LaBRI

University of Bordeaux

Master's Thesis

# On Minimization of Visibly Pushdown Automata

Author:
Romaric Duvignau

Supervisors:
Anca Muscholl∗
Olivier Gauwin♯

∗ Researcher of the Laboratoire Bordelais de Recherche en Informatique (LaBRI), professor at the Université Bordeaux and member of the Institut Universitaire de France. anca@labri.fr

♯ Member of the LaBRI, and assistant professor at Université Bordeaux. olivier.gauwin@labri.fr

June 6, 2012

# Abstract

Visibly Pushdown Automata (VPA) are a recent model of machines introduced by Alur and Madsuhudan in 2004 that has risen plenty of research interest in the last couple of years. VPA are pushdown automata working on a visibly pushdown alphabet, i.e. each input symbol of the automaton determine uniquely the stack operation (push, pop, or no operation on the stack) to perform. They model the behaviours of several kinds of recursive programs and the increasing amount of research around them testifies of how wide their range of applications in software verification is. Yet some questions remain open, e.g. whether VPA are minimizable within polynomial time. In this master's thesis, we shall argument why minimization of VPA is hard and which alternatives do exist in order to minimize them. Several variants and submodels of VPA are eventually considered in order to accomplish an efficient minimization.

**Key-words** : visibly pushdown automata, state minimization, congruences, complexity.

# Contents

# 1 Introduction

The development of programs for high-risk technologies, e.g. life-critical systems and avionics technologies, has consistently grown over the last decades. Softwares for such critical environments should be completely *error-free* as lives are at stake. Formal Methods have been proved to be an excellent mean to ensure the correctness for systems where safety is of utmost importance.

Among the techniques used to check correctness of programs, abstraction by an automaton model is one of the most used. The model of Visibly Pushdown Automata has lead to a considerable amount of research work[1] since their recent introduction by Alur and Madhusudan [2]. They provide a particularly useful tool for model checking and software verification, especially boolean programs [3] and stream applications [4, 5].

Vpa is a model of stack automata where the input alphabet is partitioned into call, return and local symbols and each set of symbols is associated to a particular action to perform on the stack (push, pop or nothing). The class of languages recognized by Vpa, called Vpl, lies strictly between the set of regular languages and the deterministic context-free languages. They represent a very robust model of automata (comparable to Finite State Automata) and provide in the same time enough expressiveness to model interesting program analysis questions.

We shall study in this master's thesis the open problem of minimization of Vpa. In fact, no polynomial time algorithm is yet known in order to minimize the Vpa, but such an algorithm will provide directly concrete improvements for several applications of the domain. In our study, we show hardness results on several kinds of minimization problems related to Vpa and their variants[2] and propose polynomial time minimization algorithms on submodels[3]. The difficulty of the general minimization problem for Vpa remains unknown, but we give a couple of interesting arguments in favor of a computational hardness result.

In Section 2, we will formally describe the model of Vpa and present a few essential theorems and results. In Section 3, we shall focus on the minimization problem and demonstrate some hardness results. Section 4 will be dedicated to a variant of Vpa: Block Visibly Pushdown Automata, that has been considered as a promising model to accomplish efficient minimization. In Section 5, we shall eventually investigate some minimizable variants and a few minimizable submodels of Vpa.

[1]P. Madhusudan lists on his website more than 50 papers related to Visibly Pushdown Automata since their initial paper from 2004, see [1].

[2]We mean by *variant* of Vpa, a family of Vpa that recognizes exactly the Vpl.

[3]We call a *submodel* of Vpa, a family of Vpa that recognizes a subclass of the Vpl.

# Preliminaries

Before getting to the heart of the matter, let us introduce some general definitions and notations in this short section.

One of the most fundamental notion that will be used through this master's thesis are formal languages, which are sets of words over finite alphabets. An alphabet is just a finite set of letters or symbols, actually like any natural language alphabet e.g. latin or greek alphabets. A word over such an alphabet is a finite sequence of letters taken from the alphabet. At last a language is just a set of such words which can be either finite or infinite. We will usually denote alphabets by capital greek letter (e.g. $\Sigma$ or $\Gamma$); symbols by small latin letters (e.g. $a, b, c, r, l$); words by the letter $w$ (or $u$, $v$) and languages by the capital letter $L$. One special word is $\varepsilon$ which is the word that has no letter (i.e. its length is 0).

