like recognition algorithm is here proposed to solve the Membership Problem for rational trace languages in polynomial time. The algorithm is first described through the formal specification of what we called a *Non Deterministic Buffer Machine* (NDBM); secondly, the recognition is detailed through a deterministic algorithm along with some running examples. In addition, we describe our prototype implementation of the algorithm used to empirically evaluate the performances and the characteristics of the proposed solution. To this end, we designed pseudo-random testing data generators that are here described as well.

1 Introduction

The relevance of instruction parallelization and optimal event scheduling is currently increasing, not only in the field of compiler and language translator design [Appel and Palsberg, 2002] but also in the optimization of software pipelining [Lam, 1988] and transactional processing of database operations. In particular, because of the high computational power available today, the industrial interest on automatic code parallelization [Bacon et al., 1994] is increasing notably.

In the last years, several contributions have arisen in these fields, exploiting the *theory of traces* [Mazurkiewicz, 1977; Cartier and Foata, 1969; Diekert and Rozenberg, 1995] to approach the problem of translating a sequence of statements into a parallelized form [Keller, 1973a,b] which execution leads to the same effect in the running environment. Dated back around 1970 [Mazurkiewicz, 1977], the theory of traces provides a powerful mathematical formalism that can be effectively used to model concurrent executions of *events*, to study and proof properties of interest for the specific purposes.

The connection between the modeled phenomenon and the trace theory stands in the use of traces to represent the behavior of the system under consideration. The occurrences of permitted events are defined as *symbols* that are drawn from a finite set called *alphabet*. Thus, each execution path is modeled as sequence of symbols: a *string*. Relying on the definitions from set theory and formal languages, *traces* can be viewed as a natural extension of the concept of strings: a trace is a string itself. Moreover, a trace is (the representative element of) a *class of strings*, that is generated by applying a finite number of allowed commutations on the representative element. Hence, without going into the details, the behavior of a system can be represented by one trace which somehow "embeds" all the runs allowed by the rules of the system itself. Such rules are formalized by mean of a dependence relation, in terms of commutations among symbols.

The relevance of this formalism is twofold: first of all, it provides a plain and abstract framework that can be applied to a wide variety of problems as it could virtually characterize any kind of execution scheme. Secondly, the theory of traces has its origins in combinatorics and formal languages; thus, from an algorithmic point of view, the solution of decision problems with traces can effectively take advantage from results and rich knowledge in both the two classic disciplines.

Despite the simplicity of its formulation and to the well established theoretical background, the research on traces and trace languages is still active. From a purely theoretical perspective, the research community has developed the theory of traces in strict relationship with formal languages, combinatorics, data structures, graph theory and algebra. There is also a non negligible amount of open problems that need to be further investigated. Instead of focusing on a direct application of the contribution in the literature, in this paper we analyze a common, shared problem in the theory of traces, the *Membership Problem* (MP). Intuitively, the MP is the problem of *deciding* whether or not a certain string belongs (i.e., is member of) a *class* of strings represented by a trace, or a trace language. The MP is a relatively new challenge and thus the solutions are limited to some basic approaches that will be reviewed in this paper. From the perspective of formal languages, the MP can be solved by designing an efficient parsing algorithm to decide whether a string is a valid sequence of symbols in the trace language; indeed, almost all the approaches go in such a direction [Bertoni et al., 1989; Avellone and Goldwurm, 1998; Breveglieri et al., 2005; Savelli, 2007].

Our contribution consists in an improved, one-pass version of a two-pass parsing algorithm [Savelli, 2007] proposed to solve the MP in polynomial time. In addition, we provide a formal specification of the abstract machine that decides the MP. Secondly, we provide an algorithmic description and analysis of the proposed parsing procedure and finally we investigate the empirical performances of a prototype implementation.

Before going into the details of the MP (Section 3) and the state-of-the-art, in Section 2 we introduce the basic definitions and the notation used throughout this paper. In Section 4 we define our algorithm for solving the MP along with a few execution samples. In Section 5 the prototype implementation is detailed and the experimental measurement results are analyzed.

2 Basic definitions

The notation, the functions and the operators used in the following are slightly modified and adapted versions taken from [Savelli, 2007; Avellone and Goldwurm, 1998; Pighizzini, 1994].

We will explicitly refer to *string* languages meaning the usual concept of a *language* L: the set of *strings* belonging to a subset of the free monoid $\Sigma^* \supseteq L$ defined over the finite *alphabet* of symbols $\Sigma = \{a, b, c, \dots\}$. Strings will be denoted by small 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 denoted by a string t with square brackets

$$[t] = \{t_1, t_2, \dots, t_k\} \tag{1}$$

to indicate that t is actually the representative element of an equivalence class. The elements of the equivalence class $[t]$ are strings that are drawn from a *partially commutative monoid* which is the quotient set $\Sigma^*/_{\equiv_\mathcal{I}}$, called the *trace monoid.* $\mathcal{I} \subseteq \Sigma \times \Sigma$ is a symmetric and reflexive equivalence relation called *independence relation.*

The trace monoid is then the quotient of the free monoid Σ^* w.r.t. the trace independency equivalence relation $\equiv_\mathcal{I}$ defined over $\mathcal{I}$. For instance, if $(a, b) \in \mathcal{I}$ (or $a — b \in \mathcal{I}$) then the two strings $u = ab, v = ba$ are $\mathcal{I}$-equivalent, thus we can write $u = ab \equiv_\mathcal{I} v = ba$. Hence, $[t]_{\equiv_{\mathcal{I}=\{(a,b)\}}} = [ab]_{\equiv_{\mathcal{I}=\{(a,b)\}}} = \{ab, ba\}$ is an example of a trace. Moreover, given $\Sigma = \{a, b, c, d\}$, we say that the string $t = abcbad$ generates $[t]_{\equiv_\mathcal{I}} = \{abcbad, bacbad, abcabd, bacabd\}$.

In comparison to a string language, a *trace language* T is the set of all the *traces* belonging to a subset of the trace monoid, $\mathbb{F}(\Sigma, \mathcal{I}) = \Sigma^*/_{\equiv_\mathcal{I}} \supseteq T$, defined over the *commutative alphabet* (or *independence alphabet*) $\langle \Sigma, \mathcal{I} \rangle$. Hence, a trace language T is (the representative element of) the equivalence class $[L]_{\equiv_\mathcal{I}}$ over the string language with the same alphabet Σ, that is:

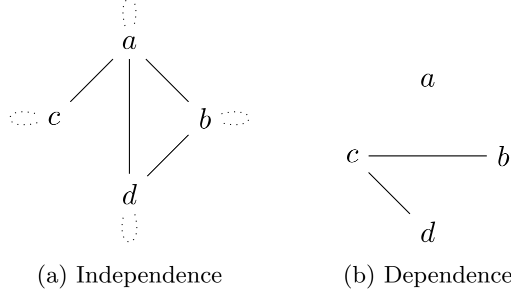
$$T = [L]_{\equiv_\mathcal{I}} = \{t \in \mathbb{F}(\Sigma, \mathcal{I}) \mid \exists u \in L : t = [u]\} \tag{2}$$

The corresponding *dependence* relation θ is also used instead of $\mathcal{I}$ to define a partially commutative alphabet: $\theta = \mathcal{I}^c = \Sigma \times \Sigma \backslash \mathcal{I}$ is the complement of $\mathcal{I}$. Note that since $(a, a) \in \mathcal{I}, \forall a \in \Sigma$, the reflexive relations are omitted if not strictly necessary (in Figure 1a are denoted by dotted arcs).

As for string languages, the *product* operator of the monoid $\mathbb{F}(\Sigma, \mathcal{I})$ is well defined; it is such that $\forall t_1, t_2 \in \mathbb{F}(\Sigma, \mathcal{I})$, $t_1 \cdot t_1 = t_1 t_1 = [uv]$, where $t_1 = [u]$ and $t_2 = [v]$. The product can be extended to trace languages: if $T_1 = [L_1], T_2 = [L_2]$, then $T_1 \cdot T_2 = \{t \in \mathbb{F}(\Sigma, \mathcal{I}) \mid t = t_1 \cdot t_2, t_1 \in T_1, t_2 \in T_2\}$.

In a similar vein, the Kleene star operation on traces is defined as $t^* = \cup_{n=0}^{+\infty} t^n$ where $t^0 = \epsilon = [\epsilon]$ is the empty trace, and $t^n = t \cdot t^{n-1}$. This definition can be extended to trace languages: $T^* = \cup_{n=0}^{+\infty} T^n$, $T^0 = \{\epsilon\}$, and $T^n = T \cdot T^{n-1}$.

The focus of this work is the family of *rational trace languages* $Rat(\Sigma, \mathcal{I})$ that has been proven [Szijarto, 1981] to be defined by the class of regular languages; the trace



(a) Independence (b) Dependence

Figure 1: Graphical representation of (a) the independence relation $\mathcal{I} = \{(a,a),(b,b),(c,c),(d,d),(a,b),(b,d),(c,a),(d,a)\}$; and (b) the corresponding dependence relation $\theta = \mathcal{I}^c = \{(b,c),(c,d)\}$.

language T is rational if and only if $T = [L]$ for some language $L \in Reg(\Sigma)$. $Rat(\Sigma, \mathcal{I})$ is the smallest class of trace languages containing all finite sets and closed under the operations of *union*, *product* and *star*.

Given a partially commutative monoid $\mathbb{F}(\Sigma, \mathcal{I})$, the independence relation $\mathcal{I}$, or its complement θ, can be represented by an undirected graph $G = \langle V, E\rangle = \langle \Sigma, \mathcal{I}\rangle$, as show by the example in Figure 1. In our proposed algorithm we will use the notion of *clique covering* and *maximal clique* of a graph.

Definition 1 (Clique). *Let G be an undirected graph. A* clique *V_i of G is any complete subgraph $G_i = \langle V_i, E_i\rangle$.*

In other words, it is a subset of the set of nodes $V \supseteq V_i$ such that $(a,b) \in E_i, \forall a,b \in V_i$ with $a \neq b$. A clique V_i is also *maximal* (w.r.t. the inclusion relation), if G_i is the maximal complete subgraph of G, that is, if it does not have any subset that is a clique itself: $\forall V_j \subset V_i \Rightarrow i = j$.

Definition 2 (Maximal clique covering). *Let $G = \langle V, E\rangle$ be an undirected graph. 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 *and only the maximal cliques.*

If $\mathbb{F}(\Sigma, \mathcal{I})$ is represented by $G = \langle V, E\rangle$, then $\mathbb{M}_{\mathcal{I}}(\Sigma)$ is such that $\forall i,j \in [1, |\mathbb{M}_{\mathcal{I}}(\Sigma)|] \mid i \neq j \wedge V_i, V_j \in \mathbb{M}_{\mathcal{I}}(\Sigma) \Rightarrow V_i \not\subseteq V_j$. The cliques are actually the subsets of the alphabet $\Sigma_1, \Sigma_2, ..., \Sigma_k \subseteq \Sigma$, with $k \geq 1$. To summarize, $\mathbb{M}_{\mathcal{I}}(\Sigma)$ will denote the independence clique covering of $\mathbb{F}(\Sigma, \mathcal{I})$, while $\mathbb{M}_\theta(\Sigma)$ will indicate the dependence clique covering; if not explicitly stated, $\mathbb{M}(\cdot) = \mathbb{M}_\theta(\cdot)$ (maximal). For instance, according to the relation represented in Figure 1a, $\Sigma = \{a,b,c,d\}$ is covered by $\mathbb{M}_{\mathcal{I}}(\Sigma) = \{\{a,c\},\{a,b,d\}\}$, while θ (Figure 1b) leads to $\mathbb{M}_\theta(\Sigma) = \{\{a\},\{b,c\},\{c,d\}\}$.

