# Recognizing Shuffled Languages

## Technical Report UMINF 11.01

Martin Berglund, Henrik Björklund, Johanna Högberg

Department of Computing Science
Umeå University, Sweden
{mbe,henrikb,johanna}@cs.umu.se

**Abstract.** Language models that use interleaving, or shuffle, operators have applications in various areas of computer science, including system verification, plan recognition, and natural language processing. We study the complexity of the membership problem for such models, i.e., how difficult it is to determine if a string belongs to a language or not. In particular, we investigate how interleaving can be introduced into models that capture the context-free languages.

## 1 Introduction

We study the membership problem for various language classes that make use of the shuffle operator $\odot$. When applied to a pair of strings $u$ and $v$, the operator returns the set of all possible interleavings of the symbols in $u$ and $v$. For example, the shuffle of $ab$ and $cd$ is $\{abcd, acbd, acdb, cabd, cadb, cdab\}$. The operator is lifted to languages by defining $\mathcal{L}_1 \odot \mathcal{L}_2$ to be the set $\bigcup\{u \odot v \mid u \in \mathcal{L}_1, v \in \mathcal{L}_2\}$. We also consider the shuffle closure operator, whose relationship to the shuffle operator resembles that of the Kleene star to concatenation.

Various aspects of shuffling have been studied in the theory of formal languages, see, e.g., [14, 17, 3, 12, 4, 10]. In this paper, we take the *shuffle languages* considered by Gischer [14] and by Jedrzejowicz and Szepietowski [17] as the starting point. These are the languages defined by regular expressions augmented with the shuffle and the shuffle closure operators.

Shuffling of languages is of interest in a number of different areas:

- In the *modelling* and *verification* of systems, shuffling is, as argued by Garg and Ragunath [11], useful for modelling the interleaving of processes. There is a close connection between shuffle languages and Petri nets [14, 11, 5].
- The shuffle operator (often called interleaving) is used in *XML database systems* for schema definitions, see, e.g., Gelade et al. [12].
- In *plan recognition*, the objective is to identify an agent's goal or plan, based on observations of the agent's actions [7, 23]. In a generalised version, a number independent agents that perform their actions in an interleaved fashion. To model this multi-agent scenario one could combine shuffle operators and context-free grammars [15]. For this approach to be of practical use, it is

necessary that the membership problem for the resulting languages remains efficiently solvable.

– In *natural language processing*, there is a growing interest in linguistic models for languages with relatively free word ordering. Recent work in this direction includes parse algorithms for so-called dependency grammars [22, 19].

Mateescu et al. [20] define a controlled form of shuffling where so-called trajectories of binary strings regulate the shuffling. This model is widely studied, mostly investigating decidability questions, see, e.g., [18, 8]. The case considered here corresponds to using the universal language $\{0,1\}^*$ as the trajectory. There has also been investigations into the sizes of the minimal automata for the shuffle of regular languages, see, e.g., [6, 2].

A number of fundamental questions regarding the complexity of the membership problem for various models remain unanswered. We answer some of them in this paper. In particular, we are interested in language classes that capture the context-free languages. Among the above application areas, such languages are primarily of interest in plan recognition and natural language processing.

It is important to distinguish the *uniform* and the *non-uniform* version of the membership problem. In the uniform version, both the string and a representation of the language is given as input. Thus it is important *how* the language is represented. In the non-uniform version, only the string to be tested is considered as input. The language is fixed, and thus its representation is not important.

**Contributions.** To facilitate the study of languages that combine restricted forms of recursion and interleaving, we define *Concurrent Finite State Automata* (CFSA). We show that emptiness for CFSA is solvable in polynomial time, investigate their closure properties, and identify the language classes that correspond to certain syntactic restrictions.

Our results for the complexity of the membership problems for various language classes are summarized in Table 1. For the full class of languages recognized by CFSA, we show that both the uniform and the non-uniform membership problem are NP-complete. For the *shuffle languages* (as used in [14, 17]), the *uniform membership problem* is NP-complete [1, 21], while the *non-uniform membership problem* can be decided in polynomial time [17]. We shed further light on the complexity of the membership problem by showing that the uniform version, parameterized by the number of shuffle operations, is hard for the complexity class W[1]. This indicates a strong dependence on the number of shufflings.

For the interleaving of a regular language and a context-free language, we show that the uniform (and thus also the non-uniform) membership problem can be solved in polynomial time. For the shuffling of a shuffle language and a context-free language, the uniform problem is of NP-hard, since this holds already for the shuffle languages. The non-uniform problem is, however, solvable in polynomial time. For the shuffling of two context-free languages, we show that already the non-uniform version of the membership problem is NP-hard.

It should be noted that we only investigate which broad complexity classes the problems belong to. In particular, for the problems that belong to P, our aim has not been to find optimal algorithms.

| | Sh | Reg $\odot$ CF | Sh $\odot$ CF | CF $\odot$ CF | CFSA |
|---|---|---|---|---|---|
| Non-Uniform | P | **P** | **P** | **NPC** | **NPC** |
| Uniform | NPC / **W[1]-hard** | **P** | NPC | **NPC** | **NPC** |

**Table 1.** Summary of results for the membership problem. The shuffle languages are abbreviated by Sh, the regular by Reg, and the context-free by CF. The results of this paper appear in bold face.

## 2 Preliminaries

**Sets and numbers.** If $S$ is a set, then $S^*$ is the set of all finite sequences of elements of $S$, and $\mathit{precl}(S)$ is the set of all finite prefix-closed subsets of $S^*$. The powerset of $S$ is denoted by $\mathit{pow}(S)$. We write $\mathbb{N}$ for the natural numbers, or $\mathbb{N}^+$ if we wish to exclude 0 from $\mathbb{N}$. For $k \in \mathbb{N}$, we write $[k]$ for $\{1, \ldots, k\}$. Note that $[0] = \emptyset$. The domain of a mapping $f$ is denoted $\mathit{dom}(f)$.

An *alphabet* is a finite nonempty set. We denote $\Sigma \cup \{\varepsilon\}$ by $\Sigma_\varepsilon$ and the set of all regular expressions over the alphabet $\Sigma$ by $\mathrm{Reg}(\Sigma)$. The length of a string $w \in \Sigma^*$ is written $|w|$, and for every $\alpha \in \Sigma$, $|w|_\alpha = |\{i \in [n] \mid \alpha_i = \alpha\}|$.

**Trees.** The set $T_\Sigma$ of *(unranked) trees* over the alphabet $\Sigma$ consists of all mappings $t \colon D \to \Sigma$, where $D \in \mathit{precl}(\mathbb{N})$. The *empty tree*, denoted $t_\varepsilon$, is the unique tree such that $\mathit{dom}(t) = \emptyset$. We henceforth refer to $\mathit{dom}(t)$ as the *nodes of* $t$ and write $\mathit{nodes}(t)$ rather than $\mathit{dom}(t)$. The size of a tree $t \in T_\Sigma$, denoted $\mathit{size}(t)$, is $|\mathit{nodes}(t)|$. The height of $t$, denoted $\mathit{height}(t)$, is $1 + \max(|v| \mid v \in \mathit{nodes}(t))$.

For a tree $t \in T_\Sigma$ and a node $v \in \mathit{nodes}(t)$, the *subtree of $t$ rooted at $v$* is denoted by $t/v$. It is defined by $\mathit{nodes}(t/v) = \{v' \in \mathbb{N}^* \mid vv' \in \mathit{nodes}(t)\}$ and, for all $v' \in \mathit{nodes}(t/v)$, $(t/v)(v') = t(vv')$. The *leaves* of $t$ is the set $\mathit{leaves}(t) = \{v \in \mathbb{N}^* \mid \nexists i \in \mathbb{N} \textit{ s.t. } vi \in \mathit{nodes}(t)\}$. The *substitution* of $t'$ into $t$ at node $v$ is denoted $t[\![v \leftarrow t']\!]$. It is defined by

$$\mathit{nodes}(t[\![v \leftarrow t']\!]) = (\mathit{nodes}(t) \setminus \{vu \mid u \in \mathbb{N}^*\}) \cup \{vu \mid u \in \mathit{nodes}(t')\} \ ;$$

and, for every $u \in \mathit{nodes}(t[\![v \leftarrow t']\!])$, if $u = vv'$ for some $v' \in \mathit{nodes}(t')$ then $t[\![v \leftarrow t']\!](u) = t'(v')$, otherwise $t[\![v \leftarrow t']\!](u) = t(u)$.

For a tree $t \in T_\Sigma$ let $v_1, \ldots, v_k \in \mathit{nodes}(t)$ be the immediate child nodes of the root ordered by numeric value. That is, $\{v_1, \ldots, v_k\} = \{v \in \mathit{nodes}(t) \mid |v| = 1\}$, ordered such that $v_i < v_{i+1}$ for all $i \in [k-1]$. Then we will write $t$ as $f[t_1, \ldots, t_k]$, where $f = t(\varepsilon)$ and $t_j = t/v_j$ for all $j \in [k]$. In the special case where $k = 0$ (i.e., when $\mathit{nodes}(t) = \{\varepsilon\}$), the brackets may be omitted, thus denoting $t$ as $f$.

**Shuffle operations and shuffle expressions.** We recall the definitions of the operations shuffle and shuffle closure, and of shuffle expressions, from [14, 17].

The *shuffle* operation $\odot : \Sigma^* \times \Sigma^* \to \mathit{pow}(\Sigma^*)$ is inductively defined as follows: for every $u \in \Sigma^*$ it is given by $\odot(u, \varepsilon) = \odot(\varepsilon, u) = \{u\}$, and by

$$\odot(\alpha_1 u_1, \alpha_2 u_2) = \{\alpha_1 w \mid w \in \odot(u_1, \alpha_2 u_2)\} \cup \{\alpha_2 w \mid w \in \odot(\alpha_1 u_1, u_2)\} \ ,$$

for every $\alpha_1, \alpha_2 \in \Sigma$, and $u_1, u_2 \in \Sigma^*$.

The operation $\odot$ extends to a mapping $\hat{\odot} : pow(\Sigma^*) \times pow(\Sigma^*) \to pow(\Sigma^*)$ with

$$\hat{\odot}(\mathcal{L}_1, \mathcal{L}_2) = \bigcup_{u_1 \in \mathcal{L}_1, u_2 \in \mathcal{L}_2} \odot (u_1, u_2) \enspace .$$

For readability, we use infix notation for $\odot$. From here on, we write the shuffle operation for languages as $\odot$ rather than $\hat{\odot}$.

The *shuffle closure* of a language $\mathcal{L} \in \Sigma^*$, denoted $\mathcal{L}^{\odot}$, is

$$\mathcal{L}^{\odot} = \bigcup_{i=0}^{\infty} \mathcal{L}^{\odot_i}, \text{ where } \mathcal{L}^{\odot_0} = \{\varepsilon\} \text{ and } \mathcal{L}^{\odot_i} = \mathcal{L} \odot \mathcal{L}^{\odot_{i-1}} \enspace .$$

*Shuffle expressions* are regular expressions that can additionally use the shuffle operators. Formally, the set of shuffle expressions over alphabet $\Sigma$ is formed as follows. Every $\alpha \in \Sigma$ is a shuffle expression, as well as $\varepsilon$ and $\emptyset$. If $s_1$ and $s_2$ are shuffle expressions, then so are $(s_1 \cdot s_2), (s_1 + s_2), (s_1 \odot s_2), s_1^*$, and $s_1{}^{\odot}$. Shuffle expressions that do not use the shuffle closure operator are called *closure free* shuffle expressions. The language $\mathcal{L}(s)$ of a shuffle expression $s$ is defined in the usual way. *Shuffle languages* are the languages defined by shuffle expressions.

## 3 Concurrent finite-state automata

In this section, we introduce *concurrent finite-state automata* (CFSA). The device is inspired by *recursive Markov models*, but differs from these in two aspects: the global state space is not partitioned into component automata, and, more importantly, recursive calls can be made in parallel. The latter feature allows for an unbounded number of invocations to be executed simultaneously, although each symbol in the input string can only be read by one invocation.