A natural operation on words noted **.** is the concatenation[4]. For instance the word $w = u.u'$ is the word obtained by juxtaposition of $u$ and $u'$. The set $\Sigma^*$ is the infinite set of words that can be written only with symbols of the alphabet $\Sigma$, which is more formally the free monoid on the set $\Sigma$. For every word $w \in \Sigma^*$, we will denote $\text{pref}(w)$ the set of prefixes of $w$, i.e. the set of words $v \in \Sigma^*$ such that there exists a $v' \in \Sigma^*$ and $v.v' = w$. In the same fashion, we note $\text{suff}(w)$ the set of suffixes of $w$, i.e. the set of words $v \in \Sigma^*$ such that there exists a $v' \in \Sigma^*$ and $v'.v = w$. These notions are naturally extended to suit for languages, thus $\text{Pref}(L)$ is the union of every $\text{pref}(w)$ for every $w \in L$ and $\text{Suff}(L)$ is the union of every $\text{suff}(w)$.

In order to denote the $i^{th}$ letter of a word $w$, we will use the notation $w[i]$. We will also note $w[i..j]$ for the substring of $w$ that starts at position $i$ and ends in position $j$ (both included[5]). However, if we write only $[i..j]$, it means the subset of the natural numbers from $i$ to $j$ (i.e. the set $\{i, i+1, ..., j\}$).

We will on several occasions mention a simple model of machines called Deterministic Finite Automata or DFA. We will use here the most standard definition of such machine as a 4-tuple $(Q, q_0, Q_F, \delta)$ over $\Sigma$ (the input alphabet of the automaton). In the definition, $Q$ is a finite set of states, $q_0 \in Q$ the initial state, $Q_F \subseteq Q$ a set of final states and $\delta : Q \times \Sigma \to Q$ its (partial) transition function. We guide the reader towards [6] for all basic results about languages and automata that are not proven here.

At last, definitions about complexity classes P (or PTIME for Polynomial Time), NP (Non-deterministic Polynomial Time), PSPACE (Polynomial Space) etc are the standard ones, see e.g. [7] for details.

[4]The concatenation can be omitted when it burdens the notation.

[5]By convention the first position of a word is 1, hence $w[0]$ denotes $\varepsilon$.

# 2 Visibly Pushdown Automata

*Visibly Pushdown Automata* (VPA) were introduced by [2] as a robust and tractable framework in order to model interesting program analysis questions. VPA are stack automata working on a partitioned alphabet, where the input letter determines uniquely how the stack will evolve (i.e. whether the automaton shall push or pop the stack).

Languages accepted by such automata are called *Visibly Pushdown Languages* (VPL). The class of such languages forms a strict subclass of deterministic context-free languages (VPA are determinizable) and a strict superclass of regular languages. An instance of a VPL can be the language $L = \{a^n b^n \mid n \in \mathbb{N}\}$ with $\boldsymbol{a}$ being a call symbol and $\boldsymbol{b}$ a return symbol, and as one can notice there are no DFA that recognize it (i.e. $L$ is not regular). However, the context-free language $L' = \{(a^n b^n + b^n a^n) \mid n \in \mathbb{N}\}$ cannot be recognize by any VPA whatever the partition of the input alphabet. Also, the class of VPL turns out to be closed under all boolean set operations, as well as renaming, concatenation and Kleene-$*$. Moreover, several fundamental decision problems such as inclusion that are undecidable for traditional pushdown automata, become *only* EXPTIME-complete for VPA.

Despite this robustness and tractability, VPA are powerful enough to model many program analysis questions, such as algorithmic verification of recursive programs (see e.g. [2, 8]) and model-checking of software programs that can be formalized as pushdown models [9]. Besides, VPA have been extensively studied in the last few years (as this website [1] can testify), and applications in different contexts have appeared: for instance the processing of XML streams [4, 5], decidability results on infinite games [10] and semantic of program languages [11].

There exists an alternative formulation of visibly pushdown automata as nested word automata [12]. This other approach describes a model of finite automata (i.e. without a stack) over a visibly pushdown alphabet where the input of the model are nested words. Such words are just string augmented by a matching relation between symbols defined by the pushdown alphabet. This model of automaton shares many aspects with VPA and will not be considered here.