Cliques covering of an alphabet is necessary to introduce one of the two formalisms used in the following to represent the traces, i.e., *trace projection*. The straightforward way to represent a trace is by choosing the representative element, within the equivalence class, according to the lexicographic order of strings: for instance, in the equivalence class

$\{abcbad, abcabd, bacabd, bacbad\}$ the lexicographic order is $abcbad < abcabd < bacabd < bacbad$, thus the trace is conveniently represented by $[abcbad]$. This form is called *lexicographic normal form*. A trace can be also represented by a morphism $\pi_{\Sigma'}(\cdot)$ called *projection*, defined as a translation mapping between strings, $\pi_{\Sigma'} : \Sigma^* \mapsto \Sigma'^*$. It eliminates from the input string all the symbols not belonging to the given Σ'. Formally, for each $a \in \Sigma$ and $u \in \Sigma^*$, the morphism is recursively defined as follows:

$$\pi_{\Sigma'}(u) = \begin{cases} \varepsilon & u = \varepsilon \\ \pi_{\Sigma'}(w)a & u = wa, a \in \Sigma' \\ \pi_{\Sigma'}(w) & u = wa, a \notin \Sigma' \end{cases}$$

Thus, $\pi_{\Sigma'}(u) = u' \in \Sigma'^*$. It is important to highlight that $\Sigma' \in \mathbb{M}(\Sigma)$, thus Σ' are cliques. For instance, let us consider $\mathbb{F}(\Sigma = \{a, b, c, d\}, \theta)$ (with θ as depicted in Figure 1b), $\mathbb{M}(\Sigma) = \{\Sigma_1 = \{a\}, \Sigma_2 = \{b, c\}, \Sigma_3 = \{c, d\}\}$ the trace $t = abcbad \in \mathbb{F}(\Sigma, \theta)$ can be represented by both its equivalence class $[t]_{\equiv_\theta} = \{abcbad, bacbad, abcabd, bacabd\}$, and its projections on cliques: $\pi_{\Sigma_1}(t) = aa, \pi_{\Sigma_2}(t) = bcbm, \pi_{\Sigma_3}(t) = cd$, which can possibly overlap on one or more symbols (c, in this case). Note that $\pi_{\Sigma'}(t)$ maintains the order of the symbols of t; that is, $\forall a_i, a_j \in \pi_{\Sigma'}(t) \wedge i < j \Rightarrow \exists i' < j' \mid a_{i'} = a_i \wedge a_{j'} = a_j \wedge a_{i'}, a_{j'} \in u$.

Finally we define the following function.

Definition 3 (Index function)**.** *Let Σ be an alphabet, $\mathbb{M}_{\mathcal{I}}(\Sigma)$ a clique covering of Σ w.r.t. the independence relation $\mathcal{I}$, and $\wp(\mathbb{M}_{\mathcal{I}}(\Sigma))$ its powerset.*

The index function $I : \Sigma \mapsto \wp(\mathbb{M}_{\mathcal{I}}(\Sigma))$ *is defined as:*

$$I(a) = \{\Sigma' \in \mathbb{M}_{\mathcal{I}}(\Sigma) \mid a \in \Sigma'\} \tag{3}$$

The index function is a convenience operator that indicates the set of all the cliques containing a certain symbol. For instance, $I(b) = \{\{a, b, d\}\}$. In the same way we define the index function for the dependence relation: $I_\theta : \Sigma \mapsto \wp(\mathbb{M}_{\mathcal{I}}(\Sigma))$.

3 The Membership Problem for Program Code Rescheduling

The *Membership Problem* (MP) is fundamental for any study of formal languages and in particular for traces, as we mentioned in Section 1. The MP is the problem of deciding whether or not a given string $u \in \Sigma^*$ is such that $u \in [u]$ with $[u] \in T$, a given trace language.

The common characteristic of the problems that can be modeled using traces —and reduced to the MP— is the fact that they can all be formalized as a set of binary, symmetric constraints among a finite set of symbols. Despite the simplicity of its formulation, the MP plays a key role in real applications where classical dynamic programming techniques are used. For instance, consider the problem of determining whether a certain sequence of program instructions is actually an acceptable schedule: in this case, each statement, except for control flow statements, is represented by a symbol of the alpha-

```
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 }
```

(a) Code

(b) DDG

$\theta = \{$ 1 — 2, 5 — 3, 8 — 6, 6 — 3, 8 — 9, 4 $\}$

$\Sigma = \{1, 2, 3, 4, 5, 6, 8, 9\}$

(c) Dependence relation

(d) Automaton

Figure 2: The trace representation of a snippet of code.

bet Σ while the *Data Dependency Graph* (DDG) is transformed into θ. The dependece relation is actually the symmetric and reflexive closure of the DDG.

A more precise analysis on the translation of the DDG into a dependence alphabet is reported in [Breveglieri et al., 2000]. The author also shows how to obtain the *Concrete Dependency Graph* (CDG) from a dependece alphabet. The CDGs differ from the DDGs since they are derived by unfolding loops and thus loop-carried dependencies are explicit; they are parametric w.r.t. the number of iteration, k, of each loop. For instance, considering the code snipped in Figure 2, we can unfold the **while** () loop two times ($k = 2$): it generates the string $5_{(1)}6_{(1)}8_{(1)}9_{(1)}5_{(2)}6_{(2)}8_{(2)}9_{(2)} \in \Delta^*_{k=2}$, where $\Delta_k = \{c_{(i)} \mid c \in \Sigma, i \in 1, ..., k\}$ is obtained by adding subscript indexes $1, ..., k$ to each symbol of Σ. This of course apply to non-loops instructions as well, since a non-iterated instruction is executed once, by definition. Given the above string, the CDG is created by directly linking all the dependent symbols $(c_{(i)}, c_{(j)}) \in \theta$ that have subsequent indexes ($i < j$). Finally, [Breveglieri et al., 2000] is interesting example on which the MP is applied is the study of the maximal parallelization of loops, that often are the most critical structure to be parallelized by optimizing compilers.

Figure 2 depicts a first example of how to derive the trace language representing a sample code. The data dependencies among code instructions are given by means of the DDG. In this this simplified example (i.e., no nested loops), *solid* — arcs of the DDG indicate direct data dependencies while *dotted* $\cdots$ arcs are the loop-carried dependencies [Bacon et al., 1994]. The resulting θ, show in Figure 2, is obtained by the reflexive symmetric closure of the DDG taking into account *both* dotted and solid arcs.

Also, it can be noticed that, if i — $j \in \theta$ means that *both* instruction i depends on j and vice-versa, while i (or j) does not necessarily depend on j (or i). For instance, in the DDG of Figure 2, 1 — $2 \in \theta$ but 1 does not depend on 2. Thus, with traces, *Read After Write* (RAW) dependencies are transformed into *Write After Write* (WAW) dependencies. For instance, let us consider the three code samples in Figure 3: (a) instruction 2, which reads a, depends on 1, which writes the same variable; the vice-versa is shown in (b). The first dependence is modeled by a direct arc $1 \to 2$ in the DDG while the second would be $2 \to 1$; the corresponding θ would carry less information, making case (a) indistinguishable from case (b). WAW dependencies are not an issue

```
1  a = 0
2  b = z + a
```
(a) RAW

```
1  b = z + a
2  a = 0
```
(b) WAR

```
1  a = 1
2  a = 0
```
(c) WAW

Figure 3: Examples of different types of data dependencies: (a) Read After Write (RAW), (b) Write After Read (WAR), and Write After Write (WAW).

since they remains WAW.

Note (Control Dependencies and Program Dependencies) Even though this topic is slightly out from the scope of this paper, we point out that the concept of *dependence* among code instructions is complex and by no means limited to the analysis of *data* dependencies through the DDG. In order to take into account *all* the types of dependence, the complete *Program Dependence Graph* (PDG) [Natour, 1988] should be considered. However, to the best of our knowledge, control dependencies or other kind of dependencies among code instructions have not been studied in correspondence of trace languages, yet. For convenience, we remind that the PDG is a super-graph of the DDG including both control and data dependencies.
Control dependence describes how each program statement affects the other statements, by taking into account dependencies carried by control structures. The *Control Dependence Graph* (CDG) is a representation of the control flow of a program, thus an edge between two statements i, j exists iff the control flow can directly reach i from j. For instance, given the code in Figure 2, some control dependence are $4 \rightarrow 9$, $5 \rightarrow 6$: indeed, the execution of instruction 9 is control dependent on the condition "`x % c`" (line 4), and instruction 6 depends on the condition "`x % 2`" (line 5). In the remainder of this paper, for the sake of simplicity, we will compute θ by taking into account DDG only.

The trace theory have been shown to be effective for representing reschedules of programs instructions, especially if exploited for the purpose of automatic code parallelization. The traces augment the expressive power of the string languages in such a way that dependency relations are taken into account. The framework of string languages captures the *structure* (i.e., the syntax) of computer programs while the traces capture the *dependencies* among instructions; from this point of view, we could state that the traces extend the string language framework by adding the ability of specifying simple *semantic* constraints (e.g., data dependencies).

As we mentioned, the automata are exploited to model the structure of each program (i.e., string of a language). In particular, automata of *local* type can be used [Savelli, 2007; Berstel and Pin, 1996]. As shown in Figure 2, in such machines each arc enters a state labeled with the same symbol of the arc itself.

Definition 4 (Local Language)**.** *Let $L \subset \Sigma^*$ be a language. L is a* local language *if there exist three subsets $P \subset \Sigma^*$, $S \subset \Sigma^*$, and $N \subset \Sigma^2$ such that:*

$$L \backslash \{q_0\} = (P\Sigma^* \cup \Sigma^* S) \backslash \Sigma^* N \Sigma^*.$$

This definition is due to [Berstel and Pin, 1996]; the authors detail that $P = \{a \in \Sigma \mid a\Sigma^* \cup L \neq \varnothing\}, S = \{a \in \Sigma \mid \Sigma^* a \cup L \neq \varnothing\}, F(L) = \{x \in \Sigma^2 \mid \Sigma^* x \Sigma^* \cup L \neq \varnothing\}, N = \Sigma^2 \backslash F(L)$ and also prove that local languages are recognized by local automata. A local automaton is such that $\forall \Sigma$, $|\{q.a \mid q \in \mathbb{Q}\}| > 0$, where $\mathbb{Q}$ is the set of the states of the automaton and q_0 is its initial state.

All the program statements are mapped to Σ with a bijective function, so $|\Sigma|$ equals the number of all possible program instructions. The execution of an instruction is thus represented by a transition entering a state that is labeled with the same symbol.

3.1 State of the Art

The contributions in this area are limited to a few key approaches, being the MP a relatively new topic. Notable examples are the studies published in [Savelli, 2007; Breveglieri et al., 2005; Avellone and Goldwurm, 1998; Bertoni et al., 1989]. Furthermore, some properties of trace languages with particular attention to the MP have been reported in [Rytter, 1984] and in the early work by M. Clerbout and M. Latteux currently included in the comprehensive [Diekert and Rozenberg, 1995].

3.1.1 Exploiting the trace prefixes

The first solution was given in [Bertoni et al., 1989]. The authors proposed an algorithm that exploits the notion of *prefixes* of a trace. Similarly to the strings, prefixes $\text{Pref}_l(t)$ of length l of a given trace t are defined as the set of words t_i such as $t = t_i \cdot v$ for some trace v. The approach requires the trace language $T = [L]$ to be *rational*, thus $L \in Reg(\Sigma)$; in turn, this implies that the automaton $A = \langle \mathbb{Q}, \Sigma, \delta, q_0, \mathbb{Q}_F \rangle$ that defines $L = L(A)$ is known.