**Definition 1 (CFSA).** A *Concurrent FSA* is a tuple $M = (Q, \Sigma, \delta, I)$, where

- $Q$ is a finite set of *states*;
- $\Sigma$ is an alphabet of *input symbols*;
- $\delta \subseteq Q \times \Sigma_\varepsilon \times T$ is a set of *transitions*, where $T$ is the finite set

  $$\{q, q[p], q[p, p'], q[p^{\odot}] \mid q, p, p' \in Q\} \cup \{t_\varepsilon\} \enspace .$$

  Here, $p^{\odot}$ is to be read as single symbol. In the upcoming definition of CFSA semantics, transitions of the form $(q, \alpha, q'[p^{\odot}])$ will be interpreted as rule schema. A transition $(q, \alpha, t) \in \delta$ is
  - *terminal* if $|nodes(t)| = 0$,
  - *horizontal* if $|nodes(t)| = 1$, and
  - *vertical* if $|nodes(t)| > 1$.
- $I \subseteq Q$ is a set of *initial* states. □

**Remark.** For simplicity, we henceforth assume that the terminal transitions form a subset of $Q \times \{\varepsilon\} \times \{t_\varepsilon\}$. It is easy to see that every CFSA can be rewritten to this normal form in linear time.

We now establish the semantics of CFSA. Whereas a FSA is in a single state at a time, a concurrent FSA maintains a branching call-stack of states, represented as an unranked tree over an alphabet of states. In each step, exactly one leaf node of the state tree is rewritten. Vertical transitions model the invocation of child processes; horizontal transitions the continued execution within a process; and terminal transitions the completion of a process. A CFSA accepts a string if, upon reading the string, it can reach a configuration in which every processes has been completed, i.e., the state tree is empty.

**Definition 2 (Concurrent FSA semantics).** A *configuration* of the CFSA $M = (Q, \Sigma, \delta, I)$ is a tuple $(w, t) \in \Sigma^* \times T_Q$. The set of all configurations of $M$ is denoted $\Delta(M)$. A configuration $(w, t) \in \Delta(M)$ is *initial* (with respect to the string $w \in \Sigma^*$) if $t \in I$.

Let $(w, t), (w', t') \in \Delta(M)$. There is a *transition step* from $(w, t)$ to $(w', t')$, written $(w, t) \to (w', t')$, if there is a transition $(q, \alpha, s) \in \delta$ and node $v \in \mathit{nodes}(t)$ such that $w = \alpha w'$, $t/v = q$ (so $v$ is a leaf), and either

– $s \in T_Q$ and $t' = t[\![v \leftarrow s]\!]$, or
– $s = p'[p^{\odot}]$ and $t' = t[\![v \leftarrow p'[\underbrace{p, \ldots, p}_{n}]]\!]$ for some for $p, p' \in Q$ and $n \in \mathbb{N}$.

As usual, the reflexive and transitive closure of $\to$ is denoted $\xrightarrow{*}$. The *language recognised by $M$* is $\mathcal{L}(M) = \{w \in \Sigma^* \mid \exists q \in I : (w, q) \xrightarrow{*} (\varepsilon, t_\varepsilon)\}$. □

For the sake of brevity only the state-tree part of a configuration, called a *configuration tree*, may be shown in cases where the string is irrelevant.

It is sometimes useful to consider the execution of a CFSA $M = (Q, \Sigma, \delta, I)$ when starting in a particular state. For $q \in Q$, we denote by $M_q$ the CFSA $(Q, \Sigma, \delta, \{q\})$.

*Example 1.* Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the Dyck languages[1] over the symbol pairs $\lfloor, \rfloor$ and $\lceil, \rceil$, respectively. Their shuffle $\mathcal{L} = \mathcal{L}_1 \odot \mathcal{L}_2$ is recognised by the concurrent FSA $M = (\{q_0, q_1, q_1', q_2, q_2'\}, \{\lfloor, \rfloor, \lceil, \rceil\}, \delta, \{q_0\})$, where

$$\begin{aligned}\delta = \{ \; & (q_0, \varepsilon, q'[q_1, q_2]), \quad (q_0', \varepsilon, t_\varepsilon), \quad (q_1, \lfloor, q_1'[q_1]), \quad (q_1', \rfloor, q_1), \\ & (q_1, \varepsilon, t_\varepsilon), \quad (q_2, \lceil, q_2'[q_2]), \quad (q_2', \rceil, q_2), \quad (q_2, \varepsilon, t_\varepsilon) \quad \} \; .\end{aligned}$$

To illustrate the automaton's semantics, we step through an accepting run of $M$ on the string $w = \lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor$ (see Figure 1). Note that since $w \in w_1 \odot w_2$ for $w_1 = \lfloor\lfloor\rfloor\lfloor\rfloor\rfloor \in \mathcal{L}_1$ and $w_1 = \lceil\rceil \in \mathcal{L}_2$, it follows that $w \in \mathcal{L}_1 \odot \mathcal{L}_2$. □

It is known that $\mathcal{L}_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ is a context-free language, but it is not a shuffle language. Conversely, $\mathcal{L}_2 = \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$ is a shuffle language but is not context-free. Both $\mathcal{L}_1$ and $\mathcal{L}_2$ is recognized by a CFSA, and so is $\mathcal{L}_1 \cup \mathcal{L}_2$, which is neither a context-free nor a shuffle language. Thus the class of languages recognized by CFSA properly extends the union of the context-free languages and the shuffle languages. As we shall see, the CFSA have comparatively nice closure properties.

[1] A Dyck language consists of all well-balanced strings over a given set of parentheses.

In its initial configuration, $M$ is in the unique initial state $q_0$ and has yet to consume any input symbol.

$$(\ \lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor,\ \ q_0\ )$$

To proceed, $M$ must nondeterministically choose the transition $(q_0, \varepsilon, q'_0[q_1, q_2])$.

$$(\ \lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor,\ \ q'_0[q_1, q_2]\ )$$

By transition $(q_1, \lfloor, q'_1[q_1])$, $M$ reaches the configuration

$$(\ \lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor,\ \ q'_0[q'_1[q_1], q_2]\ )$$

and by $(q_1, \lfloor, q'_1[q_1])$ and $(q_2, \lceil, q'_2[q_2])$ the configuration

$$(\ \rfloor\lfloor\rceil\rfloor\rfloor,\ \ q'_0[q'_1[q'_1[q_1]], q'_2[q_2]]\ )$$

Now the automaton nondeterministically guesses that it is time to read the symbol $\rfloor$. It prepares by deleting the leaf labelled $q_1$ using transition $(q_1, \varepsilon, t_\varepsilon)$ to get

$$(\ \rfloor\lfloor\rceil\rfloor\rfloor,\ \ q'_0[q'_1[q'_1], q'_2[q_2]]\ )$$

and then $(q'_1, \rfloor, q_1)$ to get

$$(\ \lfloor\rceil\rfloor\rfloor,\ \ q'_0[q'_1[q_1], q'_2[q_2]]\ )$$

Again, $(q_1, \lfloor, q'_1[q_1])$ lets $M$ read $\lfloor$,

$$(\ \rceil\rfloor\rfloor,\ \ q'_0[q'_1[q'_1[q_1]], q'_2[q_2]]\ )$$

and $(q_2, \varepsilon, t_\varepsilon), (q'_2, \rceil, q_2)$ produces

$$(\ \rfloor\rfloor,\ \ q'_0[q'_1[q'_1[q_1]], q_2]\ )\ .$$

Thereafter, applying transition sequence $(q_1, \varepsilon, t_\varepsilon), (q'_1, \rfloor, q_1)$ twice yields

$$(\ \varepsilon,\ \ q'_0[q_1, q_2]\ )\ .$$

Although the entire input has been read, $M$ does not accept until the state tree has been reduced to the empty tree. This can be done by applying $(q_1, \varepsilon, t_\varepsilon), (q_2, \varepsilon, t_\varepsilon)$ to get

$$(\ \varepsilon,\ \ q'_0\ )\ ,$$

and finally $(q'_0, \varepsilon, t_\varepsilon)$ to reach

$$(\ \varepsilon,\ \ t_\varepsilon\ )\ .$$

**Fig. 1.** An accepting run of the CFSA $M$ on input $\lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor$.

**Theorem 1.** *The languages recognised by CFSA are closed under union, concatenation, Kleene star, shuffle and shuffle closure. They are not closed under intersection with a regular language or complementation.*

*Proof.* Let $M = (Q, \Sigma, \delta, I)$ and $M' = (Q', \Sigma, \delta', I')$ be CFSA. We assume without loss of generality that $Q \cap Q' = \emptyset$, and that the automata have only one initial state each, i.e., $I = \{q_0\}$ and $I' = \{q'_0\}$. The latter assumption can be made without loss of recognising power since $\varepsilon$-transitions are allowed.

**Union.** The classical construction of a nondeterministic automaton for the union of $M$ and $M'$ carries over from the FSA case: a new initial state $q$ is added, together with $\varepsilon$-transitions from $q$ to each of $q_0$ and $q'_0$.

**Concatenation.** For concatenation, we add the new states $q$, $q'$, and $q''$, where $q$ becomes the initial state of the new automaton. We also add the vertical transitions $(q, \varepsilon, q'[q_0])$ and $(q', \varepsilon, q''[q'_0])$, and the terminal transition $(q'', \varepsilon, t_\varepsilon)$. This allows the automaton to first simulate a run of $M$ and then a run of $M'$.

**Kleene closure.** Next, we construct a CFSA for the Kleene closure of $M$. All we have to do is add a new, unique, initial state $q$ to $Q$ along with the terminal transition $(q, \varepsilon, t_\varepsilon)$ and the vertical transition $(q, \varepsilon, q[q_0])$. This allows the automaton to simulate any number of runs of $M$, one after the other.

**Shuffle.** For the shuffle of $\mathcal{L}(M)$ and $\mathcal{L}(M')$ we add states $q, q'$, where $q$ becomes the unique initial state of the new automaton. We also add the vertical transition $(q, \varepsilon, q'[q_0, q'_0])$ and the terminal transition $(q', \varepsilon, t_\varepsilon)$.

**Shuffle closure.** To construct the shuffle closure of the language of $M$, we again add states $q, q'$, where $q$ becomes the unique initial state of the new automaton. Additionally, we add the vertical transition $(q, \varepsilon, q'[q_0{}^{\odot}])$ and the terminal transition $(q', \varepsilon, t_\varepsilon)$. This construction allows the new automaton to spawn any number of copies of $M$ that can then run in parallel over the input string.

**Intersection.** Consider the languages $\mathcal{L}_1 = (abc)^{\odot}$ and $\mathcal{L}_2 = a^*b^*c^*$. The former is a shuffle language, and the latter a regular language, so both are recognisable by CFSA. As we shall see, their intersection $\mathcal{L} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not. To obtain a contradiction, assume that $\mathcal{L}(M) = \mathcal{L}$ for some CFSA $M = (Q, \Sigma, \delta, I)$.

Before we proceed, let us introduce some convenient definitions. For every $q \in Q$, $M_q$ denotes the CFSA $(Q, \Sigma, \delta, \{q\})$. The *substrings* of a language $\mathcal{L}$, written *substring*$(\mathcal{L})$, is the set $\{v \mid uvw \in \mathcal{L} \text{ for some } u, w \in \Sigma^*\}$.

If a transition $r$ of the form $(q, \alpha, q[p, p']) \in \delta$ is applied in an accepting run of $M$, then $\mathcal{L}(M_p) \odot \mathcal{L}(M_{p'}) \subseteq$ *substrings*$(\mathcal{L})$. For this reason, $\mathcal{L}(M_p) \cup \mathcal{L}(M_{p'}) \subseteq \alpha^*$ for some $\alpha \in \{a, b, c\}$. Otherwise, if for example $w \in \mathcal{L}(M_p)$ and $w' \in \mathcal{L}(M_{p'})$ with $|w|_a > 0$ and $|w'|_b > 0$, the string $w'w \in w \odot w'$ would be in *substrings*$(\mathcal{L})$, but this is impossible since a $b$ occurs before an $a$ in $w'w$. It follows that the order of $p$ and $p'$ in $r$ is irrelevant. Hence, $r$ can equivalently be replaced by a pair of transitions such that $(\varepsilon, q) \xrightarrow{*} (\alpha, q[p[p']])$. The same argument justifies the replacement of transitions of the form $(q, \alpha, q[p^{\odot}])$ with transitions that yield $(\varepsilon, q) \xrightarrow{*} (\alpha, q[p[p[\ldots[p]]]])$.

After this language-preserving normalisation, the resulting CFSA only generates monadic configuration trees, which means that no shuffling is done. However, without shuffle operations, $\mathcal{L}(M)$ is a context-free language (cnf. Theorem 2), and it is well known that $\mathcal{L}$ is not a context-free language. Consequently, $\mathcal{L}$ is not recognisable by a CFSA.