This section is intended to introduce the automaton model of VPA and to give the reader all the definitions needed to follow the rest of the thesis. Thus we shall introduce in Section 2.1 some formal definitions and propositions, then in Section 2.2 we will give the closure properties of the class of VPL. We will summarize in Section 2.3 all known results for basic operations on VPA in addition of some comparisons with other automaton models. Finally, we will conclude by giving some applications of VPA in Section 2.4.

## 2.1 Definitions

### 2.1.1 Formal Definitions

A *visibly pushdown alphabet* $\hat{\Sigma}$ is a set of input symbols partitioned into three distinct sets $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ such that $\Sigma_{call}$ is a finite set of **calls**, $\Sigma_{ret}$ is a finite set of **returns**, and $\Sigma_{loc}$ is a finite set of **local actions**[1]. For any pushdown alphabet $\hat{\Sigma}$, we note $\Sigma$ for the union of the three sets, i.e. $\Sigma = \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{loc}$.

**Definition** Let $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ be a visibly pushdown alphabet and $u$ a word of $\Sigma^*$. We denote $|u|_c$ the number of calls in $u$ and $|u|_r$ the number of returns. We define the set of matched call words as

$$MC(\hat{\Sigma}) = \{u \in \Sigma^* \mid \forall u' \in \text{suff}(u), |u'|_c \leq |u'|_r\}$$

Similarly, the set of matched return words is defined as

$$MR(\hat{\Sigma}) = \{u \in \Sigma^* \mid \forall u' \in \text{pref}(u), |u'|_r \leq |u'|_c\}$$

At last, the set of well-matched words is $WM(\hat{\Sigma}) = MC(\hat{\Sigma}) \cap MR(\hat{\Sigma})$.

**Remark** For every word $w \in \Sigma^*$ over a visibly pushdown alphabet $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$, there exist $w_1$ and $w_2$ in $\Sigma^*$ such that $w = w_1.w_2$, $w_1 \in MC(\hat{\Sigma})$ and $w_2 \in MR(\hat{\Sigma})$.

A Vpa is a pushdown automaton working on such partitioned alphabet. Intuitively, when the automaton is reading a *call* symbol it will push a symbol on the stack and change its internal state, on reading a *return* symbol it will pop the stack and update its state depending of what was on the stack, and if the input symbol is a *local action* the automaton will ignore the stack and only change its current state. Hence, after reading some word $w \in \Sigma^*$, one can know the depth of the stack of any Vpa by just looking at the pushdown alphabet and not at the transitions of a particular automaton. The formal definition is as follows:

**Definition** A Visibly Pushdown Automaton is a 5-tuple $\mathcal{A} = (Q, Q_I, Q_F, \Gamma, \Delta)$ over a visibly pushdown alphabet $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. The finite set $Q$ is defined as the set of states of $\mathcal{A}$, $Q_I \subseteq Q$ is the set of input states and $Q_F \subseteq Q$ of final states. The finite set $\Gamma$ is the stack alphabet of $\mathcal{A}$ and must not contain the special symbol $\perp$. $\Delta$ is the set of its transitions and is further divided into three distinct sets $\Delta_{call}$, $\Delta_{ret}$, $\Delta_{loc}$ such as $\Delta_{call} \subseteq (Q \times \Sigma_{call} \times Q \times \Gamma)$ is the set of call transitions, $\Delta_{ret} \subseteq (Q \times \Sigma_{ret} \times \Gamma \cup \{\perp\} \times Q)$ is the set of return transitions, and $\Delta_{loc} \subseteq (Q \times \Sigma_{loc} \times Q)$ is the set of local actions.

[1]We will indifferently name the element of $\Sigma_{call}$ *call* or *push* symbols, as well as *return* or *pop* symbols for element of $\Sigma_{ret}$, and *local actions* or *internal* symbols for element of the set $\Sigma_{loc}$.

As one can notice, the stack alphabet cannot contain the special symbol $\perp$, as this symbol will be used to identify the empty stack. Transitions of a VPA have a different signification depending whether they are call, return or local transitions. A call transition $(q, c, q', \gamma) \in (Q \times \Sigma_{call} \times Q \times \Gamma)$, will push $\gamma \neq \perp$ on reading the letter $c$ when the automaton is in state $q$, i.e. the new top of the stack is $\gamma$ (whatever was the content of the stack before) and the updated control state is $q'$. Similarly, a return transition $(q, r, \gamma, q') \in (Q \times \Sigma_{ret} \times \Gamma \cup \{\perp\} \times Q)$ on reading $r$ in state $q$, will check if the top of stack is $\gamma$, pop the symbol (except in the case $\gamma = \perp$ where nothing is popped) and finally change the control state to $q'$. A local transition $(q, l, q') \in (Q \times \Sigma_{loc} \times Q)$ will just change the control state from $q$ to $q'$ on reading $l$, and the stack remains unchanged. Note that if a VPA contains only local transitions, then it is a DFA working over $\Sigma_{loc}$.