First, the algorithm inductively computes the set of prefixes $\text{Pref}(t) = \{t_1, t_2, \ldots, t_i\}$ by exploiting the fact that $\forall t' \in \text{Pref}_l(t) \Rightarrow \exists t'' \in \text{Pref}_{l-1}(t) \mid t' = t'' \cdot [a], a \in \Sigma$. The MP (i.e., deciding whether there exists a string $v \in L$ that is $\theta-$equivalent to the given trace string $t = [u] \in T = [L]$) is reduced to checking for the existence of at least one acceptance state $q \in \mathbb{Q}_F \cap \mathbb{Q}_t$ that must be reachable by running the automaton on words θ-equivalent to u. Such a set of states $\mathbb{Q}_{t=[u]} \subseteq \mathbb{Q}$ is efficiently computed while constructing the set of prefixes: let $\mathbb{Q}_{t_j}$ be the set of states reachable by reading the trace prefixes $\text{Pref}_{|t|-1}(t) = \{t_1, \ldots, t_i\} = \{t_j \mid t = t_j \cdot [a_j], a_j \in \Sigma, j = 1, \ldots, i\}$. Thus, $\mathbb{Q}_t = \cup_{j=1}^{i} \{q \in \mathbb{Q} \mid q \in \delta(t', a_j), t' \in \mathbb{Q}_{t_j}\}$.

The time complexity of this algorithm is proven by the authors to be in $O(|t|^{\sigma})$, with $\sigma = \max_{\Sigma' \in \mathbb{M}_{\mathcal{I}}(\Sigma)} |\Sigma'|$; while the space complexity is in $O(|t|^{\sigma-1})$. The bottleneck of the algorithm is the procedure for obtaining the set of prefixes of a trace, which is efficient because the authors use a particularly efficient data graph-based structure. More precisely, the set of prefixes of t is stored as the set of nodes $V = \text{Pref}(t)$ and an edge is created for each pair of nodes t', t'' such that $t' = t'' \cdot [a]$.

On the same direction, the algorithm recently proposed in [Avellone and Goldwurm, 1998] does not require the language L to be regular; instead, L is assumed to be a context-free language. It is based on the pre-computation of trace prefixes. Surprisingly, the

performances of this algorithm are still comparable w.r.t. the aforementioned approach. In the worst case, the time complexity is still polynomial with the size of the biggest clique of the *independence* relation: it requires $O(|t|^{3\sigma})$ time and $O(|t|^{2\sigma})$ space. However, as underlined in [Savelli, 2007] such a time complexity is unacceptable for practical purposes. Indeed, an empirical analysis of the independence relation of some common programs consisting of some hundredths of instructions $|t| \propto 10^2$ has shown that σ may grow with the same order of magnitude, turning the complexity in an exponential function.

3.1.2 Earley-like recognition

The recent works [Breveglieri et al., 2005; Savelli, 2007] focused on both rational and local trace languages. The authors first present an alternative way to calculate trace prefixes and, based on it, design an algorithm for the MP which cost $O(|t|^{\sigma})$. The second contribution shows how the worst-case time complexity can be significantly reduced by restricting the problem to local automata and by further limiting the iterations to nested cycles. The algorithm we developed and the abstract machine specified in our paper is based on the first of the two contributions, which is analyzed in the reminder of this section.

[Savelli, 2007] presents an algorithm for solving the MP following the scheme of the Earley context-free grammar parser [Earley, 1970]. Beside requiring $T = [L]$ to be a rational trace language (and thus $L \in Reg(\Sigma)$) the algorithm assumes that the trace $t = [u]$ is represented by $R(t)$ that is the set of its projections on maximal cliques (see Section 2, Definition 1, 2). The representation $R(t)$ of the trace is computed by a linear scan of the input string t, which is translated from the lexicographic normal form into $R(t) = \{\pi_{\Sigma_i}(u) \mid \Sigma_i \in \mathbb{M}_\theta(\Sigma), u : t = [u]\}$.

Based on $R(t)$, the algorithm constructs an array of $|t| + 1 = n + 1$ elements, $E[0]$, $E[1]$, $\ldots$, $E[n + 1]$, following a procedure driven by the automaton A which defines $L = L(A)$. Such a procedure simultaneously consumes one symbol at time on each projection according to the current state of A. For instance, if the trace 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 advances on both the first and the third clique projections; it also keep tracks of such a transition by storing one marker per clique, indicating the current position on each projection, and the state reached by reading the last symbol on each projection.

More precisely, a cell $E[i]$ is instantiated for each computational step i. The cell $E[i], i = 1, \ldots, m$, where $m = |\mathbb{M}_\theta(\Sigma)|$, contains a sort of "working list", that is, a list of elements used to keep the information required to proceed to the next step, $i + 1$. An element $e_j \in E[i]$, is a data structure storing (1) $e_j.state \in \mathbb{Q}$ the current state on A, and (2) $e_j.cursors \in \{0, 1, \ldots, \sigma\}^m$, which holds the length of the prefix that has been read so far on each of the m projections.

For instance, if $e_j \in E[2]$, $e_j.cursors = \langle 1, 0, 1\rangle$ and $e_j.state = q_3$, then at the computational step 2 there exist a path on the automaton such that one symbol has been consumed on the first and the third projection, while no symbols are read on the second

one; and this leads to the state q_3. In other words, the markers c are cursors. To indicate such elements we will use the shorthand notation $e_j = q_h\langle 1, 0, \ldots, 1\rangle$ meaning that $e_j.state = q_h$ and $e_j.cursors = \langle 1, 0, \ldots, 1\rangle$.

To efficiently increment the cursors and to store such elements the approach relies on a zeroed matrix of dimension $|\Sigma| \times \underbrace{|t| \times \cdots \times |t|}_{\sigma}$; every time an element is added to a cell, a "one" is stored in the corresponding cell. In this way, checking for the existence of an element (i.e., checking whether the corresponding position on the matrix holds a "one" or a "zero") always takes constant time. On the other hand, the cost for initializing the matrix to all zeros is not fixed as it grows with $|\Sigma|$ and σ.

A complete sample run of the algorithm is presented in Section 4.3, Example 4.

Observation 1 From another point of view, this algorithm constructs a parallel machine consisting on m copies of A, one for each clique. The automata move simultaneously on the inputs, that are, the projections on cliques. The string is accepted if all the projections are read and all the automata make a joint move to a final state.

The total cost of the procedure, detailed and exemplified in [Savelli, 2007, Section 3.2], is polynomial with the size of the biggest clique of the dependence relation. More precisely, it costs $O(|t|^{\sigma})$ time in the worst case. In addition, we provide another example detailed in Section 4.3 in order to provide a side-by-side comparison between our approach and the original one.

Note that, this approach relies on maximal cliques of the *dependence* relation θ while [Bertoni et al., 1989; Avellone and Goldwurm, 1998] required the *independence* relation $\mathcal{I}$ to compute the trace prefixes.

The main disadvantage of this algorithm is due to the fact that it requires the trace t to be pre-processed into its representation $R(t)$. From one hand, this is actually a minor issue being it just a matter of a linear scan that would cost $O(|t|)$ and can be performed in advance. On the other hand, such an approach is still requiring two scans and thus does not permit the code to be processed as a stream. This may be unpractical for certain real cases, especially if the code is not available for pre-processing, or simply because multi-pass parsing algorithms are not suitable for the particular application requirements.

Our algorithm avoid this issue by computing the projections of the trace as the parsing goes on. Our approach shares some similarities with both [Savelli, 2007] and [Avellone and Goldwurm, 1998]. Furthermore, our proposal differs from the previous approach as we followed a threefold approach: (i) from a theoretical point of view, we designed and formally specified an abstract non-deterministic machine to decide the MP (Section 4); from a practical point of view we (ii) implemented such a machine into a deterministic algorithm (Section 5) and (iii) empirically tested its performance using an ad-hoc testbed and data generator that have been developed for this purpose.

3.1.3 Other works

Toward a slightly different line, [Breveglieri et al., 2000] presents an analysis of the maximal parallelization of loops that can be achieved under certain hypotheses and by regular approximation of a language defined to model code loops. Strictly speaking, such a topic is not necessarily related to the MP. However, it is worth including it the reviewed approaches since our focus is on solving the MP *but* under the realistic assumptions of computer programs. The work in [Breveglieri et al., 2000] was one of the first studies regarding the exploitation of the theory of traces for program transformation, with particular attention to loop parallelization, the most rewarding region of a procedure. This work has been already mentioned in this section since it provide some useful formalizations on the connection between the theoretical framework of traces and the practical aspects of programming. Finally, such a study is complementary to our paper; they both study the recognizability exploiting different properties of different normal forms to represent the traces. The former uses the Foata normal form while our work present a recognizer for trace language exploiting the characteristics of the projection form.

4 An Algorithm for the Membership Problem

Here we describe our decision algorithm to solve the MP by defining what we called the *Non-Deterministic Buffer Machine* (NDBM). Also, we provide some execution examples used to both clarify the behavior of the NDBM and to compare it with the original Earley-like recognizer.

Informally, the NBDM has one input tape a finite set of internal *First-In First-Out* (FIFO) buffers and a control device with a finite number of states. Each buffer is an infinite sequence of cells while the input tape is unbounded to the right. Note that there is no real need for more multiple tapes and buffers as it can be proved that multiple tapes and internal buffers can be mapped into one single tape with proper marking, with the same computational power; however, to specify the NDBM we prefer keep them separated for the sake of clearness. The NDBM consumes one symbol at time from the leftmost position of the input tape (*pop*), dispatches it onto an appropriate internal buffer (*enqueue*) and updates the current state of the control device, accordingly. The computation terminates when there are no symbols left, neither on the input buffer nor in any of the internal buffers. If there are symbols left on internal buffers, the machine continues the computation until a decision is reached.

Before going into the details of the specification of the NDBM, we define the fundamental building blocks of the recognizer along with proper examples to clarify the concepts.

4.1 Preliminary Definitions

First of all, we precisely define the concept of *buffer* used by the machine to perform both reading and writing operations.

Definition 5 (Buffer)**.** *Given an alphabet Σ_i, called the* buffer alphabet, *a* buffer b_i *is a FIFO queue containing only symbols in Σ_i:*

$$b_i := \boxed{\rangle\, a_I \cdots a_j \cdots a_O \,\rangle}$$

where "$\rangle a_I \cdots$" is the input side while "$\cdots a_O\rangle$" is the output side.

A buffer which contains no symbols is said to be empty *and is indicated as $b_i = \rangle\varepsilon\rangle = \varnothing$.*

Thus, each buffer b_i is associated to a subset $\Sigma_i \subset \Sigma$ of symbols. For instance, if $\Sigma_i = \{a, c\} \subset \Sigma = \{a, b, c, d, e\}$, $b_i = b_{\Sigma_i=\{a,c\}}$ holds symbols drawn from Σ_i only. Buffers are uniquely identified by their own alphabet, meaning that $\forall i \forall j : b_i = b_j \Rightarrow \Sigma_i = \Sigma_j$. For the aforementioned reason, we will also use the self-explanatory notation b_{Σ_i} to indicate b_i, the buffer with alphabet Σ_i.

Each buffer is then characterized by a set of allowed operations, which defines how the symbols are handled by the control device.

Definition 6 (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 $b_{\Sigma_i} = \varnothing$ 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 into b_{Σ_i}.*

Example 1 This example clarifies how each buffer function works.

- It is straightforward that if $b_{\Sigma_i=\{a,c\}} = \{a, b\}$ and $b_{\Sigma_j=\{d,e\}} = \{\}$, 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 $\mathrm{enqueue}(b_{\Sigma_i}, \varepsilon)$ would result in $b_{\Sigma_i} = \rangle\varepsilon aac\rangle = \rangle aac\rangle$.

Moreover, we specify the concept of *family of buffers* as follows.