**Complementation.** Since the CFSA languages are closed under union, but not under intersection, they are not closed under complementation either, since $(L_1 \cap L_2)$ can be expressed as $\overline{(\overline{L_1} \cup \overline{L_2})}$. □

**Restrictions and expressive power.** We introduce CFSA to provide an automaton model that can be syntactically restricted to capture the combination of shuffle operations with some well-known languages classes. The restrictions considered here are as follows. A CFSA $M = (Q, \Sigma, \delta, I)$ is

- *horizontal* if $\delta$ contains no vertical transitions;
- *non-branching* if every vertical transition is in $Q \times \Sigma \times \{q'[q] \mid q, q' \in Q\}$;
- *finitely branching* if no vertical transition is in $Q \times \Sigma \times \{q'[q^{\odot}] \mid q, q' \in Q\}$;
- *acyclic* if there is no configuration $(w, t) \in \Delta(M)$ and state $q \in Q$ such that $q$ appears twice on a path from the root to a leaf of $t$.

**Theorem 2.** *A language is:*

– *regular if and only if it is recognised by a horizontal CFSA;*
– *context-free if and only if it is recognised by a non-branching CFSA;*
– *a shuffle language if and only if it is recognised by an acyclic CFSA;*
– *a closure-free shuffle language if and only if it is recognised by an acyclic and finitely branching CFSA.*

*Proof (Sketch).* Horizontal CFSA are equivalent to nondeterministic finite automata in that they recognize the regular languages.

It is easy to turn a context-free grammar $G = (N, \Sigma, \gamma, S)$ on Chomsky normal form into a non-branching CFSA $M = (Q, \Sigma, \delta, I)$. Let $Q = N \cup \{\overline{q} \mid q \in N\}$, $I = \{S\}$, and define $\delta$ as follows.

– For every rule $q \to \alpha$ in $\gamma$, where $\alpha \in \Sigma_\varepsilon$, there is a horizontal transition $(q, \alpha, \overline{q})$ and a terminal transition $(\overline{q}, \varepsilon, t_\varepsilon)$ in $\delta$.
– For every rule $q \to pp'$ in $\gamma$, there is a transition $(q, \varepsilon, p'[p])$ in $\delta_2$.

For the opposite direction, it is equally easy to turn a non-branching CFSA into a language-equivalent push-down automaton.

Next, we show that acyclic CFSA correspond to the shuffle languages. The only-if direction follows directly from the proof of Theorem 1 since the constructions there preserve acyclicity.

Given a CFSA $M = (Q, \Sigma, \delta, I)$ we show how to construct a shuffle expression $s$ recognizing $\mathcal{L}(M)$. Two states $q, q' \in Q$ to be *connected* if there is a transition $(q, \alpha, t) \in \delta$, where the label of the root of $t$ is $q'$, for some $\alpha \in \Sigma_\varepsilon$. With this notion of connectivity, let $C_1, \ldots, C_k$ be the connected components of $M$. Consider the directed graph $G_M = (C_1, \ldots, C_k, E)$, where $(C_i, C_j) \in E$ if there is a state $q \in C_i$, a *vertical* transition $(q, \alpha, t) \in \delta$, and a state $p \in C_j$ such that $p$ (or $p^\odot$) labels a leaf of $t$. Since $M$ is acyclic, it follows that $G_M$ is acyclic.

Let $\delta_v \subseteq \delta$ be the set of all vertical transitions. We create an alphabet $\Sigma_v$ with one unique new symbol for each vertical transition. Let $h : \delta_v \to \Sigma_v$ be the bijection mapping each $d \in \delta_v$ to the corresponding alphabet symbol. Also, for each $d \in \delta_v$, let $q_d$ be a new state. Define $H$ to be the CFSA obtained from $M$ by replacing each vertical transition $d = (q, \alpha, q'[...])$ with the horizontal transitions $(q, \alpha, q_d)$ and $(q_d, h(d), q')$. Notice that the connected components of $H$ are the same as the connected components of $M$ and that $H$ is a finite automaton recognizing a regular language.

For each $q \in Q$, let the *regular* expression $r(q)$ be such that $\mathcal{L}(r(q)) = \mathcal{L}(H_q)$, that is, $r(q)$ describes the language that $H$ recognizes when starting from state $q$. Such a regular expression can be computed from $H$ using standard constructions.

We are now ready to describe how to construct the shuffle expression corresponding to $M$. To be precise, for each state $q \in Q$, we will define a shuffle expression $s(q)$ such that the language of $s(q)$ is the language of $M_q$, i.e., the CFSA obtained from $M$ by replacing $I$ by $\{q\}$. We do this by induction on the structure of $G_M$.

If $C$ is a leaf of $G_M$, then there are no vertical transitions in the connected component $C$. Hence, for every $q \in C$, we have $s(q) = r(q)$.

Suppose that $q$ belongs to a connected component $C_i$ such that for all states in all components reachable from $C_i$ in $G_M$, we have already computed the corresponding shuffle expressions. In this case we get the shuffle expression for $q$ by taking $r(q)$ and replacing symbols in $\Sigma_v$ by appropriate shuffle expressions. In particular, consider symbol $h(d) \in \Sigma_v$ that corresponds to $d = (q', \alpha, t) \in \delta_v$. The shuffle expression for $h(d)$ is obtained from $t$ as follows.

- If $t = p[p']$, for some $p, p' \in Q$, then the shuffle expression is $s(p')$.
- If $t = p[p'_1, p'_2]$ then the shuffle expression is $s(p'_1) \odot s(p'_2)$.
- If $t = p[p'^{\odot}]$, then the shuffle expression is $(s(p'))^{\odot}$.

The shuffle expression for $M$ is the union of those for the states in $I$, i.e.,

$$s = \bigcup_{q \in I} s(q).$$

The equivalence $\mathcal{L}(M) = \mathcal{L}(s)$ can be shown by a standard induction.

Finally, the fact that acyclic and finitely branching CFSA correspond to the closure free shuffle languages can bee seen from the constructions in the proof of Theorem 1. Only the shuffle closure operator induces unbounded branching. □

Since the closure free shuffle languages are regular [13], we can conclude that acyclic and finitely branching CFSA also recognize the regular languages.
The next theorem shows that CFSA do not provide us with the full power of linear bounded Turing machines.

**Theorem 3.** *The languages recognised by CFSA are properly contained in the context-sensitive languages.*

*Proof.* Let $M = (Q, \Sigma, \delta, I)$ be a CFSA and $w$ an input string. If there is an accepting run of $M$ on $w$ from some initial state $q_0$, then a nondeterministic Turing machine can guess and verify this run in linear space by proceeding as follows. (1) The TM simulates a run of $M$ on $w$ starting in $q_0$, but every time a vertical transition $(q, \alpha, q'[s])$ is used on the top level, where $s$ is a sequence of labels, the TM guesses what part of the subsequent string is to be consumed by the states trees derived from $s$, marks this segment off with brackets and a pointer to $s$, and continues in state $q'$ after the closing bracket until it has read all of $w$. If it accepts what it has seen so far, it goes on to verify each of the bracketed segments.

Let $w'$ be such a segment, annotated with $s$. If $s$ is a single state $p$, the TM recursively verifies that $w'$ is accepted by $M$ when starting from state $p$, i.e., $w' \in \mathcal{L}(M_p)$. If $s$ is a pair $p, p' \in Q$, the TM guesses a way to partition $w'$ into subsequences $u, u'$ so that $w' \in u \odot u'$. It then recursively verifies that $u \in \mathcal{L}(M_p)$ and $u' \in \mathcal{L}(M_{p'})$. Finally, if $s = p^{\odot}$, the TM guesses an $n \leq |w'|$ and a way to partition $w'$ into $n$ subsequences, and verifies recursively that each such subsequence belongs to $\mathcal{L}(M_p)$. This process continues recursively until no unprocessed bracketed segment has non-zero length. We note that the total amount of information that was recorded in the process is linear in $|w|$, so the non-uniform membership problem for CFSA languages can be decided by a linearly bounded nondeterministic TM. As shown in the proof of Theorem 1, no

CFSA recognizes the language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$, so it follows that the CFSA languages form a proper subset of the context-sensitive languages. □

Since not all CFSA-languages are context-free (e.g., there are non-context-free shuffle languages), we conclude that their expressive powers lies strictly between that of context-free grammars and that of context-sensitive grammars. Also unlike linear bounded TMs, CFSA can be efficiently checked for emptiness.

**Theorem 4.** *The emptiness problem for CFSA is decidable in polynomial time.*

*Proof.* Let $M = (Q, \Sigma, \delta, I)$ be a CFSA. A state $q$ of $M$ is *live* if $\mathcal{L}(M_q)$ is nonempty. Let $\mathcal{F} \subseteq Q$ be the smallest set satisfying the following conditions.

1. $F_0 = \{q \mid (q, \varepsilon, t_\varepsilon) \in \delta\}$
2. $F_i \subseteq F_{i+1}$
3. if $(q, \alpha, q') \in \delta$ and $q' \in F_i$ then $q \in F_{i+1}$
4. if $(q, \alpha, q'[s]) \in \delta$ for some $q' \in F_i$ and some $s$ such that every state that appears in $s$ belongs to $F_i$, then $q \in F_{i+1}$
5. $\mathcal{F} = \cup_{i=0}^{\infty} F_i$

**Claim.** A state $q$ of $M$ is live if and only if $q \in \mathcal{F}$.

For the if-direction, we prove by induction on the smallest $i$ such that $q \in F_i$ that $q$ is live. For $i = 0$ this is trivially true, since $(q, \varepsilon, t_\varepsilon) \in \delta$, and thus $M_q$ accepts the string $\varepsilon$.

Assume that every state in $F_i$ is live, and consider the state $q \in F_{i+1} \setminus F_i$. If $(q, \alpha, q') \in \delta$, with $q' \in F_i$, then there is a string $w$ such that $M_{q'}$ accepts $w$. This means that $M_q$ accepts $\alpha w$ and we conclude that $q$ is live. If there is no such rule, there must be a rule $(q, \alpha, q'[s])$ in $\delta$ such that $q'$ and either $s = p^{\odot}$ or every state that appears in $s$ belong to $F_i$. If this is the case, then there is a word $w_{q'}$ accepted by $M_{q'}$. If $s = p$, there is a word $w_p \in \mathcal{L}(M_p)$ and conclude that $M_q$ accepts $\alpha \cdot w_p \cdot w_{q'}$. Similarly, if $s = p, p$ there are strings $w_p \in \mathcal{L}(M_p)$, $w_{p'} \in \mathcal{L}(M_{p'})$, and $w_{p \odot p'} \in w_p \odot w_{p'}$ such that $M_q$ accepts $\alpha \cdot w_{p_1 \odot p_2} \cdot w_{q'}$. Finally, if $s = p^{\odot}$, we know that $M_q$ accepts $\alpha \cdot w_{q'}$. Thus $q$ is live.

For the other direction, assume that $q$ is live as witnessed by some word $w = \alpha_1 \cdots \alpha_m$ in $\mathcal{L}(M_q)$. Let

$$(w, q) = (w_1, t_1) \to \cdots \to (w_m, t_m) = (\varepsilon, t_\varepsilon)$$

be an accepting sequence of transition steps of $M_q$ on $w$. We show by induction that every state that appears in $t_1, \ldots, t_m$ is in $\mathcal{F}$. In particular, this means that $q$ belongs to $\mathcal{F}$, since $t = q$. Since $t_m = t_\varepsilon$, all states in $t_m$ belong to $\mathcal{F}$. Assume that all states appearing in $t_i$ belong to $\mathcal{F}$ and consider $t_{i-1}$. One of the following cases apply (for some leaf node $v$).

1. $t_{i-1} = t[\![v \leftarrow q]\!]$, $t_i = t[\![v \leftarrow q']\!]$, and there is a transition $(q, \alpha_i, q') \in \delta$. If this is the case, $q \in \mathcal{F}$ and thus all states of $t_{i-1}$ belong to $\mathcal{F}$.
2. $t_{i-1} = t[\![v \leftarrow q]\!]$, $t_i = t[\![v \leftarrow q'[u_1, \ldots, u_n]]\!]$, and there is a transition $(q, \alpha_i, q'[s]) \in \delta$ such that

  - $s = p$, $n = 1$, and $u_1 = p$,
  - $s = p_1, p_2$, $n = 2$, $u_1 = p_1$ and $u_2 = p_2$, or
  - $s = p^{\odot}$ and $u_1 = \cdots = u_n = p$.