**Remark** It is worth noting it is indeed the alphabet that is **visible**, and especially not the stack as a VPA can only look at the content of the stack on return symbols.

A **configuration** of a VPA $\mathcal{A}$ is a tuple $(q, \sigma)$ where $q \in Q$ and $\sigma \in \perp\Gamma^*$. We note $(q, \sigma) \xrightarrow{a}_{\mathcal{A}} (q', \sigma')$ if one of the following conditions holds:

**[Push]** $a \in \Sigma_{call}$ and $\exists (q, a, q', \gamma) \in \Delta_{call}$ such that $\sigma' = \sigma\gamma$

**[Pop]** $a \in \Sigma_{ret}$, $\gamma \neq \perp$ and $\exists (q, a, \gamma, q') \in \Delta_{ret}$ such that $\sigma = \sigma'\gamma$
$a \in \Sigma_{ret}$, $\gamma = \perp$ and $\exists (q, a, \gamma, q') \in \Delta_{ret}$ such that $\sigma = \sigma' = \perp$

**[Local]** $a \in \Sigma_{loc}$ and $\exists (q, a, q') \in \Delta_{loc}$ and $\sigma = \sigma'$

We define $\longrightarrow_{\mathcal{A}}$ as the reflexive and transitive closure of $\rightarrow_{\mathcal{A}}$. For every word $w \in \Sigma^*$, a **run** $\rho$ of $\mathcal{A}$ on $w$ is a finite sequence $(q_1, \sigma_1), (q_2, \sigma_2), ..., (q_{|w|}, \sigma_{|w|})$ with $(q_1, \sigma_1) \in Q_I \times \{\perp\}$, which satisfies

$$\forall 1 \leq i < |w|, (q_i, \sigma_i) \xrightarrow{w[i]}_{\mathcal{A}} (q_{i+1}, \sigma_{i+1})$$

A run $\rho$ is accepting if $q_{|\rho|} \in Q_F$. A word $w \in \Sigma^*$ is **accepted** by $\mathcal{A}$ if there exists an accepting run on $w$, i.e. there exist $q_0 \in Q_I$, $q_f \in Q_F$ and $\sigma \in \perp\Gamma^*$ such that $(q_0, \perp) \overset{w}{\longrightarrow}_{\mathcal{A}} (q_f, \sigma)$. Remark that there is absolutely no conditions on the final stack content for a word to be accepted.

**Remark** When a well-matched word is accepted by a VPA $\mathcal{A}$, the automaton always finishes with an empty stack and it has not used any transitions of the form $(q, r, \perp, q')$ during its execution.

In the following chapters, we will write $|\mathcal{A}|$ for the number of states of $\mathcal{A}$ (i.e. $|Q|$), and $||\mathcal{A}||$ for the global size of $\mathcal{A}$ (i.e. $|Q| + |\Delta|$) (we shall see in Section 3.1, page 17, why the stack alphabet size is not a relevant parameter of the automaton).

Finally, we note $\mathcal{L}(\mathcal{A})$ for the language **accepted** by $\mathcal{A}$, i.e. the set of words $w \in \Sigma^*$ such that $w$ is accepted by $\mathcal{A}$. Two VPA are said to be equivalent if they recognize the same language. We are now ready to define the class of VPL:

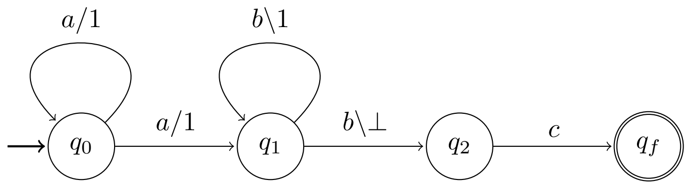

Figure 2.1: Illustration of a VPA recognizing $L = \{a^n.b^{n+1}.c \mid n \in \mathbb{N}^+\}$ over $\hat{\Sigma} = (\{a\}, \{b\}, \{c\})$. The states are depicted as circles and labelled with their respective name. Call transitions are labelled with $c/\gamma$ where $c$ is a call symbol and $\gamma$ the stack symbol pushed on top of the stack. Similarly, return transitions are labelled $r\backslash\gamma$ where $r$ is a return symbol and $\gamma$ the stack symbol that should be popped. Finally, local transitions are just labelled with the input letter. Initial states have leading arrows, and final states are double circled. All transitions not drawn are assumed to lead to a sink state (also not drawn).