Definition 7 (Family of buffers specific to a symbol)**.** *Let $b_{\Sigma_1}, \ldots, b_{\Sigma_k}$ be a set of buffers. A* family of buffers $\mathscr{B}_a$ specific to the symbol $a \in \Sigma$ *is defined as the union of all the buffer whose alphabet contains the symbol $a \in \Sigma$. That is:*

$$\mathscr{B}_a = \{b_{\Sigma_i} \mid a \in \Sigma_i\}. \tag{4}$$

For instance, consider the following set of alphabets $\mathbb{M}_\theta(\Sigma) = \{\{a\}, \{b, c\}, \{c, d\}\}$ and the corresponding set of buffers $\{b_{\Sigma_1}, b_{\Sigma_2}, b_{\Sigma_3}\}$. The family of buffers specific to the symbol c is $\mathscr{B}_c = \{b_{\Sigma_1}, b_{\Sigma_3}\}$, that is the set of the buffers belonging to the subset of the clique covering induced by the symbol c.

The definition of family of buffers can be further generalized, introducing the concept of *family of buffers specific to a set of alphabets.*

Definition 8 (Family of buffers specific to a set of alphabets). *Let $b_{\Sigma_1}, \ldots, b_{\Sigma_k}$ be a set of buffers and $\Gamma = \{\Sigma_i\}_N$ a set of N alphabets. A* family of buffers $\mathscr{B}_\Gamma$ specific to a set of alphabets Γ *is defined as the union of all the buffers b_{Σ_i} whose alphabet Σ_i is in Γ:*

$$\mathscr{B}_\Gamma = \{b_{\Sigma_i} \mid \Sigma_i \in \Gamma\}. \tag{5}$$

Hence, a family of buffer $\mathscr{B}'$ can be specific either to a symbol or to a set of alphabets. The above functions we defined for buffers can be generalized to *families* of buffers in the following manner.

Definition 9 (Family of buffer functions). *Let $\mathscr{B}'$ be a generic family of buffers. The functions* empty($\mathscr{B}'$), dequeue($\mathscr{B}'$), *and* enqueue($\mathscr{B}', a$) *are said to be* family of buffer functions.

- empty($\mathscr{B}'$) $\Longleftrightarrow \bigwedge_{b_{\Sigma_i} \in \mathscr{B}'} \mathrm{empty}(b_{\Sigma_i})$.
- dequeue($\mathscr{B}'$) *returns the set of symbols* $\Sigma' = \{a_i \mid \{a_i\} = \mathrm{dequeue}(b_{\Sigma_i}) \wedge b_{\Sigma_i} \in \mathscr{B}'\}$.
- enqueue($\mathscr{B}', a$) *is such that* enqueue(b_{Σ_i}, a) *is executed* $\forall b_{\Sigma_i} \in \mathscr{B}'$.

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

Definition 10 (Input Tape). *Let Σ be an alphabet. The* input tape T_I *is*

$$T_I := b_\Sigma$$

where b_Σ is the buffer associated to Σ. The operation enqueue(T_I, a) *is not defined for T_I while the other buffer operations are defined as in Definition 6.*

The input tape must indeed accept any symbol drawn from Σ. Moreover it can be assumed, with no loss of generality, that the symbols on the input tape are consumed in the same order as they are positioned; that is, the first symbol is consumed before the second, and so on. Then it can be deduced that the input tape is a special FIFO queue: once positioned in a given order, the symbols can be just consumed; further enqueue operations are not allowed. Hence, the input tape is a buffer associated to the alphabet Σ. Since the input tape is a special type of buffer, it can be indicated with both of the following, equivalent notations: $T_I = a_1 a_2 \cdots a_n = |a_n \cdots a_2 a_1\rangle$.

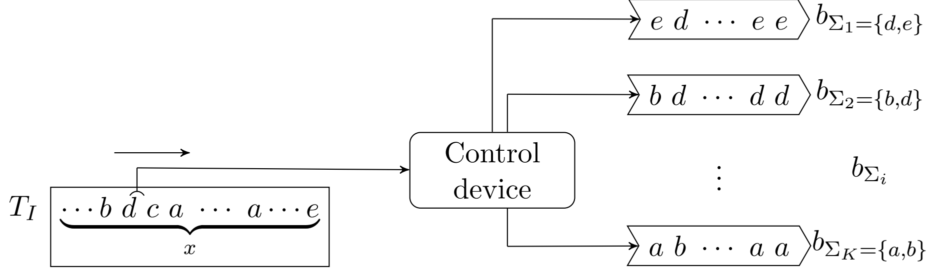


Figure 4: A pictorial representation of a NDBM.

4.2 The Non-Deterministic Buffer Machine

Formally, given a rational trace language $T = [L] \subseteq \mathbb{F}(\Sigma, \theta)$, the NDBM M to decide the membership problem for that specific language is defined as follows.

Definition 11 (Non-Deterministic Buffer Machine)**.** *A* Non-Deterministic Buffer Machine *(NDBM) M is a 4-ple $\langle A, \tau, \mathbb{B}, T_I \rangle$, where T_I is the input tape, and:*

- $\mathbb{B} = \mathscr{B}_{\mathbb{M}_\theta(\Sigma)} = \{b_1, b_2, \ldots, b_i, \ldots, b_K\} = \{b_{\Sigma_1}, b_{\Sigma_2}, \ldots, b_{\Sigma_i}, \ldots, b_{\Sigma_K}\}$ *is the family of buffers specific to the set of alphabets* $\mathbb{M}_\theta(\Sigma)$*.*
- $A = \langle \mathbb{Q}, \Sigma, \delta, q_0, \mathbb{Q}_F \rangle$ *is the deterministic recognizer of the string language* $L = L(A)$ *that defines the trace language* $T = [L(A)]$*. A is a finite state automaton with:*
  - $\mathbb{Q}$ *and* $\mathbb{Q}_F \subseteq \mathbb{Q}$ *are the finite set of states and acceptance states, respectively;*
  - q_0 *is the initial state;*
  - 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: is triggered if $a \neq \varepsilon$ (i.e., $\mathrm{empty}(T_I) = \perp$) and is such that*

    $$\tau(q, a, \mathbb{B}) = \langle q, \mathbb{B} \rangle$$

    and it is followed by a BufferWrite.
  - *BufferWrite: is such that*

    $$\begin{gathered} \mathbb{B}' : \mathrm{enqueue}(b_{\{\Sigma_i | a \in \Sigma_i\}}, a) \\ \tau(q, a, \mathbb{B}) = \langle q, \mathbb{B}' \rangle \end{gathered}$$

  - *BufferRead: is triggered if $\mathrm{dequeue}(b_{\{\Sigma_i | a \in \Sigma_i\}}) = \{a'\}$:*

    $$\tau(q, \varepsilon, \mathbb{B}) = \langle q' = \delta(q, a'), \mathrm{dequeue}(b_{\{\Sigma_i | a' \in \Sigma_i\}}) \rangle$$

    The choice of the buffer(s) b_{Σ_i} is assumed to be non-deterministic.

The choice among the three modes is non deterministic.

The initial configuration *of M is such that:*

- $\mathbb{B} = \{\varnothing, \varnothing, \dots, \varnothing\}$,
- *A on the initial state q_0, and*
- *the reading head of T_I pointing to the first symbol.*

Similarly, the acceptance configuration *is such that:*

- $\mathbb{B} = \{\varnothing, \varnothing, \dots, \varnothing\}$,
- *A on a final state $q \in \mathbb{Q}_F$, and*
- *the reading head of T_I pointing to the last symbol of the input.*

The Read transition is a normal read operation: it consumes one symbol from T_I; if this transition cannot be performed because $a = \varepsilon$ (i.e., there is no symbols left on T_I, then a BufferRead is performed. On the other hand, if a Read is performed, it *must* be followed by the corresponding BufferWrite which enqueues the current symbol a on all the buffers identified by an alphabet Σ_i such that $a \in \Sigma_i$ (in other words, the current symbol is pushed onto all the buffers belonging associated to the clique the symbol belongs to).

The BufferRead reads from one or more of the internal buffers; this transition consumes the symbol a from all the buffers having a on the output side. The following example illustrates how the transition functions works.

Example 2 (NDBM transitions) Let us assume the NDBM $M_1 = \langle A_1, \tau, \mathbb{B}, T_I \rangle$ where A_1 is depicted on Figure 5a; $T_I = |bdca\rangle$, $\mathbb{B} = \{b_{\Sigma_1}, b_{\Sigma_2}\}$:

- $b_{\Sigma_1} = b_{\{a,b,c\}} = \rangle abb\rangle$
- $b_{\Sigma_2} = b_{\{b,d,e\}} = \rangle dbbed\rangle$

In this configuration:

- A BufferRead cannot be performed from any of the buffers since $\nexists b_{\Sigma_i} \in \mathbb{B}$: $\text{dequeue}(b_{\Sigma_i}) = \{a'\} \wedge q' = \delta(q_0, a')$ 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:

$$\mathbb{B} = \{\rangle aabb\rangle, \rangle dbbed\rangle\}$$

On the other hand, if we assume that:

- $b_{\Sigma_1} = b_{\{a,b,c\}} = \rangle abcc\rangle$

- $b_{\Sigma_2} = b_{\{b,d,e\}} = \rangle dbed\rangle$

then a $\mathsf{BufferRead}$ can be performed on $b_{\Sigma_1} = \rangle abc\rangle c$ and the configuration of M_1 changes:

$$\mathbb{B} = \{\rangle abc\rangle, \rangle dbbed\rangle\}$$

and the current state of the machine becomes $q' = q_0$.

It must be noticed that the cardinality of $\mathbb{B}$ 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, by definition.

A first property of the NDBMs can be proven.

Property 1. *$\mathbb{B}$ is a family of buffers specific to the set Γ of alphabets, such that*

$$\bigcup_{\Sigma_i \in \Gamma} \Sigma_i = \Sigma.$$

Proof. It is straightforward to notice that the clique covering $\mathbb{M}_\theta(\Sigma)$ of the commutative alphabet $\mathbb{F}(\Sigma, \theta)$ 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 of NDBM, one buffer b_{Σ_i} exists for each $\Sigma_i \in \mathbb{M}_\theta(\Sigma) = \Gamma$. Thus, each buffer $b_{\Sigma_i} \in \mathbb{B}$ is such that $a \in \Sigma_i \Leftrightarrow a \in \Sigma$. 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. ■

Note that the above property does not assume the cliques to be non-overlapping (i.e., $\Sigma_i \cap \Sigma_j \neq \varnothing$ for some i, j) since the union eliminates the symbols contained in more than one alphabet (i.e., clique). In other words, $\{a, b\} \cup \{b, c, d\} \equiv \{a, b\} \cup \{c, d\}$.

The reader might have noticed the following observation.

Observation 2 An NDBM can be seen as a multi-tape *Turing Machine* (TM) but with a restricted set of operations allowed. However, there are two key differences between multi-tape TMs and NDBMs. In a NDBM:

1. there is no concept of non-terminal symbol; indeed, by definition, symbols are just buffered but no intermediate representations are used.

2. all the symbols contained in each buffer are *dependent* to each others. This is derived by the fact that each buffer is associated to an alphabet $\Sigma_i \subseteq \Sigma$ which is a clique of the dependence relation θ.

An extension to the Observation .2 is that, if no $\mathsf{BufferRead}$ operations were performed by a given NDBM M, then the buffers $b_{\Sigma_i}, \ldots, b_{\Sigma_K}$ of M would represent the input trace $t = |u|$ in its *projection* representation, $\pi_{\Sigma_1}(t), \ldots, \pi_{\Sigma_K}$. It is indeed true the following

Proposition 1 Let $M = \langle A, \tau, \mathbb{B}, T_I \rangle$ be a NDBM recognizing traces belonging to $T = [L]$ and $t = |u| \in T$. Let $M' = A, \tau, \mathbb{B}', T'_I$ be another machine $M' \equiv M$ that executes in parallel to M performing *all but* $\mathsf{BufferRead}$ operations (i.e., enqueue only buffers). At the end of the computation of M the buffers of M' are such that:

$$\forall b_{\Sigma_i} \in \mathbb{B}' \wedge b_{\Sigma_i} = a_1 a_2 \cdots a_{k_i} = w_i \Longrightarrow \exists \pi_{\Sigma_i}(t) = w'_i \mid w'_i = w_i^R$$

where $(\cdot)^R$ is the reverse string operator.