   In either case, $q \in \mathcal{F}$ and thus all states of $t_{i-1}$ belong to $\mathcal{F}$.
3. $t_{i-1} = t[\![v \leftarrow q]\!]$, $t_i = t[\![v \leftarrow t_\varepsilon]\!]$. In this case, $q$ belongs to $F_0$ and we can conclude that all states appearing in $t_{i-1}$ belong to $\mathcal{F}$.

The set $\mathcal{F}$ can be computed in polynomial time and $\mathcal{L}(M)$ is empty if and only if $\mathcal{F} \cap I = \emptyset$. Thus emptiness for CFSA can be decided in polynomial time. □

## 4 Membership problems

### 4.1 The membership problem for unrestricted CFSA

The membership problem for unrestricted CFSA is intractable, both in the uniform and the non-uniform case.

**Theorem 5.** *Both the uniform and the non-uniform membership problem for CFSA is NP-complete.*

NP-hardness for the uniform membership problem for shuffle expressions is already known; see, e.g., [1, 21]. We postpone the hardness proof for the non-uniform case until Theorem 9 in Section 4.4, where it is proved for a subclass of CFSA. Lemma 1 states that the membership problem for CFSA is in NP.

**Lemma 1.** *Given a CFSA $M = (Q, \Sigma, \delta, I)$ and a string $w \in \Sigma^*$, it can be decided if $w \in \mathcal{L}(M)$ in nondeterministic polynomial time.*

*Proof (Sketch).* We show that there is a polynomial $P$ such that for every $w \in \mathcal{L}(M)$, there is a state $q_0 \in Q$ and a sequence of transition steps

$$(w, q_0) = (w_1, t_1) \to \cdots \to (w_n, t_n) = (\varepsilon, t_\varepsilon)$$

such that $n \leq P(|Q| + |w|)$. This result allows an accepting sequence of transition steps to be "guessed" as part of a nondeterministic polynomial-time decision algorithm for the membership problem.

As a starting point, assume that $M$ contains rules fulfilling the following.

1. For every $q \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, t_\varepsilon)$ then $(\varepsilon, q) \to (\varepsilon, t_\varepsilon)$.
2. For every choice of $q, q' \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, q')$ then $(\varepsilon, q) \to (\varepsilon, q')$.
3. For every choice of $q, q', p, p' \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, q'[p, p']) \xrightarrow{*} (\varepsilon, q'[p])$ then $(\varepsilon, q) \to (\varepsilon, q'[p])$.

This causes no loss of generality since there is a simple procedure to add these transitions to $M$ in polynomial time using the emptiness test (recall Theorem 4). For example, construct the automaton $M' = (Q, \Sigma, \delta', \{q\})$ where $\delta' \subseteq \delta$ contains only the transitions that do not generate any symbol. Then $(\varepsilon, q) \xrightarrow{*} (\varepsilon, t_\varepsilon)$ if and only if $M'$ is nonempty. Once Condition 1 is satisfied, the transitions needed to satisfy the remaining two conditions can be added through similar constructions.

For every pair of configurations $c = (\varepsilon, t)$, $c' = (\varepsilon, t') \in \Delta(M)$, if there is a sequence of transition steps from $c$ to $c'$, then there is also a sequence of length at most $n \leq |t| + 2|t'|$. Such a short sequence can be found by organising the transitions as follows: $(\varepsilon, t) \xrightarrow{*} (\varepsilon, \hat{t}) \xrightarrow{*} (\varepsilon, t')$ where the $t \rightarrow \hat{t}$ part of the derivation *only deletes nodes*, and the $\hat{t} \rightarrow t'$ part *never deletes nodes*. This is possible thanks to the transitions added above, since all possible node deletions/relabelings can be performed without generating extraneous nodes. In turn, this means that no node needs to be generated only to subsequently be deleted. It follows that at most $|nodes(t)|$ may need to be deleted, and at most $|nodes(t')|$ nodes may need to be created and/or relabeled with a new state.

Finally, in a sequence of transition steps that accepts the string $w$ and is of minimum length, no intermediary configuration tree needs to have more than $|w|$ leaves or be of height greater than $|Q|(|w| + 1)$. Only $|w|$ symbols are consumed by the transitions, so if there are $|w| + 1$ leaves, then one of them must be eventually consume $t_\varepsilon$. The existence of such a leaf violates the assumption that the sequence is of minimal length (notice that conditions 1–3 above ensure that useless nodes never have to be added). The height bound holds since a higher tree would have to have $|w| + 2$ or more copies of some state $q$ along some path. By a standard pumping argument the sequence could have chosen not to recognise any $q$-delimited section of the path (that is, loop once less on $q$). With $|w| + 2$ instances of $q$-labeled nodes, there are $|w| + 1$ such $q$-delimited sections on the path. Only $|w|$ symbols are consumed, so one of those sections will be matched up against the empty string. The redundant section could be omitted without affecting the accepted string, which violates the assumption that the original sequence of transition steps was of minimum length.

In conclusion, the size of the configuration trees necessary to accept a string $w$ is bounded by $|w|^2|Q|$, and any sequence of transitions on polynomially sized trees can be limited to a polynomial number of steps. There is thus, for every $w \in \mathcal{L}(M)$, a sequence of polynomial length, which means that a nondeterministic algorithm can check membership by just guessing the sequence. □

## 4.2 The membership problem for acyclic CFSA

We now turn to the membership problem for acyclic CFSA, i.e., the restriction of CFSA that recognises the shuffle languages.

**Corollary 1.** *For acyclic CFSA*

1. *the non-uniform membership problem is solvable in polynomial time, and*
2. *the uniform membership problem is NP-complete.*

*Proof.* The result for non-uniform membership follows directly from Theorem 2 and the fact, proved in [17], that non-uniform parsing for shuffle expressions is polynomial. For the uniform membership problem, membership in NP is obvious – just guess and verify a run of the automaton. NP-hardness follows by an easy adaptation of a result by Barton [1]. □

The uniform membership problem is NP-complete already for acyclic and finitely branching CFSA, which only recognise regular languages. The explanation is that for some languages, CFSA offer a more succinct form of representation than nondeterministic finite automata (and than the shuffle automata from [17]). One example is the language family $\{\{a^n\} \mid n \in \mathbb{N}\}$, for which the smallest NFAs (and shuffle automata) have sizes linear in $n$, while the smallest CFSAs are logarithmic in $n$.

Corollary 1 states that the problem is polynomial for a fixed automaton but NP-hard if the automaton is considered input. The question then remains whether the size of the automaton merely influences the coefficients of the polynomial or if it affects the degree itself. We give a partial answer by showing that when parameterized by the maximal size of a configuration tree for the automaton, the uniform membership problem for acyclic and finitely branching CFSAs is *not fixed-parameter tractable*, unless FPT = W[1]. This class equivalence is considered very unlikely and would have far-reaching complexity-theoretic implications. For more on parameterized complexity theory, see, e.g., [9].

We state the result for acyclic and finitely branching CFSA, but it could be equivalently stated for closure-free shuffle expressions. We first define the parameterized version of the problem.

**Definition 3.** An instance of the parameterized uniform membership problem for acyclic and finitely branching CFSA is a pair $(M, w)$ where $M$ is an acyclic and finitely branching CFSA over a finite alphabet $\Sigma$ and $w$ is a string in $\Sigma^*$. The parameter is the maximal size of any configuration tree for $M$. The question is whether $w \in \mathcal{L}(M)$. □

Notice that given an acyclic and finitely branching CFSA, we can easily compute the maximal size of its configuration trees. This number depends only on the automaton, not on the word it is currently reading.

If the problem were fixed-parameter tractable, there would be an algorithm for it with running time $f(k) \cdot n^c$, where $f$ is a computable function, $k$ is the parameter (the maximal size of a configuration tree), $n$ is the instance size, and $c$ is a constant. Theorem 6 gives strong evidence to the contrary.

**Theorem 6.** *The parameterized uniform membership problem for acyclic and finitely branching CFSA is W[1]-hard.*

The proof is by a fixed-parameter reduction from parameterized clique, which is known to be $W[1]$-complete [9].

**Definition 4.** An instance of $k$-Clique is a pair $(G, k)$, where $G = (V, E)$ is an undirected graph and $k$ is an integer. The question is whether there is a set $C \subseteq V$ of size $k$ such that the subgraph of $G$ induced by $C$ is complete. The parameter is $k$. □

*Proof.* The proof consists in a reduction from $k$-Clique to the membership problem at hand. Let $(G = (V, E), k)$ be an instance of $k$-Clique, and let $n = |V|$ and $m = |E|$. We construct an alphabet $\Sigma$, a shuffle expression $r$, and a string $w \in \Sigma^*$ such that $|\Sigma| = O(n+m)$, $|r| = O(k \cdot n^2 + k^2 \cdot m)$, $|w| = O(k \cdot n + m)$, the shuffle operator appears $O(k^2)$ times in $r$, and $w \in \mathcal{L}(r)$ if and only if $G$ has a clique of size $k$. To construct $\Sigma$, we assume that the vertices in $V$ are named $v_1, v_2, \ldots, v_n$ and that the edges are named $e_{i,j}$ where $i < j$ are the numbers of the two incident vertices and let $\Sigma = V \cup E$. The word $w$ is $v_1^k \cdot v_2^k \cdots v_n^k \cdot \mathit{edges}$, where *edges* is any enumeration of the edges in $E$. We define the regular languages $s, t, u$ by

- $s = (v_1^k + v_2^k + \cdots + v_n^k)^{n-k}$;
- $t = V^* \cdot E^*$;
- $u = \Sigma_{e_{i,j} \in E}(v_i \cdot v_j \cdot e_{i,j})$.

Finally, we define

$$r = s \odot t \odot (\bigodot_{i=1}^{k(k-1)/2} u).$$

The intuition behind the reduction is as follows:

- The expression $s$ matches $n-k$ sequences of $k$ copies of a vertex name. This leaves only $k$ such sequences in $w$ for the rest of $r$ to match against. In other word, the rest of the expression can only use $k$ distinct vertex names.
- Each instance of expression $u$ matches one sequence $v_i \cdot v_j \cdot e_{i,j}$. Thus, the $k(k-1)/2$ instances of $u$ match against $k(k-1)$ vertex names and $k(k-1)/2$ edge names. Due to the matching of $s$, the $k(k-1)$ vertex names can only be chosen from among $k$ vertices. Thus the $k(k-1)/2$ edge names, which are distinct since *edges* is an enumeration of $E$, represent edges that have both their endpoints in a set of vertices of size $k$.
- The expression $t$ matches any extra vertex and edge names that are left over.
- Any graph that has $k(k-1)/2$ distinct edges whose endpoints are all in a set of vertices of size $k$ has a clique of size $k$.

Thus $w$ belongs to $\mathcal{L}(r)$ if and only if $G$ has a clique of size $k$. Notice that $|r|$ is polynomial in $|G|$ and that the number of shuffle operators depends only on $k$.

Using Theorem 2 it is easy to find an acyclic and finitely branching CFSA $M_r$ such that $\mathcal{L}(M_r) = \mathcal{L}(r)$, the size of $M_r$ is polynomial in the size of $G$, and the maximum size of a configuration tree for $M_r$ is $O(k^2)$. Thus there is a fixed-parameter reduction from $k$-Clique to parameterized membership for acyclic and finitely branching CFSA, so the latter problem is W[1]-hard. □

The following corollary is immediate.

**Corollary 2.** *The uniform membership problem for closure-free shuffle expressions, parameterized by the number of shuffle operators, is W[1]-hard.*

## 4.3 The membership problem for Reg $\odot$ CF and Sh $\odot$ CF

We next show that the shuffle of a context-free language and a regular language is efficiently recognizable, even if the language descriptions are part of the input.

**Theorem 7.** *The uniform membership problem for the shuffle of two languages, one represented by context-free grammar and one represented by a nondeterministic finite automaton, is solvable in polynomial time.*

The above theorem actually has a shorter proof than the one given below, based on the fact that a shuffle language shuffled with a context-free language is a context-free language. We give the slightly longer proof because it is a good preparation for the proof of Theorem 8.