**Definition** A language $L \subseteq \Sigma^*$ is a visibly pushdown language over $\hat{\Sigma}$ if there exists a VPA $\mathcal{A}$ over $\hat{\Sigma}$ such that $L = \mathcal{L}(\mathcal{A})$.

### 2.1.2 Example

Now that all preliminary definitions are set, we can give a rather simple example of such automaton. This example will serve also to introduce our drawing conventions that will be used through the following chapters, since the specificity of VPA permits to make meaningful illustrations. Consider the following VPL $L = \{a^n.b^{n+1}.c \mid n \in \mathbb{N}^+\}$ over the visibly pushdown alphabet $\hat{\Sigma} = (\{a\}, \{b\}, \{c\})$.

We can define $\mathcal{A} = (\{q_0, q_1, q_2, q_f\}, \{q_0\}, \{q_f\}, \{1\}, \Delta = \Delta_{call} \cup \Delta_{ret} \cup \Delta_{loc})$ such that $\Delta_{call} = \{(q_0, a, q_0, 1), (q_0, a, q_1, 1)\}$, $\Delta_{ret} = \{(q_1, b, 1, q_1), (q_1, b, \perp, q_2)\}$ and $\Delta_{loc} = \{(q_2, c, q_f)\}$. $\mathcal{A}$ is depicted in Figure 2.1 and one can notice it is indeed a VPA by using the definition. One can also check without much effort that $\mathcal{A}$ recognizes exactly $L$.

### 2.1.3 Determinism

Like finite automata, a VPA can either be deterministic or non-deterministic depending on its transition set. A VPA is said to be *deterministic*[2] when there is no ambiguity to chose the next state given a particular configuration and an input letter. Formally, a VPA $\mathcal{A} = (Q, Q_I, Q_F, \Gamma, \Delta)$ is deterministic if $|Q_I| = 1$ and $\forall q \in Q$:

- $\forall c \in \Sigma_{call}$, there is at most one $q' \in Q$ and one $\gamma \in \Gamma$ s.t. $(q, c, q', \gamma) \in \Delta_{call}$

[2]Since we will mostly use deterministic automata, if no information is given a VPA is assumed to be deterministic (sometimes explicitly noted DVPA). Otherwise, we will write NVPA to refer to non-deterministic VPA.

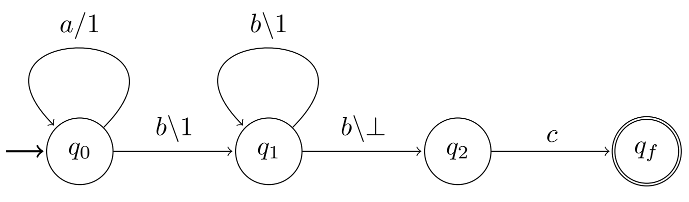

Figure 2.2: A deterministic VPA recognizing $L = \{a^n.b^{n+1}.c \mid n \in \mathbb{N}^+\}$ over $\hat{\Sigma} = (\{a\}, \{b\}, \{c\})$, the same language used in Figure 2.1.

- $\forall r \in \Sigma_{ret}, \forall \gamma \in \Gamma \cup \{\bot\}$, there is at most one $q'$ s.t. $(q, r, \gamma, q') \in \Delta_{ret}$
- $\forall l \in \Sigma_{loc}$, there is at most one $q'$ s.t. $(q, l, q') \in \Delta_{loc}$

Note that the automaton of Figure 2.1 was *non-deterministic* (NVPA), since two transitions labelled $a$ were outgoing from the state $q_0$. A deterministic automaton for this language can easily be built by changing the transition $(q_0, a, q_1, 1)$ to $(q_0, b, 1, q_1)$ (this deterministic automaton is illustrated in Figure 2.2). A typical question about automaton models is whether there exists an algorithmic way to determinize an automaton, i.e. can we build a deterministic automaton from any non-deterministic ones. For instance such an algorithm does not exist for traditional pushdown automata since deterministic ones are strictly less expressive than non-deterministic ones. The determinization of VPA appears to be computable (so both DVPA and NVPA recognize the same class of languages) and a deterministic automaton with $2^{n^2}$ states and a stack alphabet of size $2^{n^2} \cdot |\Sigma_{call}|$ can be computed from any NVPA of size $n$.