Proof (By construction). By definition, a trace t can be represented by its projections $\{\pi_{\Sigma_1}(t), \ldots, \pi_{\Sigma_{|\mathbb{M}_\theta(\Sigma)|}}(t)\}$ on the cliques $\mathbb{M}_\theta(\Sigma)$ of the dependence relation θ. All the strings $\pi_{\Sigma_i}(t) = w'_i$ contain all the symbols of t that belong to Σ_i, and such symbols are in the same order as they appear in t (see p. 5). Obviously, all the strings $s \in t = |u|$ have the same projections (see [Pighizzini, 1994, Section 1.3.3, Theorem 1.5]).

Symbols of u are processed from left to right and enqueued into buffers according to the alphabet associated to the buffers (see Definition 6). Performing no BufferRead simply means that buffers of M' are never emptied, thus each buffer b_{Σ_i} contains all the symbols $a' \in u$ such that $a' \in \Sigma_i$. Due to the FIFO policy, symbols are enqueued in the reverse order as they appear into the string u; that is, $\forall a_i, a_j \in b_{\Sigma_i} \wedge i < j \Rightarrow \exists i' > j' \mid a_{i'} = a_i \wedge a_{j'} = a_j \wedge a_{i'}, a_{j'} \in u$. This, in turn, means that the string represented by the buffer $b_{\Sigma_i} = a_1 a_2 \cdots a_{k_i} = w_i$ is the reverse of the projection $\pi_{\Sigma_i}(t)$. ■

Example 3 (Buffers vs. Projection on cliques) This example gives an idea of the intuitive proposition we proved above. Let us consider the trace $t = |u| = |a_1 b_1 b_2 c_1 c_2 d_1 c_3 d_2 d_3 e_1 e_2| \in T = [L]$, the machine M recognizing the traces of language $T \subseteq \mathbb{F}(\Sigma, \theta)$, and $\mathbb{M}_\theta(\Sigma) = \{\Sigma_1 = \{a, b\}, \Sigma_2 = \{b, c, d\}, \Sigma_3 = \{e\}\}$. Note that the subscript indexes are used to mark the number of occurrences of the same symbol in the string, thus a_2 means the second occurrence of a in the string.

The trace is represented by $\Pi(t) = \{\pi_{\Sigma_1}(t), \pi_{\Sigma_2}(t), \pi_{\Sigma_3}(t)\}$ and the buffers of M' are the following $b_{\Sigma_1}, b_{\Sigma_2}, b_{\Sigma_3}$:

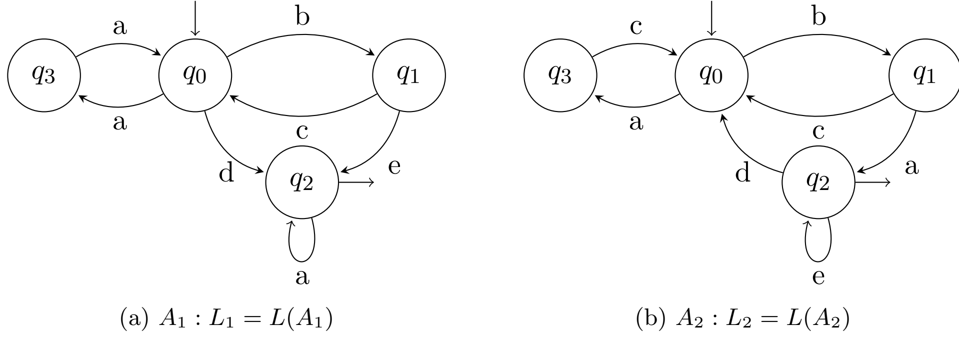
$$\begin{array}{l} \pi_{\Sigma_1}(t) = a_1 a_2 b_1 b_2 c_1 c_2 c_3 \\ \pi_{\Sigma_2}(t) = b_1 b_2 c_1 c_2 c_3 d_1 d_2 d_3 \\ \pi_{\Sigma_3}(t) = e_1 e_2 \end{array} \iff \begin{array}{l} b_{\Sigma_1} = c_3 c_2 c_1 b_2 b_1 a_2 a_1 \\ b_{\Sigma_2} = d_3 d_2 d_1 c_3 c_2 c_1 b_2 b_1 \\ b_{\Sigma_3} = e_2 e_1 \end{array}$$

For instance, the first (leftmost) symbol of $\pi_{\Sigma_1}(t)$ is a_1 that is indeed the first (rightmost) symbol enqueued in b_{Σ_1}.

4.3 Execution Examples

In order to give a more concrete and complete description of how our algorithm works, a few runs of a sample NDBM, M, are here provided. To better compare our approach vs. the original one that has inspired it, we first give an example of how the algorithm described in Section 3.1.2, in addition to the sample run illustrated in [Savelli, 2007].

In the following, the trace language $T_1 = [L_1] = \{t \in \mathbb{F}(\Sigma_1, \mathcal{I}_1) \mid \exists u \in L_1 : t = [u]\}$ is used; $L_1 = L(A_1)$ is the language recognized by the automaton A_1 in Figure 5a, $\Sigma = \{a, b, c, d, e\}$ and $\mathcal{I}_1 = \{a — c, a — b, a — d, b — d, a — e, b — e, c — e\}$ (reflexive arcs are omitted), thus $\theta_1 = \Sigma \times \Sigma \backslash \mathcal{I}_1 = \{b — c, c — d, d — e\}$. Slightly modified versions of L_1, T_1, θ_1 will be used in the following.



(a) $A_1 : L_1 = L(A_1)$ (b) $A_2 : L_2 = L(A_2)$

Figure 5: The automata A_1 (a) and A_2 (b) recognizing (i.e., defining) the languages L_1 and L_2, respectively, used in the examples.

Example 4 (Original Earley-like algorithm) Let us consider the string $u_1 = abcdbe$ and check whether or not it is such that $t_1 = [u_1] \in T_1$. We have that:

$$\mathbb{M}_{\theta_1}(\Sigma) = \{ \begin{array}{l} \Sigma_1 = \{a\}, \\ \Sigma_2 = \{b, c\}, \\ \Sigma_3 = \{c, d\} \\ \Sigma_4 = \{e\}\} \end{array} \qquad R(t_1) = \{ \begin{array}{l} \pi_{\Sigma_1}(u_1) = a, \\ \pi_{\Sigma_2}(u_1) = bcb, \\ \pi_{\Sigma_3}(u_1) = cd \\ \pi_{\Sigma_4}(u_1) = e\} \end{array}$$

At the beginning, the working list is empty: $E[0] = \{q_0\langle 0,0,0,0\rangle\}$; in this configuration there exist $\delta(q_0, a) = q_3$, $\delta(q_0, b) = q_1$ and $\delta(q_0, d) = q_2$, thus the first symbol on the first and the second projection can be read and consumed. This move lead to the generation of two elements in the next step cell: $E[1] = \{q_3\langle 1,0,0,0\rangle, q_1\langle 0,1,0,0\rangle\}$ since neither of them existed in $E[0]$. At step 2, each of the elements in $E[1]$ are expanded and $E[2]$ is populated accordingly: in this case, given $q_1\langle 0,1,0,0\rangle$, e can be consumed from the fourth projection leading to $q_2\langle 0,1,0,1\rangle$; the same is for c except for the fact that one single element is updated on two positions at once. Hence, $E[2] = \{q_3\langle 1,0,0,0\rangle, q_2\langle 0,1,0,1\rangle, q_0\langle 0,2,1,\rangle)\}$. At this point, b and d can be consumed giving $E[3] = \{q_3\langle 1,0,0,0\rangle, q_2\langle 0,1,0,1\rangle, q_1\langle 0,3,1,0\rangle, q_2\langle 0,2,2,0\rangle\}$, but the algorithm is blocked since no further symbol can be read on either the projection. Thus, the trace is not accepted. A valid string is $u_2 = aabcbe$ which leads to the following computation steps:

$E[0]$	$E[1]$	$E[2]$	$E[3]$	$E[4]$	$E[5]$	$E[6]$
$q_0\langle 0,0,0,0\rangle$	$q_3\langle 1,0,0,0\rangle$	$q_0\langle 2,0,0,0\rangle$	$q_1\langle 2,1,0,0\rangle$	$q_0\langle 2,2,1,0\rangle$	$q_1\langle 2,3,1,0\rangle$	$q_2\langle 2,3,1,1\rangle$
	$q_1\langle 0,1,0,0\rangle$	$q_0\langle 0,2,1,0\rangle$	$q_1\langle 0,3,1,0\rangle$	$q_2\langle 0,3,1,1\rangle$	$q_2\langle 1,3,1,1\rangle$	
		$q_2\langle 0,1,0,1\rangle$	$q_2\langle 1,1,0,1\rangle$	$q_2\langle 2,1,0,1\rangle$		

The trace string is recognized since $q_2 \in \mathbb{Q}_F$ and $\langle 2,3,1,1\rangle$ is such that all the symbols on each projection have been read.

The following example shows how the strings used in the previous case are parsed by a NDBM.

Example 5 (Parsing with a NDBM - Invalid trace) The initial configuration of the machine is $M_1 = \langle A_1, \tau, \{\varnothing, \varnothing, \varnothing, \varnothing\}, u_1, \varnothing\rangle$ and A_1 in its initial state. Here we show the run that performs the correct calculation: however, the NDBM guesses this (correct) run non-deterministically.

More precisely, the buffers are: $b_{\Sigma_1} = \rangle\varepsilon\rangle, b_{\Sigma_2} = \rangle\varepsilon\rangle, b_{\Sigma_3} = \rangle\varepsilon\rangle, b_{\Sigma_4} = \rangle\varepsilon\rangle, T_I = abcdbe$.

At the first step, a Read is performed and a is consumed from T_I, thus $\tau(q, a, \mathbb{B}) = \langle q, \mathbb{B}\rangle$; a BufferWrite is issued and a is enqueued in the family of buffers specific to a: enqueue$(\mathscr{B}_a, a)$ which consists in $b_{\Sigma_1} = a\rangle\varepsilon\rangle = \rangle a\rangle$. Since the current state on the automaton is q_0, the a that has been buffered can now be read as well with a BufferRead, thus: $\tau(q_0, \varepsilon, \mathbb{B}) = \langle q_3 = \delta(q_0, a), \text{dequeue}(\mathscr{B}_a)\rangle$. This last move equals to the creation of the first element $q_3\langle 1, 0, 0, 0\rangle \in E[1]$ (Example 4).

The second step performs exactly as the previous one: Read of b, BufferWrite of b into b_{Σ_2}, BufferRead by the same buffer, leading to empty$(\mathbb{B}) = \top$, $T_I = cdbe$. Similarly, when c is *Read* and written to $\mathscr{B}_c$, it can be then read from $\mathscr{B}_c$ since $\delta(q_1, c) = q_0$; d follows the same process but the subsequent b cannot be read so it would remain in $\mathscr{B}_b = \{b_{\Sigma_2} = \rangle b\rangle\}$. The e is buffered as well and it is subsequently dequeued, though the computation is blocked and the string is not accepted as expected. The set of buffers evolves as follows:

	0	1	2	3	4	5	6
$b_{\Sigma_1} =$	$\rangle\varepsilon\rangle$	$\rangle a\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$
$b_{\Sigma_2} =$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle b\rangle$	$\rangle c\rangle$	$\rangle\varepsilon\rangle$	$\rangle b\rangle$	$\rangle b\rangle$
$b_{\Sigma_3} =$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle c\rangle$	$\rangle d\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$
$b_{\Sigma_4} =$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle e\rangle$

Observation 3 As the reader may have noticed, the machine simulates the original algorithm and it simultaneously creates the projections just in time.