*Proof.* Let $G = (N, \Sigma, \delta, S)$ and $M = (Q, \Sigma, \gamma, I, F)$ be a context-free grammar on Chomsky normal form and an NFA, respectively.

To test membership in $\mathcal{L}(G) \odot \mathcal{L}(M)$, we extend the CYK algorithm for context-free grammars. For $A \in N \cup \{\varepsilon\}$ and $q_1, q_2 \in Q$ let $M_{q_1,q_2} = (Q, \Sigma, \gamma, \{q_1\}, \{q_2\})$ and $G_A = (N, \Sigma, \delta, A)$ unless $A = \varepsilon$, in which case $\mathcal{L}(G_A)$ contains only the empty string. Then $(A, q_1, q_2)$ is a *parse triple* for $G$ and $M$ over a string $w$ if and only if $w \in \mathcal{L}(G_A) \odot \mathcal{L}(M_{q_1,q_2})$. That is, $(A, q_1, q_2)$ is a parse triple for $w$ if $w$ can be partitioned into two subsequences, $w_1$ and $w_2$, such that the nonterminal $A$ can produce $w_1$ (or $w_1 = \varepsilon$ if $A = \varepsilon$) and $M$ can read $w_2$ by going from state $q_1$ to $q_2$. There are at most $(|N| + 1) \cdot |Q|^2$ distinct parse triples.

The idea, like in the CYK algorithm, is to compute the parse triples for each substring, starting with the substrings of length 1, and then combine triples to form new triples for successively longer strings. In the end, $w \in \mathcal{L}(G) \odot \mathcal{L}(M)$ if and only if there is a parse triple $(S, q_I, q_F)$ for the whole of $w$ such that $S$ is the start symbol of $G$, $q_I \in I$, and $q_F \in F$. Since $w$ has $O(m^2)$ substrings we will compute at most $O(m^2 \cdot |N| \cdot |Q|^2)$ parse triples.

For substrings of length one, computing the triples is trivial. Assume that we have computed all the parse triples for all substrings of length $k-1$. We show how to compute the parse triples for a substring of length $k$. Let $v = v_1 \cdots v_k$ be such a substring. To find out whether $(\varepsilon, q_1, q_2)$ is a parse triple for $v$, we proceed as follows. We check whether there is an $i \in [k-1]$ and a state $q$ such that $(\varepsilon, q_1, q)$ is a parse triple for $v_1 \cdots v_i$, and $(\varepsilon, q, q_2)$ is a parse triple for $v_{i+1} \cdots v_k$. If this is the case, $(\varepsilon, q_1, q_2)$ is a parse triple for $v$.

To determine whether $(A, q_1, q_2)$, $A \in N$, is a parse triple for $v$, we proceed in two steps. First, if there is a rule $A \to a$ in $\delta$, for some $a \in \Sigma$, we check whether there is an $i \in [k]$ and a $q \in Q$ such that $v_i = a$, $(\varepsilon, q_1, q)$ is a parse triple for $v_1 \cdots v_{i-1}$, and $(\varepsilon, q, q_2)$ is a parse triple for $v_{i+1} \cdots v_k$. If this is the case, $(A, q_1, q_2)$ is a parse triple for $v$. Second, we check, for each rule $A \to BB'$ whether there is an $i \in [k]$ and a $q \in Q$ such that $(B, q_1, q)$ is a parse triple for $v_1 \cdot v_i$ and $(B', q, q_2)$ is a parse triple for $v_{i+1} \cdot v_k$. In this case too, $(A, q_1, q_2)$ is a parse triple for $v$. $\square$

Since acyclic and finitely branching CFSA only contribute a more compact representation of the regular languages, Theorem 7 extends to non-uniform membership for the shuffle of a context-free language and a closure-free shuffle language:

**Corollary 3.** *The non-uniform membership problem for the shuffle of two languages, one represented by a context-free grammar and one represented by an acyclic and finitely branching CFSA, is solvable in polynomial time.*

Extending Theorem 7 with techniques inspired by [17], we get the following:

**Theorem 8.** *The non-uniform membership problem for the shuffle of a shuffle language and a context-free language is solvable in polynomial time.*

Since the languages are not part of the input, we may assume that they are represented by an acyclic CFSA $M$, and a context-free grammar $G$, respectively. We prove the above theorem in several steps. First, we show that we can assume that the CFSA for a shuffle language has certain structural properties. Second, we define *simple* configuration trees, and show that any computation of a CFSA in normal form for a shuffle language can be assumed to use only simple configuration trees. Third, we show an upper bound on the number of different simple configuration trees that need to be taken into account during a computation, and provide a compact representation for these. Finally, we prove the theorem along the lines of the proof of Theorem 7.

The first structural property of CFSAs that we consider is *stratification.* A CFSA is *stratified* if its state-space is layered in the following sense: for every state $q$, if $q$ appears in a configuration tree below a node $v$, then there is at most one state $p$ that can label $v$.

**Definition 5.** An acyclic CFSA $M = (Q, \Sigma, \delta, I)$ is *stratified* if, for every $q \in Q$, there is at most one $p \in Q$ such that, in a configuration tree, a node with label $p$ can be the parent of a node with label $q$. □

**Observation 1.** Let $s$ be a shuffle expression, and let $M_s = (Q, \Sigma, \delta, q_0)$ be the CFSA constructed from $s$ as in the proof of Theorem 1. Then $M_s$ has the following properties.

- It is *stratified.*
- It is *acyclic.*
- For each $q \in Q$, there is at most one vertical transition $(p, \alpha, t)$ in $\delta$ with $q$ labelling the root of $t$. We say that $M_s$ is *vertically separated.* We write $scp(M_s)$ (for shuffle-closure-parent) for the set of states that can have an unbounded number of children, i.e., $scp(M_s) = \{q \mid \exists p, p', \alpha : (p', \alpha, q[p^{\odot}]) \in \delta\}$. □

Having covered the first step our proof outline, we continue to introduce and reason about so-called simple configuration trees. For this purpose, we need the notion of pruned configuration trees and symmetrically equivalent nodes.

**Definition 6 (Symmetrical equivalence).** Let $M_s$ be a CFSA obtained from a shuffle expression, let $t$ be a configuration tree of $M_s$, and let $v, v'$ be a pair of nodes of $t$. The *pruning* of $t$ with respect to $v, v'$, written $prune(t, v, v')$, is the tree obtained from $t$ as follows. Let $P$ be the set of nodes $u$ of $t$ such that

1. $t(u) \in scp(M_s)$,
2. $u$ is a descendant of $v$ or $v'$, and
3. there is no node with a label in $scp(M_s)$ on the path from $v$ (or $v'$) to $u$.

In other words, $P$ is the set of closest shuffle-closure-parent descendants of $v$ and $v'$. Now, the partial configuration tree $prune(t, v, v')$ is derived from $t$ by removing all subtrees rooted at children of nodes in $P$.

A pair of nodes $v, v'$ in $t$ are *symmetrically equivalent* if there is an automorphism $f$ on the nodes of $t' = prune(t, v, v')$ such that

– $f(v) = v'$ and $f(v') = v$,
– for every $u \in \mathit{nodes}(t')$, $t'(f(u)) = t'(u)$, and
– for every $u, u' \in \mathit{nodes}(t')$, $f(u)$ is a child of $f(u')$ if and only if $u$ is a child of $u'$. □

It is easy to check that symmetrical equivalence is an equivalence relation in the algebraic sense, and thus reflexive, symmetric and transitive. When considering a configuration tree from a computational point of view, we cannot distinguish between symmetrically equivalent nodes. For our purposes this is an advantage, because it means that we do not have to remember where a configuration subtree attaches among symmetrically equivalent nodes.

**Observation 2.** Let $k \in \mathbb{N}$, let $t$ and $s_1, \ldots, s_k$ be configuration trees, and let $v_1, \ldots, v_k$ be symmetrically equivalent nodes in $\mathit{nodes}(t)$. All configuration trees in the set

$$\{t[\![v_1 \leftarrow s_{\phi(1)}, \ldots, v_k \leftarrow s_{\phi(k)}]\!] \mid \phi \text{ is a permutation on } [k]\}$$

are isomorphic. □

It is never necessary, for the sake of accepting an input string, to add children to a descendant $u$ of a node $v$ using shuffle-closure, if there is a node $v'$ that is symmetrically equivalent to $v$, and which has a descendant $u'$ that has already been given children using shuffle-closure. This claim, which will be proved later on, means that the search space can be reduced to *simple* configuration trees.

**Definition 7.** A configuration tree $t$ is *simple* if is it does not contain symmetrically equivalent nodes $v$ and $v'$, such that both $v$ and $v'$ have descendants which are labeled by states in $\mathit{scp}(M_s)$ and have children.

A run of a CFSA is *simple* if all configuration trees of the run are simple. □

**Lemma 2.** *Let $s$ be a shuffle expression, $M_s$ the corresponding CFSA, and $w$ a word. Then $M_s$ has a simple accepting run on $w$, if and only if $M_s$ has any accepting run on $w$.*

*Proof (Sketch).* For the “only if” direction we note that every simple accepting run is an accepting run.

For the opposite direction, we provide a rewrite procedure that rearranges the configuration trees in an accepting, but not simple, run $\rho$. As we shall see, it suffices to apply this procedure a finite number of times to turn any accepting run into a simple accepting run.

Assume that $M_s$ has an accepting run $\rho = t_0, t_1, \ldots, t_n$ on $w$, and that $\rho$ is not simple. Let $t_i$ be the first non-simple tree. Then the transition from $t_{i-1}$ to $t_i$ must have been a vertical transition of the form $(p', \varepsilon, q[p^{\odot}])$ that changed the label of some leaf node $u$ from $p'$ to $q$ and gave it a number of children with label $p$, say $m$ children. Also, there must be an ancestor $v$ of $u$ (possibly, $v = u$)

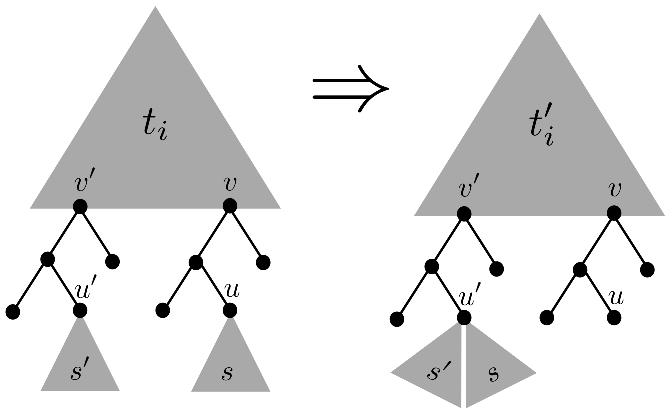

**Fig. 2.** Children obtained through shuffle-closure can be moved between descendants of symmetrically equivalent nodes.

and a node $v'$ such that $v$ and $v'$ are symmetrically equivalent in $t_i$. Let $\phi$ be the corresponding automorphism on $prune(t_i, v, v')$. Let $u' = \phi(u)$. If all the children of $u$ were instead children of $u'$, the tree $t_i$ would be a simple configuration tree. And, indeed, because of the vertical separation of $M_s$, the transition that labeled $u'$ by $q$ must have been $(p', \varepsilon, q[p^{\odot}])$. Thus, it could as well have created $m$ extra children of $u'$ with label $p$, in addition to the children it originally created. This would not have affected any transitions up to configuration tree $t_{i-1}$. Symmetrically, the transition from $t_{i-1}$ to $t_i$ might not have created any children at all under $u$. Thus, with the same sequence of transitions, we might as well have ended up with the configuration tree $t'_i$ which is identical to $t_i$ except that $u$ has no children in $t'_i$ and $u'$ has $m$ more $p$-labeled children than in $t_i$.

It remains to argue that any sequence of transitions used in $\rho$ from $t_i$ forward is also possible from $t'_i$. Let $j > i$ be the smallest number such that in $t_j$, either $v$ has no children or $v'$ has no children. We show that the partial run $\rho_{i,j} = t_i, \ldots, t_j$ can be mirrored in a partial run $\rho'_{i,j} = t'_i, \ldots, t'_j$, using the same transitions. If a transition of $\rho_{i,j}$ affects a node in $t_k$ that doesn't belong to the subtree of $v$ or $v'$, we mirror it directly on $t'_k$. Now consider a transition from $t_k$ to $t_{k+1}$ that affects a node in a subtree of $v$ or $v'$. If the same operation is possible on $t'_k$, we perform it. If not, this can only have two causes.