**Theorem 2.1.1** *For every* NVPA $\mathcal{A}$*, there exists an equivalent deterministic* VPA $\mathcal{D}$*. Moreover, we can build* $\mathcal{D}$ *such that* $|\mathcal{D}| \leq 2^{|\mathcal{A}|^2}$.

**Proof** Let $\mathcal{A} = (Q, Q_I, Q_F, \Gamma, \Delta)$ be a non-deterministic VPA over some visibly pushdown alphabet $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. We will build a deterministic VPA $\mathcal{D}$ which uses states from the set $2^{Q \times Q}$. The idea is to do some kind of powerset construction from the original automaton $\mathcal{A}$ and postpone push-transitions until a corresponding pop-transition is done.

For instance, a state $S$ of $\mathcal{D}$ is a set of pairs of states from $\mathcal{A}$. After reading a word $w = w_1.c.w_2$, with $w, w_1 \in \Sigma^*$, $c \in \Sigma_{call}$ and $w_2 \in WM(\hat{\Sigma})$, for each pair[3] $(q, q') \in S$, $q$ is a state reachable after reading the last unmatched call symbol $c$ (note when reading $c$, we did not know which stack symbol to put on the stack) and $q'$ is a state reachable from $q$ after reading $w_2$. Whenever a return symbol $r$ is read, let us say just after reading $w_2$, we will unstack the last call symbol $c$ and $S'$, the state reached by $\mathcal{D}$ before reading $c$. Now we can compute the new state $S''$ of $\mathcal{D}$ after reading $w = w_1.c.w_2.r$, as we can compute what call

[3]Such a pair is usually named a **summary edge** of the automaton as it indicates a summary of possible sequences of transitions in the original automaton.

symbols could have been pushed when reading $c$ (using $r$) and what transitions could have been used until reading $r$.

In order to define formally $\mathcal{D}$, we define the identity set $Id_Q$ for some set $Q$ as $Id_Q = \{(q,q) \mid q \in Q\}$ (remark $Id_Q \in 2^{Q\times Q}$) and the projection function $\Pi_2 : 2^{Q\times Q} \to 2^Q$ for some set $Q$ as $\Pi_2(S) = \{q' \in Q \mid \exists q \in Q, (q,q') \in S\}$. Similarly $\Pi_1$ is defined to return the set of first arguments of some set $S$.

We construct now the following deterministic VPA $\mathcal{D} = (Q', q'_i, Q'_F, \Gamma', \Delta')$ such that $Q' = 2^{Q\times Q}$, $q'_i = Id_{Q_I}$, $Q'_F = \{S \in Q' \mid \Pi_2(S) \cap Q_F \neq \emptyset\}$, $\Gamma' = Q' \times \Sigma_{call}$, and the transition set is as follows:

**[Push]** For every $c \in \Sigma_{call}$, $(S, c, Id_{S'}, (S,c)) \in \Delta'_{call}$
$\Leftrightarrow S' = \{q' \in Q \mid \exists q \in Q, \gamma \in \Gamma, q \in \Pi_2(S) \wedge (q,c,q',\gamma) \in \Delta_{call}\}$

**[Pop]** For every $r \in \Sigma_{ret}$,

– $(S, r, \bot, S') \in \Delta'_{ret}$
$\Leftrightarrow S' = \{(q,q'') \in Q^2 \mid \exists q' \in Q, (q,q') \in S \wedge (q', r, \bot, q'') \in \Delta_{loc}\}$

– $(S, r, (S'', c), S') \in \Delta'_{ret}$
$\Leftrightarrow S' = \{(q,q') \in Q^2 \mid \exists q_1, q_2, q_3 \in Q, \gamma \in \Gamma, (q,q_1) \in S'' \wedge (q_1, c, q_2, \gamma) \in \Delta_{call} \wedge (q_2,q_3) \in S \wedge (q_3, r, \gamma, q') \in \Delta_{ret}\}$