Example 6 (Parsing with a NDBM - Valid trace) Let us consider the string $u_2 = eeecacbbad$ and check the membership $t_2 = [u_2] \in T_2 = [L_2]$ (Figure 5b), where $T_2 \in \mathbb{F}(\Sigma, \theta_2)$ with $\theta_2 = \{a — b, c — d\}$, thus the cliques generate the following covering:

$$\mathbb{M}_{\theta_2}(\Sigma) = \{ \begin{array}{l} \Sigma_1 = \{a, b\}, \\ \Sigma_2 = \{c, d\} \\ \Sigma_3 = \{e\}\} \end{array}$$

The first three symbols eee cannot be Read thus they are buffered into $b_{\Sigma_3} = \rangle eee\rangle$; indeed, according to θ_2 the symbol e does not depend on either of the others. At this point, A_2 can read the substring $cacb$: this corresponds to four BufferWrites and four BufferReads to enqueue-and-dequeue each symbol to the corresponding buffer. After these moves, the machine is on state q_1 where the (second) b (in the string) cannot be read and thus it is enqueued into $b_{\Sigma_1} = \rangle b\rangle$ and it is not dequeued immediately; then a must enqueued as well into $b_{\Sigma_1} = \rangle ab\rangle$ as it depends on b. The automata ends this calculations in q_2 where two possible moves can be chosen; either

- perform a dequeue operation on $b_{\Sigma_3} = \rangle eee\rangle$, ending in q_2, or

- enqueue the last symbol (the d) into b_{Σ_2} and dequeue it, ending in q_0.

Both the choices bring to correct recognition in the same number of steps: in this example, we assume that the machine performs the first of the two and iterates it for three times since $empty(b_{\Sigma_3}) = \top$. At this point, the d is enqueued/dequeued into/from b_{Σ_2} and the last two symbols remaining in $b_{\Sigma_1} = \rangle ab \rangle$ can be dequeued as well, ending in $q_2 \in \mathbb{Q}_F$. The content of the buffers is summarized in the following:

	0	1	2	3	4	5	6	7	8	9	10	11	12
$b_{\Sigma_1} =$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle a\rangle$	$\rangle\varepsilon\rangle$	$\rangle b\rangle$	$\rangle ab\rangle$	$\rangle ab\rangle$	$\rangle ab\rangle$	$\rangle ab\rangle$	$\rangle a\rangle$
$b_{\Sigma_2} =$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle c\rangle$	$\rangle\varepsilon\rangle$	$\rangle c\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$	$\rangle d\rangle$	$\rangle\varepsilon\rangle$
$b_{\Sigma_3} =$	$\rangle\varepsilon\rangle$	$\rangle e\rangle$	$\rangle ee\rangle$	$\rangle eee\rangle$	$\rangle eee\rangle$	$\rangle eee\rangle$	$\rangle eee\rangle$	$\rangle eee\rangle$	$\rangle eee\rangle$	$\rangle ee\rangle$	$\rangle e\rangle$	$\rangle\varepsilon\rangle$	$\rangle\varepsilon\rangle$

Proposition 2 Let $T = [L] \subseteq \mathbb{F}(\Sigma, \theta)^*$ be a *rational trace language* defined by the regular string language $L = L(A) \in Reg(\Sigma)$, where A is a deterministic finite state machine (that defines, i.e., recognizes L) and θ is the dependence relation. Also, let M be a NDBM $M = \langle A, \tau, \mathbb{B}, T_I \rangle$.

$\forall t = |u| \in T, \forall s \in |u|$, the string s is accepted by M.

Proof (Reductio ad absurdum). Let us suppose that the hypotheses are satisfied and $\exists t' = |u'|$ s.t. $\exists s' \in t'$ s.t. s' is not accepted by M. Thus, it must be that, at the end of the execution of M:

1. the current state of A is $q \notin \mathbb{Q}_F$; or
2. $\exists b_{\Sigma'} \in \mathbb{B}$ s.t. $\text{empty}(b_{\Sigma'}) = \bot \wedge q \in \mathbb{Q}_F$, where q is the current state of A; or
3. the reading head of T_I is *not* pointing to the last input symbol.

Let us analyze the consequences of each case.

1. $\Rightarrow s'$ is not accepted by A, i.e. $s' \notin L$, but this means that $L \neq L(A)$. $\lightning$
2. $\Rightarrow s'$ is accepted by A since $q \in \mathbb{Q}_F$. Also, $\exists_{\geq 1} a' \in b_{\Sigma'}$ that was not dequeued from the buffer. In the case $\exists_{>1}$ the symbol a' prevents other symbols to be dequeued as well, otherwise 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 operation of the transition function τ (Definition 11). $\lightning$
   b) a' cannot be dequeued, but this must be that, either:
      i. $\nexists q' \mid \delta(q', a')$, but this implies that A is in $q \notin \mathbb{Q}_F$.
      ii. $a' \notin \Sigma'$, but this implies that Σ' is not the alphabet of $b_{\Sigma'}$, otherwise a' would not be in $b_{\Sigma'}$.
   c) (straightforward) the Read operation preceding the BufferWrite has read a wrong symbol, that both contradicts the definition of Read and implies that $a' \notin s'$. Then, i. and ii. contradict the definition of BufferRead operation of the transition function τ (Definition 11).

Then 2. leads to contradictions. ↯

3. (straightforward) ⇒ the machine would not be at the end of the computation. ↯

Finally, it also implies that a run with the correct sequence of operations does not exist, but this would contradict the assumption of non-deterministic recognition as of Definition 11. ■

4.4 Time and Space Complexity

The time complexity of the NDBM depends on the number of cliques of the dependece relation, that is $\mathbb{M}_\theta(\Sigma)$, and obviously on the length of the input string, that is $n = |u| = |t|$.

For each of the n symbols of u, after a Read, a BufferWrite is performed on all the buffers the symbol belongs to; such buffers are $|\mathbb{M}_\theta(\Sigma)|$. Within that cycle, all the buffers are scanned and a BufferRead is issued if possible, and this repeated $|\mathbb{M}_\theta(\Sigma)|$ times. In the worst case, the input strings is permuted in such a way that a complete buffering is required; after the buffering phase is completed, all the buffers must be scanned and each dequeued symbol must be further checked against all the buffers before actually removing it. Overall, the time complexity is bounded by $|\mathbb{M}_\theta(\Sigma)| \cdot |\mathbb{M}_\theta(\Sigma)| \cdot n$; considering that in the worst —but unrealistic and degenerative— case, the cliques are such that $|\Sigma| = |\mathbb{M}_\theta(\Sigma)|$, the time complexity of the non-deterministic recognizer is bounded by $n|\Sigma|^2$.

The space required by the machine grows with the number and the size of cliques. In the worst case there are $|\Sigma|$ cliques of size 1. However, as the number of cliques grows, the size of each clique decreases, and the space required by each buffer as well. In the average case, with $\frac{1}{2}|\Sigma|$ and equal probability for each symbol to occur in the string, the memory required by an NDBM is proportional to $n|\Sigma|$.

As pointed out in Section 6, a further and more deep analysis of both the time and space complexity is needed, also in order to refine this results and to calculate precise upper/lower bounds. As for the implementation of the NDBM through a deterministic algorithm (Section 5.1), it is straightforward to prove that Algorithm 1 simulates the original two-pass algorithm presented in [Savelli, 2007, Section 3.2] which has been proven to be $O(n^\sigma)$, where σ is the size of the larger clique.

5 Implementation and Results

This section describes the features of our *Quick Earley-Like Membership Evaluator* (QELME), a deterministic algorithm implementing the NDBM. Moreover, the results and the experimental setup are here investigated.

5.1 Implementing the Recognizer Through a Deterministic Algorithm

Before going into the details of the QELME architecture, a procedural description of the Algorithm 1 executed by the deterministic implementation of the NDBM 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 checked for membership. The procedure needs some variables to perform intermediate computations, namely:

- the variable *decision* holds the result, either $\top$ or $\bot$;
- E is a set that contains structured elements e in the same form described in Section 3.1.2 and used in the examples in Section 4.3. 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 5 and 11);
- $\underline{\underline{M}}$ is a matrix of dimension $|C| \times |\mathbb{Q}|$ initialized to $\bot$; as in [Savelli, 2007, Section 3.2], it is used to keep track of the actually existing elements 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 $M(e) \Leftrightarrow \underline{\underline{M}}(e.state, e.cursors)$;
- R is a vector of cursors, one per clique; each element of the vector is $R_j \in \{0, ..., |\pi_{\Sigma_j}(u)|\}$, it 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 NDBM is emulated by (1) one projection $\pi_{\Sigma_j}(u)$, (2) one pointer to the head R_j and (3) one pointer to the tail $e.cursor_j$.

Given the above variables, the buffer functions (see Definition 6 and 9) are emulated as follows:

- $\text{enqueue}(b_{\Sigma_j}, a) \Leftrightarrow \text{enqueue}(\pi_{\Sigma_j}(u), a) \wedge R_j \leftarrow R_j + 1$
- $\text{dequeue}(b_{\Sigma_j}) \Leftrightarrow e.cursors_j \leftarrow e.cursors_j + 1, \forall e \in E$
- $\text{empty}(b_{\Sigma_j}) \Leftrightarrow \forall e \in E \Rightarrow e.cursors_j = R_j$

Furthermore, Algorithm 1 relies on the shorthand procedure update($\cdot$) (Algorithm 2), which is performed only if the current character u_s is on the tail of all (i.e., $\forall \Sigma_k$) the buffers whose alphabet contains u_s itself. 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 $(\pi(u)_{\Sigma_k})_{e.cursors_k}$.

```
Input
A ← ⟨ℚ, Σ, δ, q_0, ℚ_F⟩                                  /* The automaton */
C ← Σ_1, ..., Σ_m                                        /* Clique covering */
u ← u_1 ··· u_n                                          /* The string */
Variables
decision ← ⊥
E ← {}                                                   /* The working list */
Π(u) ← {π_{Σ_1}(u), ..., π_{Σ_m}(u)} = {ε, ..., ε}       /* The projections */
M : C × ℚ ↦ {⊤, ⊥}                                       /* Initialized to ⊥ */
R ← ⟨0, ..., 0⟩ (m)                                      /* Right cursors */

while i ≤ 2n ∧ ¬decision do
    E' ← {}
    if i ≤ n then
        forall Σ' ∈ I(u_i) do
            enqueue(π_{Σ'}(u), u_i)                      /* Perform a BufferWrite */
        end
    end
    forall e ∈ E do
        if ∃j | e.cursors_j = R_j then E' ← E' ∪ e
        for j ← 0; j < m; j ← j + 1 do                   /* For all the cursors */
            if e.cursors_j < R_j then
                u_s ← (π_{Σ_j}(u))_{e.cursors_j}         /* Perform a BufferRead */
                if ∃δ(e.state, u_s) then                 /* Check readability */
                    if ∀Σ_k ∈ I(u_s) ⇒ (π_{Σ_k}(u))_{e.cursors_k} = u_s then
                        update(u_s, e, E', C, R, M, A)
                    end
                end
            end
        end
    end
    i ← i + 1                        /* Move the virtual cursor one step ahead */
    E ← E'                           /* Update the next working list */

end
```

Algorithm 1: Deterministic algorithm implementing the NDBM.