1. The affected node in $t_k$ is a descendant of $u$, that doesn't exist in $t'_k$. In this case, we perform the operation on the corresponding child of $u'$.
2. The affected node in $t_k$ is $u'$, which, in $t'_k$ still has children. In this case, we perform the operation on $u$.

In each of $t_i$ and $t_j$, we have that exactly one of $v$ and $v'$ is childless. If this is the same node in both trees, they are identical and we are done. If not, we still have to argue that the transitions from $t_j$ forward can be mirrored from $t'_j$. If not, we use the fact that in $t_i$ and $t'_i$, $v$ and $v'$ were symmetrically equivalent. Thus we are free to use the automorphism $\phi$ to reinterpret the sequence $t'_i, \ldots, t'_j$. Under this reinterpretation, $t_j$ and $t'_j$ are identical.

After performing the above operation, all configuration trees up to and *including* $t_i$ are simple. This means that after going through the procedure at most a linear number of times, all configuration trees will be simple. $\square$

Lemma 2 concludes the second step in our proof outline. What remains is to provide a compact representation for simple configuration trees.

**Definition 8.** Let $M_s = (Q, \Sigma, \delta, q_0)$ be the CFSA corresponding to shuffle expression $s$ and let $t$ be a simple configuration tree of $M_s$. We define the *compact configuration tree* $\mathrm{cct}(t)$ corresponding to $t$ by induction on the structure of $t$.

– If $t = q$, then $\mathrm{cct}(t) = q$.
– If $t = q[t_1, \ldots, t_k]$ and $q \in Q \setminus \mathit{scp}(M_s)$, then $\mathrm{cct}(t) = q[\mathrm{cct}(t_1), \ldots, \mathrm{cct}(t_k)]$.
– If $t = q[t_1, \ldots, t_k]$ and $q \in \mathit{scp}(M_s)$, then

$$\mathrm{cct}(t) = q[(\mathrm{cct}(t'_1), n_1), \ldots, (\mathrm{cct}(t'_m), n_m)] \enspace ,$$

where
1. $t'_1, \ldots, t'_m$ is an enumeration of the elements in $\{t_1, \ldots, t_k\}$, so $t'_i$ is not isomorphic to $t'_j$ for any $i, j \in [m]$, making $m$ the number of unique trees, up to isomorphism, in $t_1, \ldots, t_k$,
2. $n_i = |\{j \mid j \in [k], t_j \text{ isomorphic to } t'_i\}|$ for all $i$.

Intuitively, under $q \in \mathit{scp}(M_s)$, we only remember which types of subtrees appear, and annotate each of them with a "repetition counter", which encodes the number of times they appear.

We write $\mathrm{CCT}(M_s)$ for the set of all compact configuration trees of $M_s$. $\square$

It should be clear that there is a one-to-one correspondence between simple configuration trees $t$ and their respective compact configuration trees $\mathrm{cct}(t)$.

**Lemma 3.** *Let $M = (Q, \Sigma, \delta, I)$ be a CFSA corresponding to a shuffle expression. Then there exists a constant $k \in \mathbb{N}$ that depends only on $M$ such that for any string $w \in \Sigma^*$ of length $n$, the number of possible simple configuration trees of $M$ on $w$ is bounded by $O(n^k)$.*

*Proof.* **Small configuration trees.** First, let us note that any intermediate configuration tree needs to contain at most $n + 1$ leaf nodes. Whenever a configuration tree contains $n + 1$ leaf nodes, by the pigeon hole principle, at least one of the states must ultimately derive $\varepsilon$, since there are only $n$ symbols in the string. As such, whenever a configuration contains $n + 1$ leafs we can safely nondeterministically choose a leaf state which can derive $\varepsilon$ and replace it by $t_\varepsilon$ in the next step. We can arrive at $n + 1$ leaf node states in total when a shuffle operation produces its two children, but the above procedure can then remove one state in the next step.

Since an acyclic CFSA will have configuration trees of height at most $|Q|$, no configuration tree needs to be of size greater than $(n+1)|Q|$.

**Small simple configuration trees.** Let $t \in \mathrm{CCT}(M)$ be an arbitrary intermediate compact configuration tree of $M$ running on $w$. Since $M$ is acyclic we know that $t$ is in the set $S = \{t \in \mathrm{CCT}(M) \mid \mathit{height}(t) \leq |Q|\}$. No tree in $S$ has more than $(|Q|+1)^{2^{|Q|}}$ nodes which are pairwise *not* symmetrically equivalent. This follows directly from Definition 6, since it prunes all descendant $\mathit{scp}(M)$-nodes, leaving trees that have a branching factor of at most 2, height at most $|Q|$, and labeled by at most $|Q|$ different symbols (plus one to account for non-existing nodes).

Note that $t$ is (derived from) a simple configuration tree, as in Definition 7, which means that in any set of symmetrically equivalent nodes there is at most one whose corresponding subtree contains an $\mathit{scp}(M)$-labeled node that has children. Take a set $\{v_1, \ldots, v_n\}$ of symmetrically equivalent nodes ($n$ can be arbitrarily large). Then, $\{t/v_1, \ldots, t/v_n\}$ contains at most two unique trees, the single one with $\mathit{scp}(M)$-labeled nodes with children being one, while all other subtrees are necessarily isomorphic. This immediately implies that $t$ contains at most $2(|Q|+1)^{2^{|Q|}}$ unique, up to isomorphism, subtrees.

All that remains is to note that $t$ is a compact configuration tree, so every node either has at most two children (non-$\mathit{scp}$ nodes) or it has only unique, up to isomorphisms, children. Since there are only $2(|Q|+1)^{2^{|Q|}}$ unique subtrees, and $\mathit{height}(t) \leq |Q|$ this means that $t$ has (in a vast overestimate) at most $c = ((|Q|+1)^{2^{|Q|}})^{|Q|}$ nodes. Notice that this number depends only on $M$.

**Conclusion.** We have established that for $M$ running on $w$ every compact configuration tree has at most $c$ nodes. This also means that they contain at most $c$ repetition counters (the counters that are placed as part of the children in $\mathit{scp}$-nodes in the $\mathrm{CCT}(M)$ construction). We have also shown that any intermediary configuration tree of $M$ running on $w$ contains at most $(n+1)|Q|$ nodes.

To conclude, we note that during any step of a run of $M$ on $w$, there are less than $(|Q|+1)^c$ possible compact configuration trees when ignoring the values of the repetition counters. Furthermore, there are less than $(n+1)|Q|$ "units" to be divided among the $c$ counters, which can be done in less than $((n+1)|Q|)^c$ ways. Consequently, there are less than $(|Q|+1)^c((n+1)|Q|)^c$ possible compact configuration trees for any step of $M$. Since $c$ depends only on $M$, and the simple configuration trees are one-to-one with the compact configuration trees, this establishes the desired bound of $O(n^k)$ with $k$ depending only on $M$. $\square$

Finally, we are ready to prove Theorem 8.

*Proof (of Theorem 8).* As in the proof of Theorem 7, we outline an extension of the CYK algorithm. The extension maintains triples consisting of a nonterminal from the grammar $G$ and two configuration trees with respect to $M$. A triple $(A, t, t')$ is assigned to a substring $w'$ of the input string $w$ if

1. $w' = w'_1 \odot w'_2$,

2. the string $w_1'$ can take $M$ from $t$ to $t'$, and
3. the string $w_2'$ can be derived from $A$ in the grammar $G$.

A pair of triples $(A, t, t')$ and $(B, t', t'')$ for the substrings $w'$ and $w''$ can be combined into a triple $(C, t, t'')$ for the substring $w'w''$ if there is a derivation rule $C \to AB$ in $G$. To decide whether there is a parse for $w$, one starts by deriving all possible triples for every substring of $w$ of length 1, and then uses the above combination rule to dynamically complete the parse chart.

A string of length $n$ has $O(n^2)$ substrings, which means that $O(n^2)$ sets of triples have to be computed. From Lemma 3 we know that there is a $k \in \mathbb{N}$, that depends only on the shuffle language involved, such that no more than $O(n^k)$ distinct configuration trees have to be considered. If $G$ has $m$ nonterminals, there are thus no more than $O(m \cdot n^k)$ possible triples. Given that we have the sets of triples for all substrings of $w$, deciding whether a particular triple belongs to the set of triples for $w$ can be done in polynomial time. Thus, since $m$ and $k$ are constants, the problem is polynomial in the length $n$ of the string. $\square$

## 4.4 The membership problem for CF $\odot$ CF

Next, we show that the uniform membership problem for $\mathcal{L}(A_1) \odot \mathcal{L}(A_2)$, where $\mathcal{L}(A_1)$ and $\mathcal{L}(A_2)$ are context-free languages, is NP-complete.

**Proof outline.** First, Definition 9 and 10 recall the definitions of push-down automata, which are equivalent to context-free grammars, and two-stack push-down automata, which are equivalent to Turing machines. These definitions are well known, but for completeness, and because we will use a slightly specialized version of the definitions, we include them here. Then, Definition 13 gives a reduction from an arbitrary two-stack push-down automaton $A$ to a push-down automaton $A_{sim}$, such that $A$ accepts a string $a$ if and only if $A_{sim}$ accepts some string $a \cdot \$ \cdot s$ where $s$ is a valid sequence of stack operations (shown in Lemma 4). The idea is that $A_{sim}$ uses its own, single, stack to simulate the first stack in $A$, and, whenever $A$ would perform an operation on its second stack, for example popping "0", $A_{sim}$ instead reads a string encoding of that operation, for example "$[\text{pop}_0]$". This means that as long as the suffix $s$ of stack operations behaves as a stack should (that is, the symbols popped correspond to those pushed) $A_{sim}$ will behave just like $A$. To force $A_{sim}$ to always read such a valid stack operation sequence we start (in Definition 16) by constructing an input string $a' = a \cdot \$ \cdot \$ \cdot S$ where $S$ is a template repetition of stack operations, and then (in Definition 15) construct a context-free language $\mathcal{L}(A_{comp})$ which contains strings $\$ \cdot \hat{s}$ where $\hat{s}$ is the *complement* of a valid stack operation sequence with respect to the template $S$. This means that $a' \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$ if and only if $A$ accepts $a$, since $A_{comp}$ forces $A_{sim}$ to read only valid stack operations from its part of $a'$ (concluded in Theorem 9).

We will mostly represent context-free languages by push-down automata. It is well known that we can convert any context-free grammar into an equivalent push-down automaton (and vice versa) in polynomial time [16]. Lets recall a simple definition of push-down automata, here fixed to using only a binary stack alphabet (without loss of generality).[2]

**Definition 9 (Push-down automata).** A push-down automaton (PDA) is a tuple $(Q, \Sigma, \delta, q_0, F)$ where
- $Q$ is a finite set of states,
- $\Sigma$ is a finite alphabet of input symbols ($\varepsilon \notin \Sigma$),
- $\delta \subset Q \times ((\Sigma \cup \{\varepsilon\}) \times \{\varepsilon, 0, 1\}) \times (Q \times \{\varepsilon, 0, 1\})$ is a finite set of transitions,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ are the final states.

We write $(q, a, s, q', s') \in \delta$ as $q \xrightarrow{a,s/s'} q'$. This means that if the automaton is in state $q$ and it can read the symbol $a$ from the input (if $a = \varepsilon$ nothing is read) and can pop the binary value $s$ off the top of the stack (if $s = \varepsilon$ nothing is popped) it may choose to go to state $q'$, pushing $s'$ onto the top of the stack (if $s' = \varepsilon$ nothing is pushed).

As usual, the computation starts in state $q_0$ and the machine accepts if and only if some sequence of transitions leave the automaton in a state in $F$ when the entire input string has been read. Whenever the stack is empty the bottom bit 0 will be read, and may be popped without effect. $\square$

Recall also, that when nondeterministic push-down automata are extended to have two independent stacks they become computationally equivalent to a nondeterministic Turing machine, and can simulate each step in a Turing machine run using only a constant number of transitions.[3] Lets recall the definition of these automata as well.

**Definition 10 (2-PDA).** A two-stack push-down automata (2-PDA) is a tuple $(Q, \Sigma, \delta, q_0, F)$, where
- $Q$ is a finite set of states,
- $\Sigma$ is a finite alphabet of input symbols,
- $\delta \subset (Q \times (\Sigma \cup \{\varepsilon\}) \times \{\varepsilon, 0, 1\} \times \{\varepsilon, 0, 1\}) \times (Q \times \{\varepsilon, 0, 1\} \times \{\varepsilon, 0, 1\})$ is a finite set of transitions,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ are the final states.

We write $(q, a, s_1, s_2, q', s_1', s_2') \in \delta$ as $q \xrightarrow{a,s_1/s_1',s_2/s_2'} q'$. This automaton operates just like a PDA, only with two independent stacks. $\square$

To simplify the reduction, we define some additional properties that the input 2-PDA $A$ must exhibit. The following definition establishes the property of $A$ being *input-partitioned.* This property requires that $A$ always starts by reading its

[2] Since $\varepsilon$-transitions are allowed the automata can simulate a richer stack alphabet by representing each symbol by some fixed-length binary string.

[3] The stacks can be used to simulate the work tape by letting the first stack contain the portion of the tape to the left of the head, in reverse order, while the second stack contain the portion to the right of the head. The head can then be moved by popping a symbol from one stack and pushing it onto the other.

entire input without using its second stack, and, only when all input is consumed, switches over to performing arbitrary computations using both stacks.

**Definition 11 (Input-partitioning).** Given a 2-PDA $A = (Q, \Sigma, \delta, q_0, F)$, an *input-partitioning* of $A$ is a tuple $(Q_{\text{input}}, Q_{\text{compute}})$ with $Q = Q_{\text{input}} \cup Q_{\text{compute}}$, $Q_{\text{input}} \cap Q_{\text{compute}} = \emptyset$, $q_0 \in Q_{\text{input}}$, $F \subseteq Q_{\text{compute}}$, and for all transitions $q \xrightarrow{a,s_1/s_1',s_2/s_2'} q'$ in $\delta$ it holds that

- $q \in Q_{\text{input}}$ implies that $s_2 = \varepsilon$, $s_2' = \varepsilon$,
- $q \in Q_{\text{compute}}$ implies that $q' \in Q_{\text{compute}}$ and $a = \varepsilon$,
- $q \in Q_{\text{input}}$ and $q' \in Q_{\text{compute}}$ only for one unique transition, which also has $a = \varepsilon$, $s_1 = \varepsilon$ and $s_1' = \varepsilon$. □

This means that an input-partitioned automaton $A$ starts out in a state in $Q_{\text{input}}$, and no transition from a state in $Q_{\text{input}}$ ever touches the second stack, but may read input. It must then, sooner or later, take the unique switching transition $q \xrightarrow{\varepsilon,\varepsilon/\varepsilon,\varepsilon/\varepsilon} q'$, with $q \in Q_{\text{input}}$ and $q' \in Q_{\text{compute}}$, after which it will always be in some state in $Q_{\text{compute}}$. No transitions from states in $Q_{\text{compute}}$ may read input, or go to a state in $Q_{\text{input}}$, but they may use both stacks.

**Remark.** In the sequel, we will, without loss of generality, assume that we have an input-partitioning for any 2-PDA used. A general 2-PDA is equivalent to a Turing machine, and input partitioning simply forces the machine to start by transferring all the input to its work tape. Clearly, every Turing machine can be rewritten into an equivalent TM which accepts the same input strings (when suitably encoded) as the starting content of its work tape. □

For convenience, we name two alphabets that will be frequently used in the remainder of this section.

**Definition 12 ($\Gamma$,$\hat{\Gamma}$).** $\Gamma = \{\text{push}_0, \text{push}_1, \text{pop}_0, \text{pop}_1\}$ and $\hat{\Gamma} = \Gamma \cup \{], [, \$\}$. □

The next definition contains the key construction of this section. It shows how to, given a 2-PDA $A$, construct a PDA $A_{sim}$ such that $A$ accepts input string $w$ if and only if there exists a special string $s$ such that $w \cdot \$ \cdot s \in \mathcal{L}(A_{sim})$. The requirement is that $s$ encodes a valid sequence of stack operations. The construction works by letting the stack of $A_{sim}$ simulate the first stack of $A$, and making $A_{sim}$ read the stack operations for the second stack from $s$.

**Definition 13 ($A_{sim}$).** Given the 2-PDA $A = (Q, \Sigma, \delta, q_0, F)$, with input--partitioning $(Q_{\text{input}}, Q_{\text{compute}})$, we construct the PDA $A_{sim} = (Q', \Delta, \delta', q_0, F)$ as follows.

- $\Delta = \Sigma \cup \hat{\Gamma}$, (we assume $\Sigma \cap \hat{\Gamma} = \emptyset$.)
- To construct $Q'$ we use the mapping $f : \{\text{push}, \text{pop}\} \times \{\varepsilon, 0, 1\} \to (\Sigma \cup \hat{\Gamma})^*$ that is defined by

$$f(x, v) = \begin{cases} \varepsilon & \text{when } v = \varepsilon, \\ [\cdot x_v \cdot] & \text{otherwise.} \end{cases}$$

Now, $Q'$ is the union of $Q$ and the set of states

$$\{\tau_S^{[q']} \mid q \xrightarrow{a,s_1/s_1',s_2/s_2'} q' \in \delta, S \text{ is a suffix of } f(\text{pop}, s_2) \cdot f(\text{push}, s_2')\} \ .$$

Lastly, $\delta'$ contains the rules $\{\tau^{[q]}_{a_1\cdots a_n} \xrightarrow{a_1,\varepsilon/\varepsilon} \tau^{[q]}_{a_2\cdots a_n} \mid \tau^{[q]}_{a_1\cdots a_n} \in Q' \setminus Q\}$ and $\{\tau^{[q]}_{\varepsilon} \xrightarrow{\varepsilon,\varepsilon/\varepsilon} q \mid q \in Q\}$. Additionally, for every transition $q \xrightarrow{a,s_1/s_1',s_2/s_2'} q'$ in $\delta$

- if $q, q' \in Q_{\text{input}}$ then the transition $q \xrightarrow{a,s_1/s_1'} q'$ is in $\delta'$,
- if $q \in Q_{\text{input}}$ and $q \in Q_{\text{compute}}$ then the transition $q \xrightarrow{\$,\varepsilon/\varepsilon} q'$ is in $\delta'$,
- if $q, q' \in Q_{\text{compute}}$ then the transition $q \xrightarrow{\varepsilon,s_1/s_1'} \tau^{[q']}_{f(\text{pop},s_2)\cdot f(\text{push},s_2)}$ is in $\delta'$.

□

In the sequel, we will often be using strings of symbols to represent stack operation sequences. To simplify this, let us define the set VSR (for Valid Stack Runs) to contain all valid sequences of stack operations for a binary stack alphabet. Note especially that this includes the possibility of push operations with no corresponding pop, corresponding to sequences which end with a non-empty stack. Additionally, we define two functions to format these strings in convenient ways.

**Definition 14 (Valid stack run).** Define VSR and $S_{\text{VSR}}$ as follows.

- VSR $= \mathcal{L}(G)$ where $G$ is the context-free grammar $G = (Q, \Sigma, \delta, q_0)$ with nonterminals $Q = \{q_0, b\}$, alphabet $\Sigma = \Gamma$ and $\delta$ containing the following rules.

$$\begin{aligned} q_0 &\to \text{push}_0\, q_0 \mid \text{push}_1 q_0 \mid q_0 q_0 \mid b \\ b &\to \text{push}_0\, b\, \text{pop}_0 \mid \text{push}_1\, b\, \text{pop}_1 \mid bb \mid \varepsilon \end{aligned}$$

- $S_{\text{VSR}} : \text{VSR} \to \hat{\Gamma}^*$, such that we have $S_{\text{VSR}}(r_1 \cdots r_n) = [r_1] \cdots [r_n]$ for all $r_1 \cdots r_n \in$ VSR. Also, let

$$\bar{S}_{\text{VSR}}(p_1 \cdots p_n) = \begin{cases} \varepsilon & \text{if } n = 0, \\ [\text{push}_1\, \text{pop}_0\, \text{pop}_1] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) & \text{if } p_1 = \text{push}_0, \\ [\text{push}_0\, \text{pop}_0\, \text{pop}_1] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) & \text{if } p_1 = \text{push}_1, \\ [\text{push}_0\, \text{push}_1\, \text{pop}_1] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) & \text{if } p_1 = \text{pop}_0, \\ [\text{push}_0\, \text{push}_1\, \text{pop}_0] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) & \text{if } p_1 = \text{pop}_1. \end{cases}$$

□

**Observation 3 (Stack runs).** Note that for all $s \in$ VSR, all prefixes of $s$ are also in VSR. Also, note that $S_{\text{VSR}}$ and $\bar{S}_{\text{VSR}}$ complement each other in the sense that for all $p \in$ VSR we have

$$\underbrace{[[\text{push}_0\, \text{push}_1\, \text{pop}_0\, \text{pop}_1]] \cdots [[\text{push}_0\, \text{push}_1\, \text{pop}_0\, \text{pop}_1]]}_{|p| \text{ times}} \in S_{\text{VSR}}(p) \odot \bar{S}_{\text{VSR}}(p).$$

□

Now we have all the tools needed to prove that $A_{sim}$ indeed have all the properties claimed for it, thereby establishing the link between our $A_{sim}$-construction and the 2-PDA $A$.

**Lemma 4.** *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a 2-PDA with input-partitioning and let $w \in \Sigma^*$ be any input string. Then the following holds.*

1. *$A$ has an accepting run on $w$ if and only if there exists some $p \in VSR$ such that $w \cdot \$ \cdot S_{VSR}(p) \in \mathcal{L}(A_{sim})$ where $A_{sim} = (Q', \Delta, \delta', q_0, F)$ is constructed as in Definition 13.*
2. *If $A$ has an accepting run on $w$ of length $n$, then there exists a $p \in VSR$ that fulfills the above with $|p| \leq 2n$.*

*Proof.* For all strings $s \in (\Sigma \cup \hat{\Gamma})^*$ define $\gamma : (\Sigma \cup \hat{\Gamma})^* \to \Gamma^*$ (recall $\hat{\Gamma}$ and $\Gamma$ from Definition 12) so that $\gamma(s)$ produces $s$ with all non-push/pop symbols removed (notably $\gamma(w \cdot \$ \cdot S_{\mathrm{VSR}}(p)) = p$). Then define $\sigma : \mathrm{VSR} \to \{0,1\}^*$ as the function which for all $s \in \Gamma^*$ produces the stack contents resulting from applying the stack operations in $s$, in order, to an initially empty stack. Notice that $\sigma(\gamma(s))$ is well-defined for all prefixes of $w \cdot \$ \cdot S_{\mathrm{VSR}}(p)$.

Let $(Q_{\mathrm{input}}, Q_{\mathrm{compute}})$ be the input partitioning of $A$. Write configurations of $A$ (when in a $Q_{\mathrm{compute}}$ states) as tuples of the form

$$(q, B_1, B_2) \in Q_{\mathrm{compute}} \times \{0,1\}^* \times \{0,1\}^*$$

where $q$ is the current state and $B_1$ and $B_2$ the current contents of Stack 1 and Stack 2, respectively. Write configurations of $A_{sim}$ as tuples of the form

$$(q, B_1, B_2) \in Q_{\mathrm{compute}} \times \{0,1\}^* \times \{0,1\}^*$$

where $q$ is the current state, $B_1$ is the current stack contents, and $B_2 = \sigma(\gamma(s))$ where $s \in \Sigma^*$ is the part of the input string already read.

**Base case** Assume that $A$ has a run on $w$. Let $(q', B_1, \varepsilon)$ be the configuration of $A$ after the unique transition $q \xrightarrow{\varepsilon,\varepsilon/\varepsilon,\varepsilon/\varepsilon} q'$ with $q \in Q_{\mathrm{input}}$ and $q' \in Q_{\mathrm{compute}}$ is taken (this unique transition must be taken at some point and Stack 2 will be empty by Definition 11). Since $A_{sim}$ by construction contains all the same rules for the states in $Q_{\mathrm{input}}$ it will be able to, with $q \xrightarrow{\$,\varepsilon/\varepsilon} q'$ as the last transition, reach the configuration $(q', B_1, \varepsilon)$ reading the string $w \cdot \$$. This establishes the base case, for all runs of $A$ on the string $a$ we will get to a configuration which $A_{sim}$ can reach on the string $w \cdot \$$, and trivially $\gamma(w \cdot \$) \in \mathrm{VSR}$.