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) step is executed only if there are

```
Input
u_s                                                    /* Current symbol */
e'                                                     /* Current element */
E'                                                     /* The working list */
C                                                      /* Clique covering */
R                                                      /* Right cursors */
M                                                      /* Marking of E' */
A ← ⟨ℚ, Σ, δ, q_0, ℚ_F⟩                                /* The automaton */

procedure update(u_s, e', E', C, R, M, A)  begin
    e' ← e                                             /* Temporary copy */
    e'.state ← δ(e'.state, u_s)      /* Assign the new state to the element */
    forall Σ_k ∈ I(u_s) do                             /* For all the cliques */
        e.cursors_k ← e.cursors_k + 1                  /* Update the left cursor */
    end
    if M(e') = ⊥ ∧ ∃j | e'.cursors_j = R_j then        /* If not existing */
        E' ← E' ∪ {e'}                     /* Add it to the next working list */
        M(e') ← ⊤                                      /* Mark it as existing */
        decision ← e'.state = q ∧ q ∈ ℚ_F ∧ ∀j ⇒ e'.cursors_j = R_j
    end
end
```

Algorithm 2: The procedure update used by Algorithm 1.

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 main **forall** cycle iterates over the existing element into the current working list E; for all the left cursors in the current e, if there are symbols to read in the buffer (i.e., if $e.cursors_j < R_j$), then if the symbol on the top of the buffer (i.e., projection) is read and an update is performed (only if the aforementioned conditions are satisfied).

5.2 Prototype implementation

The abstract NDBM has been implemented into a highly configurable, parametric, and scalable testbed application written in Python 2.5. The high-level structure is of the main components depicted in Figure 6.

Sigma — This module implements Σ. The internal representation is a Python `List`. Beside a method to generate random strings of given length, a method to generate a random covering of the alphabet is provided and it is particularly important for conducting mass tests and evaluations. The covering is generated w.r.t. a given density, which is an aggregated indicator to characterize the clique covering.

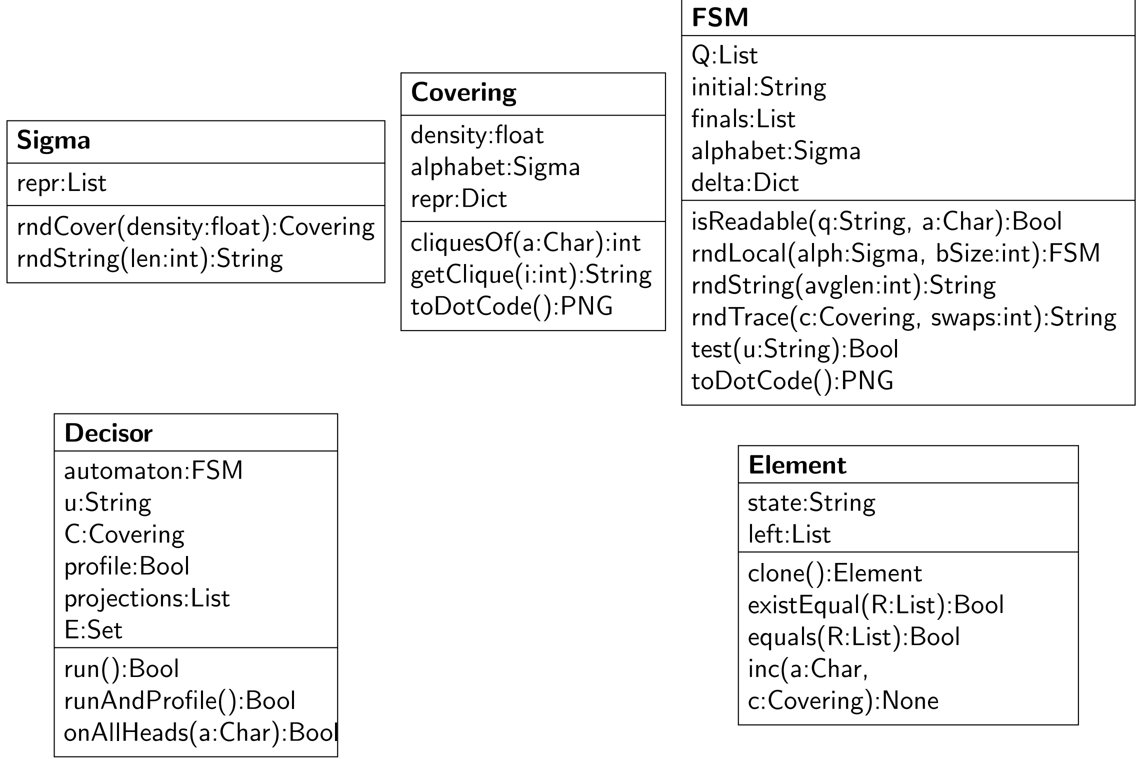


Figure 6: High-level structure of the main components of the Python implementation of QELME, the deterministic algorithm running the abstract NDBM.

Definition 12 (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*

$$d(\mathbb{M}(\Sigma)) = \frac{|\mathbb{M}(\Sigma)|}{|\Sigma|}. \tag{6}$$

The definition is given in general, regardless of the type of relation (e.g., dependence, independence). 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**. To be efficient, it has

been implemented with an indexed mapping, exploiting Python hash tables (i.e., the `Dict` data type). The index is such that accessing *all* the cliques the symbol a belongs to, always takes $O(1)$; this obviously requires slightly more space. 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 convenience method is also provided to generate a visual representation of the clique covering using Graphviz.

FSM — This module implements A. It has an instance of **Sigma** that 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 convenience 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; this approximation is due to the randomization strategy we chosen to traverse the automaton. All the transitions have the same probability to be chosen (uniform distribution over all the possible $\delta(q, a)$, for each q) ; at each state, one of the output arcs is picked up at random and the corresponding symbol a is appended to the final string. 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 permutes digrams swaps times. This allows somehow to tune a certain measure of "distance" between the random trace and the original string, representative of the whole class.
- 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; it represents, for each state, the average number of arcs pointing to preceding states. We call this indicator the "back size" as such arcs are backward directed. 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 randomly traversing the automaton.

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.

Example 7 (Random generation and recognition) Here we provide a quick overview of what can be done with QELME. In the following, the available options are shown. If one or more options are missing, the tool either randomizes such values (if it makes sense) or guesses them according to the other options. For instance, one may specify only the alphabet: the string are randomized as well as the automaton and the clique covering. This gives full control to the user to experiments with all the possible combinations of parameters that are important to the topic of the analysis.

```
QELME - Quick Earley-Like Membership Evaluator.
Ver. 0.6.1
(c)2007-2008 Federico Maggi, Stefano Crespi Reghizzi, Alessandra Savelli

Usage: qelme.py options

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -a AUTOMATON, --automaton=AUTOMATON
                        The automaton.
  -c COVERING, --covering=COVERING
                        The clique covering.
  -l RANDOM_STRING_LENGTH, --random-string-length=RANDOM_STRING_LENGTH
                        The length of the random generated string.
  -u STRING, --string=STRING
                        The string, comma separated!
  -p, --code-profiling  Turn on profiling. Off by default.
  -e EVALUATIONS, --evaluations=EVALUATIONS
                        Turn on performance evaluation (none or positive
                        integer). Example (non-default): 10 (ten simulations)
  -f MIN_SIGMA_SIZE, --min-sigma-size=MIN_SIGMA_SIZE
                        Random alphabet minimum size.
  -s MAX_SIGMA_SIZE, --max-sigma-size=MAX_SIGMA_SIZE
                        Random alphabet maximum size.
  -b BACK_SIZE, --back-size=BACK_SIZE
                        The number of back edges in the generated local
                        automaton: it does make sense in eval-mode only.
  -g, --save-automaton  Save the automaton in PNG. Off by default.
  -m STRING_RANDOMIZATION, --string-randomization=STRING_RANDOMIZATION
                        Test the algorithm on mutated string: choices are
                        'random', 'accepted' (default), 'mutated', 'trace'
  -z ALPHABET, --alphabet=ALPHABET
                        Specify a fixed alphabet to be used in the evaluation.
  -o OUTPUT_FILE, --output-file=OUTPUT_FILE
                        Output file (prefix) for saving simulation results.
                        Only in evaluation mode. Example (non-default): "foo"
```

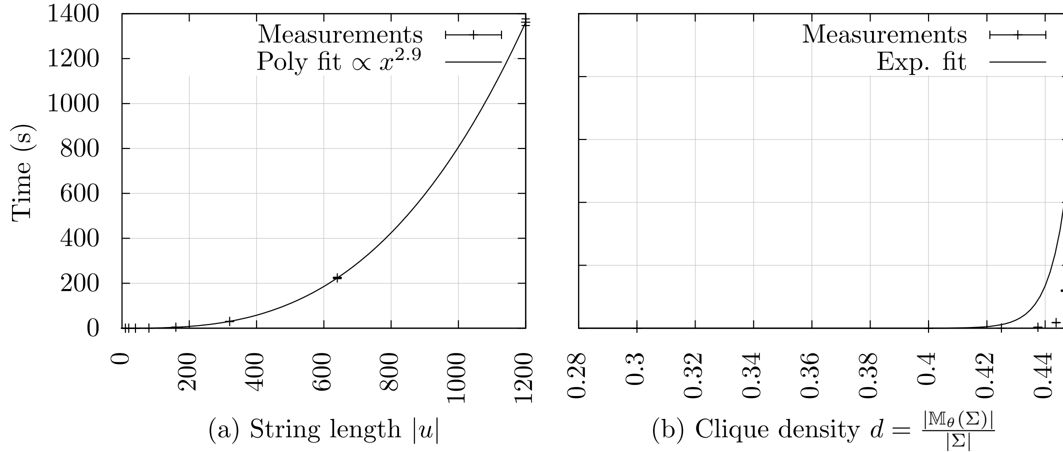


Figure 7: Time versus (a) input length and (b) clique covering density. We measured the execution time of 570 recognitions on valid (random) traces: each point corresponds to a fixed string length versus the average time calculated on 10 to 100 instances of the same experiment run on different input data, at each length.

```
-d COVERING_DENSITY, --covering-density=COVERING_DENSITY
                      In the case of random generated covering, a "density"
                      metric can be specified as the float |C|/|Alphabet|.
                      Example (non-default): 0.6
-v, --verbose         Be verbose
-V, --more-verbose    Be even more verbose
```

For instance, one may want to perform 1000 different experiments to evaluate the recognition of the trace strings (randomly) generated given a fixed alphabet, to see how different clique covering can influence the speed of the recognition. In this case, setting up such an experiment is as easy as typing:

```
qelme.py -e 1000 -p -z 'a,b,c,d,e,f,g,h,i,l,m,n,o' -b 5 -m 'trace' -v
```

The switch `-m` is particularly useful as it allows to generate random (1) accepted traces, (2) accepted strings, (3) random strings (high unacceptance rate), and (4) randomly mutated strings (likely to be unaccepted). This can be used to quickly set up targeted tests against specific cases. Results are exportable to *Comma Separated Values* (CSV) files for maximum portability and ease of post processing.