**Inductive step** Assume that $A$ and $A_{sim}$ are in configuration $(q, B_1, B_2)$, let $s \in \Delta^*$ be the input already read by $A_{sim}$, and assume that $\gamma(s) \in \mathrm{VSR}$. Let $s_1, s_2 \in \{0,1\}$ be the top elements of the stack contents $B_1$ and $B_2$ respectively. This means that $A$ can take a transition $q \xrightarrow{w,s_1/s_1',s_2/s_2'} q'$. Let $w_1 \cdots w_m = f(\mathrm{pop}, s_2) \cdot f(\mathrm{push}, s_2')$ for $f$ as in Definition 13. By construction there must exist transitions

$$q \xrightarrow{\varepsilon,s_1/s_1'} \tau^{[q']}_{w_1 \cdots w_n} \xrightarrow{w_1,\varepsilon/\varepsilon} \tau^{[q']}_{w_2 \cdots w_n} \xrightarrow{w_2,\varepsilon/\varepsilon} \ldots \xrightarrow{w_n,\varepsilon/\varepsilon} \tau^{[q']}_{\varepsilon} \xrightarrow{\varepsilon,\varepsilon/\varepsilon} q'$$

in $A_{sim}$. The first transition mimics the stack operations on Stack 1 in $A$, while the remainder makes the input read so far $s' = s \cdot w_1 \cdots w_n$ and $\gamma(w_1 \cdots w_n)$ will be exactly the stack operations performed on Stack 2 by $s_2/s_2'$ in $A$. We already had $B_2 = \sigma(\gamma(s))$ so $\sigma(\gamma(s'))$ will match the resulting Stack 2 in $A$, making the configurations match again after the transitions. Since we know that the (possible) pop $s_2$ was matched in $B_2$ and we assumed that $\gamma(s) \in \text{VSR}$ we also know that $\gamma(s') \in \text{VSR}$.

**Conclusion** The other direction is easily shown in the same way, the string read by $A_{sim}$ being mimicked by the operations on Stack 2 by $A$, and it is necessarily possible by virtue of the stack operation sequence being in VSR. This proves Part 1.

Part 2 follows trivially, the bound on the length of $p$ follows directly from the induction, where $p$ turns out to encode the sequence of stack operations performed on Stack 2 in $A$ during the run. □

Next, we define the PDA $A_{comp}$, which will serve to read the "complement" of a valid stack run.

**Definition 15 ($A_{comp}$).** The language of the PDA $A_{comp}$ is

$$\mathcal{L}(A_{comp}) = \{\$ \cdot \bar{S}_{\text{VSR}}(p) \mid p \in \text{VSR}\}.$$

It can be constructed from the context-free grammar $G = (Q, \Sigma, \delta, q_0)$ with nonterminals $Q = \{q_0, s, b\}$, and rules as follows.

$$\begin{aligned} q_0 &\to \$s \\ s &\to [\text{push}_1\,\text{pop}_0\,\text{pop}_1]s \ \mid\ [\text{push}_0\,\text{pop}_1\,\text{pop}_2]s \ \mid ss \ \mid\ b \\ b &\to [\text{push}_1\,\text{pop}_0\,\text{pop}_1]b[\text{push}_0\,\text{push}_1\,\text{pop}_1] \ \mid \\ &\quad\ [\text{push}_0\,\text{pop}_0\,\text{pop}_1]b[\text{push}_0\,\text{push}_1\,\text{pop}_0] \ \mid\ bb \ \mid \varepsilon \end{aligned}$$

□

Next, the last definition of this section shows how an input string for the 2-PDA $A$ construct a string to serve as input to the membership problem for the constructed language $\mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$.

**Definition 16 (Formatted input).** For every string $w$ over the alphabet $\Sigma$ and every $n \in \mathbb{N}$ we define the function INPUT : $\Sigma^* \times \mathbb{N} \to \Delta^*$, where $\Delta = \Sigma \cup \hat{\Gamma}$ (assume that $\Sigma \cap \hat{\Gamma} = \emptyset$), as

$$\text{INPUT}(w) = w \cdot \$ \cdot \$\, \underbrace{[[\text{push}_0\,\text{push}_1\,\text{pop}_0\,\text{pop}_1]] \cdots [[\text{push}_0\,\text{push}_1\,\text{pop}_0\,\text{pop}_1]]}_{n \text{ times}}.$$

□

The next lemma establishes that $A_{comp}$ will in fact necessarily leave only valid stack runs as the suffix of strings produced by INPUT when shuffled with $A_{sim}$.

In the statement of Lemma 5, the PDA $A_{sim}$ is obtained from the TM $A$ using the construction in Definitions 13, and the PDA $A_{comp}$ is as defined in Definition 15.

**Lemma 5.** *Let $\Sigma \cap \Gamma = \emptyset$. Then, for every TM $A$, every string $w \in \Sigma^*$, and every $n \in \mathbb{N}$ it holds that* INPUT$(w, n) \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$ *if and only if there exists some $p \in VSR$ with $|p| = n$ such that $w \cdot \$ \cdot S_{VSR}(p) \in \mathcal{L}(A_{sim})$.*

*Proof.* Given $w \in \Sigma^*$ and $n \in \mathbb{N}$, the string produced by $\textsc{input}(w, n)$ is of the form $w \cdot \$ \cdot \$ \cdot [[\text{push}_0\, \text{push}_1\, \text{pop}_0\, \text{pop}_1]] \cdots [[\text{push}_0\, \text{push}_1\, \text{pop}_0\, \text{pop}_1]]$. Both $A_{sim}$ and $A_{comp}$ will, by construction, accept only strings with balanced, non-nested brackets. Additionally, we know that $A_{comp}$ reads only strings of the form $\$ \cdot \bar{S}_{\text{VSR}}(p)$ for some $p \in \text{VSR}$. These facts alone forces $A_{sim}$ to read a string of the form $w \cdot \$ \cdot [p_1] \cdots [p_n]$ for some $p_1, \ldots, p_n \in \Gamma$. As noted in Observation 3 $\bar{S}_{\text{VSR}}$ and $S_{\text{VSR}}$ behave as complements, so we will in fact have $p_1 \cdots p_n = p \in \text{VSR}$, so the string will be accepted if and only if $w \cdot \$ \cdot S_{\text{VSR}}(p) \in \mathcal{L}(A_{sim})$ □

Finally, the following theorem summarizes the main result of the section, bringing together the results of the previous lemmas.

**Theorem 9.** *For an input string $w$ it is an NP-complete problem to decide whether or not $w \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$ when $\mathcal{L}(A_{sim})$ and $\mathcal{L}(A_{comp})$ are context-free languages, even when $\mathcal{L}(A_{sim})$ and $\mathcal{L}(A_{comp})$ are fixed. That is, the non-uniform membership problem for the shuffle of two context-free languages is NP-complete.*

*Proof.* The problem is trivially *in* NP. Membership in context-free languages can be decided in polynomial time, and we can, in polynomial time, guess any $w_1$ and $w_2$ such that $w \in w_1 \odot w_2$ and check if $w_1 \in \mathcal{L}(A_{sim})$ and $w_2 \in \mathcal{L}(A_{comp})$.

NP-hardness can now be shown using the tools we have established. Take any nondeterministic input-partitioned 2-PDA $A$ that solves some NP-hard problem in polynomial time. Let $F : \mathbb{N} \to \mathbb{N}$ be a polynomial such that running $A$ on an input string $w$ takes less than $F(|w|)$ steps. These assumptions can be made since we can convert an arbitrary nondeterministic Turing machine into such a 2-PDA, and a nondeterministic TM can of course solve any problem in NP in polynomial time. Modify $A$ so that it may loop indefinitely on all final states.

Construct $A_{sim}$ from $A$ using Definition 13, take $A_{comp}$ as in definition 15. A string $w$ is accepted by $A$ if and only if $\textsc{input}(w, 2F(|w|)) \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$, by applying Lemma 5 to show that the part of the input left to $A_{sim}$ is restricted to only valid stack runs, and then Lemma 4 to show equivalence with the 2-PDA. Notice also that since the run of $A$ takes at most $F(|w|)$ steps we need at most $2F(|w|)$ stack symbol blocks in the $\textsc{input}$ construction, by Lemma 4. □

## 5 Conclusions and future work

Concurrent finite-state automata combine the expressive power of context-free and shuffle languages. The CFSA languages are properly included in the context-sensitive languages, and minor restrictions of the device suffice to obtain the regular, context-free, and shuffle languages, respectively. CFSA have comparatively nice closure properties, and can be sanity-checked in polynomial time.

To be of practical use, at least the non-uniform membership problem for a language class needs to be efficiently decidable. This is known to be true for the shuffle languages, but our analysis shows that the efficiency depends heavily on the number of shuffle operations used. We also obtain that the non-uniform membership problem remains polynomial for the shuffle of a shuffle language and a context-free language. For the shuffle of two context-free languages, however, it is NP-complete.

Ideally, the uniform membership problem should be solvable in polynomial time. The only language class we studied for which this is the case, unless P=NP, is the shuffle of a regular language and a context-free language.

Future work should strive to determine the complexity of the non-uniform membership problem for further restrictions of CFSA. If even very sparse use of shuffling has a large negative impact on the complexity, one could consider replacing the shuffle operator with weaker alternatives, such as unordered shuffle.

## References

1. G. E. Barton. On the complexity of ID/LP parsing 1. *Comput. Linguist.*, 11(4):205–218, 1985.
2. F. Biegler, M. Daley, and I. McQuillan. On the shuffle automaton size for words. In *DCFS*, pages 79–89, 2009.
3. H. Björklund and M. Bojańczyk. Shuffle expressions and words with nested data. In *Proc. MFCS'07*, pages 750–761, 2007.
4. S. B. Bloom and Z. Ésik. Axiomatizing shuffle and concatenation in languages. *Information and Comptuation*, 139(1):62–91, 1997.
5. M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proc. LICS'06*, pages 7–16, 2006.
6. C. Câmpeanu, K. Salomaa, and S. Yu. Tight lower bound for the state complexity of shuffle of regular languages. *J. Autom. Lang. Comb.*, 7:303–310, January 2002.
7. S. Carberry. Techniques for plan recognition. *User Modeling and User-Adapted Interaction*, 11(1-2):31–48, 2001.
8. M. Daley, M. Domaratzki, and K. Salomaa. Orthogonal concatenation: Language equations and state complexity. *J. Universal Comp. Sci.*, 16(5):653–675, 2010.
9. R. Downey and M. Fellows. *Parameterized Complexity.* Springer-Verlag, 1999.
10. Z. Ésik and M. Bertol. Nonfinite axiomatizability of the equational theory of shuffle. *Acta Informatica*, 35(6):505–539, 1998.
11. V. Garg and M. Ragunath. Concurrent regular expressions and their relationship to petri nets. *Theor. Comput. Sci.*, 96(2):285–304, 1992.
12. W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM J. on Comp.*, 39(4):1486–1530, 2009.
13. S. Ginsburg. *The Mathematical Theory of Context Free Languages.* McGraw-Hill, 1966.
14. J. Gischer. Shuffle languages, petri nets, and context-sensitive grammars. *Comm. ACM*, 24(9):597–605, 1981.
15. J. Högberg and L. Kaati. Weighted unranked tree automata as a framework for plan recognition. In *Proc. Fusion'10*, 2010. To appear.
16. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Ed.).* Pearson Education International, 2003.
17. J. Jedrzejowicz and A. Szepietowski. Shuffle languages are in P. *Theor. Comput. Sci.*, 250(1-2):31–53, 2001.
18. L. Kari and P. Sosík. Aspects of shuffle and deletion on trajectories. *Theor. Comput. Sci.*, 332:47–61, February 2005.

19. M. Kuhlmann and G. Satta. Treebank grammar techniques for non-projective dependency parsing. In *Proc. EACL*, pages 478–486, 2009.
20. A. Mateescu, G. Rozenberg, and A. Salomaa. Shuffle on trajectories: syntactic constraints. *Theor. Comput. Sci.*, 197:1–56, May 1998.
21. A. Mayer and L. Stockmeyer. Word problems – this time with interleaving. *Inform. and Comput.*, 115:293–311, 1994.
22. J. Nivre. Non-projective dependency parsing in expected linear time. In *Proc. ACL-IJCNLP '09*, pages 351–359, 2009.
23. C. Schmidt, N. Sridharan, and J. Goodson. The plan recognition problem: An intersection of psychology and artificial intelligence. *A.I.*, 11(1,2), 1978.