5.3 Experimental setup and results

We conducted several experiments and a reasonable amount of data has been collected. We mainly focused on the time required by the algorithm to recognize the input string. It is our care to recall that the measurements have been performed using a prototype implementation, which has been created with the goal of experimenting; in addition, the

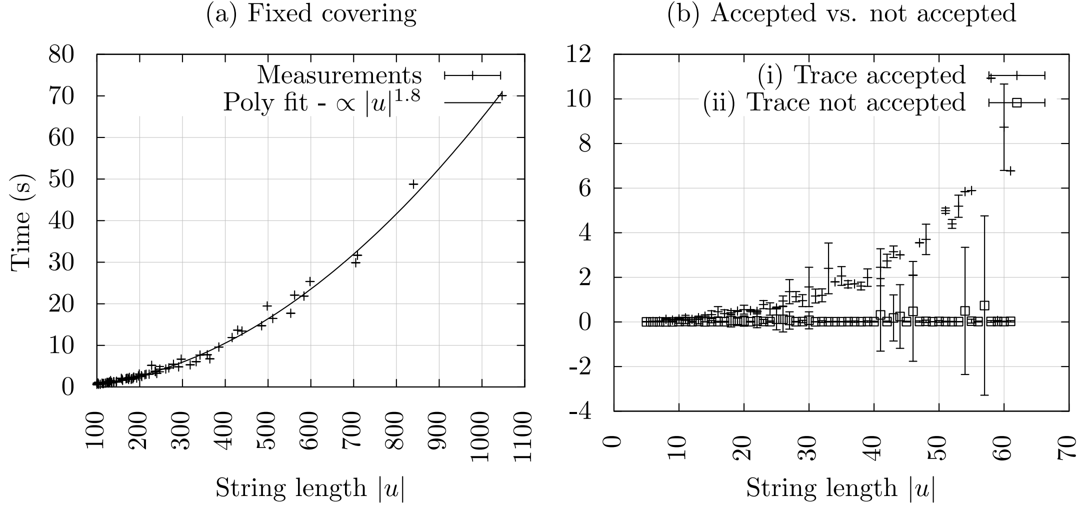


Figure 8: Time versus string length measured (a) with a fixed clique covering such that $|\Sigma| = 100$ and $\sigma = 7$ (larger clique size); (b) in case of (b-i) accepted and (b-ii) not accepted trace string. In (b) the execution time of 3000 recognitions on both valid and invalid (random) traces has been measured: each point corresponds to a fixed string length versus the average time calculated on several instances of the same experiment run on different input data, at each length.

scripts running the experiments have been executed on slow machines. Thus, the results must not be taken absolutely as they have been set up to find empirical hints about what the parameters influencing the performance of the recognition can be.

The first experiment investigates how the input length affects the time required for recognition. In this session, we used QELME to generate about 570 different *valid* traces, alphabets, clique covering and automata. For each of the eight different string lengths (i.e., 10, 20, 40, 80, 160, 320, 640, 1200), 10 to 100 different samples have been generated; this allowed us to measure the time required in the average case with high-precision with an error as low as 0.009–1.234%. The results are plotted in Figure 7(a). The interesting result of this experiment is that the recognition seems to be *independent* from the size σ of the largest clique in the covering of Σ; instead, according to the the analysis in [Savelli, 2007, Section 3.2] (reported in Section 3.1.2), the time was expected to be $t \propto |u|^{\sigma}$. On the other hand we found that the time in function of the input length can be perfectly fitted with $\sigma = 2.9$, even if we used a wide range of values of σ.

A more accurate analysis confirmed the above intuition. We fixed the clique covering (and thus $|\Sigma| = 100$) such that $\sigma = 7$. Figure 8 shows the execution time versus string length; the time is still polynomial w.r.t. the string length even if the two exponents slightly differs $(2.9 \pm 0.002 vs. 1.8 \pm 0.1314)$ due to the reduced amount of measurements. However, the important result is that we confirmed that $t \not\propto |u|^{\sigma}$, otherwise σ should be

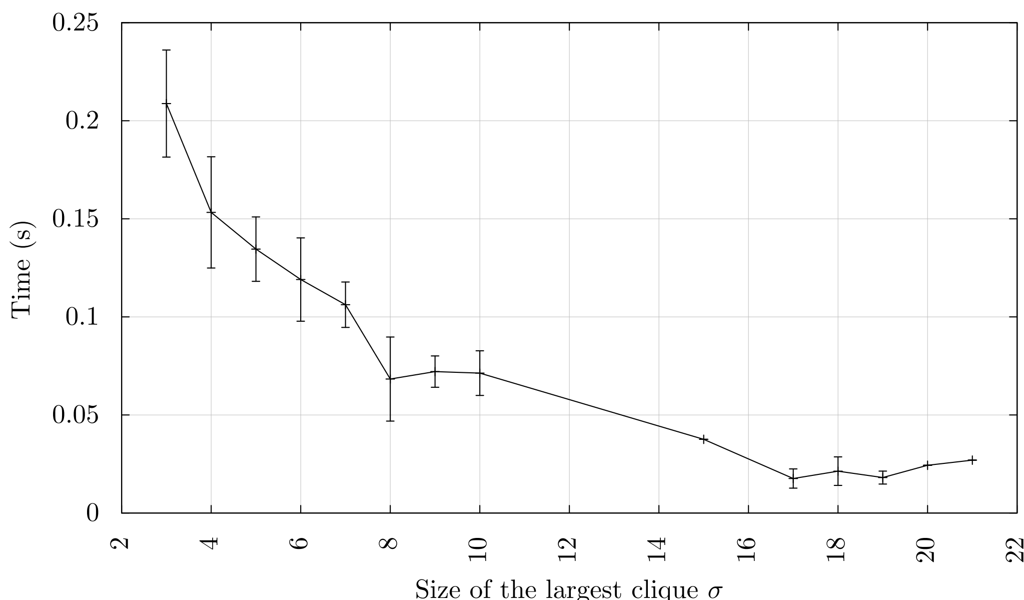


Figure 9: Time versus the size of the largest clique σ. The string length and the alphabet size are fixed (i.e., $|u| = 50, |\Sigma| = 50$) while the size of the larger clique increases from 3 up to $21 < |\Sigma|/2$.

$\sigma = 1.8$, which is obviously a contradiction since we used 7.

A further experiment has been conducted in order to see how the execution time varies if the string length and the alphabet size are fixed (i.e., $|u| = 50, |\Sigma| = 50$) while the size of the larger clique increases from 3 up to $21 < |\Sigma|/2$. Results are plotted on Figure 9: we avoided $\sigma > |\Sigma|/2$ in order to keep the covering significant. Unsurprisingly, the time required decreases as the size of the largest clique increases; this happens because, if the alphabet is fixed, it is less probable for each symbol to belong to more than one clique. Thus, for each symbol, a limited group of cliques (likely to be only the largest one) are inspected at each scan.

The second experiment investigates how the characteristics of the clique coverage affects the time required for recognition. We re-used the same data collected for the first experiment and calculated the average time required at different clique density values (see Definition 12). Results are plotted on Figure 7(b); if the string length varies, as the execution time increases exponentially with the covering density d (up to 0.45) and exhibits a quick grow for values of $d \in [0.42, 0.45]$. That was expected, even if the precise function between time and density was unknown by the theory.

The last experiment regards the comparison between valid and non-valid traces w.r.t. the time required to compute the result. Figure 8(b) shows how, in the average case and with a certain degree of unavoidable errors, the algorithm is able to stop the computation in advance if the string is not a valid trace. This is a detail of the first version of the

implementation and allows the stop of the execution when no valid elements are found in the working list E; that is, none of the elements contain a symbol that can be read according to the automaton.

6 Conclusions

In this paper we proposed a one-pass version of the two-pass, Earley-like recognizer for rational trace languages, previously described in [Savelli, 2007, Section 3.2]. More precisely, our contribution is threefold: we formally define the *Non Deterministic Buffer Machine* (NDBM), an abstract buffer machine to solve the membership problem for rational trace languages. Secondly, we describe the implementation of the NDBM through a deterministic algorithm; the details on how each component of the machine is simulated by the real algorithm are provided as well. Thirdly, we present the results obtained on real experiments conducted on arbitrarily long inputs and arbitrarily complex commutative alphabets. We also suggest a metric to quantify such a "complexity".

The first contribution is the result of an accurate analysis conducted on the existing algorithms proposed in the literature [Bertoni et al., 1989; Avellone and Goldwurm, 1998; Breveglieri et al., 2005; Savelli, 2007]. In Section 4.2 we define a buffer machine to solve the membership problem. The machine can be viewed as a multi-tape Turing machine which operates on buffers instead of on tapes; regardless of this similarity, we noticed two important differences between a normal Turing machine and a NDBM (Observation 4.2). Beside the description the main blocks, i.e., the buffers, we provide more useful definition to specify the behavior of (1) each buffer and (2) the families of buffers. The second concept is used to give a more compact description of the procedures invoked on groups of buffers. In addition, all the operations to access the symbols are given for both single buffers and families of buffers. We also provide examples of executions of the machine, in comparison to the original two-pass algorithm.

The non-determinism operates at two levels: first, at each computational step, the choice of the buffer(s) b_{Σ_i} is assumed to be non-deterministic (see Definition 11); second, the choice among the three possible modes (i.e., BufferRead, BufferWrite, *Read*) is assumed non-deterministic as well. Even though a more deep study about the non-deterministic recognition of trace languages is needed, at a first glance it seems that such a machine will not be able to recognize all the rational trace languages. This intuition arises by observing that the an implementation of the NDBM through a deterministic algorithm requires polynomial time, $n^{3\sigma}$ (where σ depends on the language); on the other hand, a quick analysis of the time complexity of the NDBM shows that it can recognize in $n|\Sigma|^2$. However, it is important to remark that a further and more deep analysis of both the time and space complexity is needed, also in order to refine this results and to calculate precise upper/lower bounds.

Another important point that require further studies is the implementation of a smart procedure capable of look-ahead on both the next symbol on the input string and the automaton transitions. Observing the moves of the NDBM it is quite evident that some sort of parallelism could be added. In particular, it seems that when an enqueue op-

eration is performed, a dequeue operation can be invoked on the same step, without affecting the soundness. However, this improvement would require to calculate, in advance, which buffers are enabled to dequeue symbols from, given the dependence relation and the currently enqueued symbol.

The second contribution consist in QELME, the testbed application we developed using the Python programming language. The purpose of QELME is to provide an easy-to-use (see Example 7) tool to aid the researcher to quickly set up a large amount of experiments with minimal effort. To the other end, QELME is a deterministic implementation of the NDBM and gave us deep understandings about the practical aspects and barriers that must cope with while trying to feel the gap between traces and real-world applications. Last but not least, it is our care to remark that QELME was not developed with the goal of realizing the *most efficient* implementation of a parsing algorithm for traces, in an absolute sense; instead, we preferred to focus on the ease of setting up experiments to investigate what are the factors (e.g., clique structure) that influence the algorithm.

The third contribution of this work is the experimental section we conducted to complement the theoretical part. We found that the size of the largest clique influences the computation time only if the alphabet size is fixed; on the other hand, according to our experiments the time is still a polynomial function of exponent between 1.8 and 2.9. It must be underlined that the string length is, as expected, more influencing than the characteristics of the clique covering, in terms of both the density (see Definition 12) and the size of the largest clique. To complete our experimental sessions, a more detailed profiling of the memory requirements of the algorithm have to be performed.

Acknowledgments

The author is thankful to Alessandra Savelli for the initial help and to Prof. Stefano Crespi Reghizzi for the constant help, suggestions and reviews.

References

Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java.* Cambridge University Press.

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

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

Berstel, J. and Pin, J.-E. (1996). Local languages and the Berry-Sethi algorithm. *Theor. Comput. Sci.*, 155(2):439–446.

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

Breveglieri, L., Crespi Reghizzi, S., and Garatti, M. (2000). Maximal parallel scheduling and trace theory. Technical report, Politecnico di Milano.

Breveglieri, L., Crespi Reghizzi, S., and Savelli, A. (2005). 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.

Cartier, P. and Foata, D. (1969). Problèmes combinatoires de commutation et réarrangements.

Diekert, V. and Rozenberg, G. (1995). *The Book of Traces*. World Scientific.

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

Keller, R. M. (1973a). Parallel Program Schemata and Maximal Parallelism I. Fundamental Results. *Journal of the ACM (JACM)*, 20(3):514–537.

Keller, R. M. (1973b). Parallel Program Schemata and Maximal Parallelism II: Construction of Closures. *Journal of the ACM (JACM)*, 20(4):696–710.

Lam, M. (1988). Software pipelining: an effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7):318–328.

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

Natour, I. A. (1988). On the control dependence in the program dependence graph. In *CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, pages 510–519, New York, NY, USA. ACM.

Pighizzini, G. (1994). *Linguaggi traccia riconoscibili ed automi asincroni*. PhD thesis, Università degli Studi di Milano, Università degli Studi di Torino.

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

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

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