**[Local]** For every $l \in \Sigma_{loc}$, $(S, l, S') \in \Delta'_{loc}$
$\Leftrightarrow S' = \{(q,q'') \in Q^2 \mid \exists q' \in Q, (q,q') \in S \wedge (q', l, q'') \in \Delta_{loc}\}$

Let $w = w_0.c_1.w_1.c_2.w_2...c_n.w_n$ a word[4] of $\Sigma^*$ such that $w_0 \in MC(\hat{\Sigma})$, $c_1, c_2, ..., c_n \in \Sigma_{call}$, and $w_1, w_2, ..., w_{n-1}, w_n \in WM(\hat{\Sigma})$. After reading such a word $w$, we maintain as an invariant that the configuration of $\mathcal{D}$ is $(S, \sigma)$ s.t.:

- $\Pi_2(S)$ is the set of reachable states in $\mathcal{A}$ after reading $w$ and $\Pi_1(S)$ is the set of reachable states in $\mathcal{A}$ after reading $w'$, where $w = w'w_n$.
- $\sigma = \bot(S_1, c_1), ..., (S_n, c_n)$ such that $\forall 1 \leq i \leq n$, $S_i$ is the set of pairs $(q,q')$ such that $q$ is reachable in $\mathcal{A}$ after reading $w' = w_0c_1...c_{i-1}$ and $q'$ is reachable in $\mathcal{A}$ after reading $w'w_{i-1}$.

At last, remark the initial state $q'_i$ of $\mathcal{D}$ satisfies the above property if you set $w = \varepsilon$, we have $\Pi_1(q'_i) = \Pi_2(q'_i) = Q_I$ (i.e. exactly the set of states reachable after reading $\varepsilon$) and the stack is $\bot$. Such an invariant can be easily checked. □

The above proof is based on the construction introduced in [13], which represents a slight improvement on the original proof from [2]. Moreover, we know that a NVPA can be exponentially smaller than any equivalent DVPA, since when we restrict the alphabet to local actions, VPA are equivalent to DFA and it is a well known result that there is an exponential blow-up between DFA and NFA (Non-deterministic Finite Automata). Also we know that $2^{n^2}$ is a lower bound of the minimal deterministic VPA for some NVPA of size $n$, as claimed in [12].

[4]Note every word of $\Sigma^*$ can be decomposed this way.

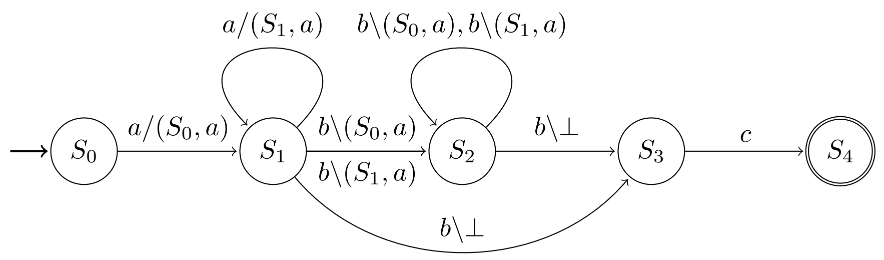
$$\begin{tikzcd}
\arrow[r] & S_0 \arrow[r, "a/(S_0,a)"] & S_1 \arrow[loop above, "a/(S_1,a)"] \arrow[r, "b\backslash(S_0,a)", "b\backslash(S_1,a)"'] \arrow[rr, bend right, "b\backslash\perp"'] & S_2 \arrow[loop above, "{b\backslash(S_0,a), b\backslash(S_1,a)}"] \arrow[r, "b\backslash\perp"] & S_3 \arrow[r, "c"] & S_4
\end{tikzcd}$$

Figure 2.3: A deterministic VPA built using the proof of Theorem 2.1.1 from the NVPA of Figure 2.1 recognizing $L = \{a^n.b^{n+1}.c \mid n \in \mathbb{N}^+\}$. We have $S_0 = \{(q_0, q_0)\}, S_1 = \{(q_0, q_0), (q_1, q_1)\}, S_2 = \{(q_1, q_1)\}, S_3 = \{(q_1, q_2)\}, S_4 = \{(q_1, q_f)\}$.

To conclude this section let us illustrate the determinization construction by an example. Let $\mathcal{A}$ be the NVPA depicted in Figure 2.1 recognizing the VPL $L = \{a^n.b^{n+1}.c \mid n \in \mathbb{N}^+\}$ over $\hat{\Sigma} = (\{a\}, \{b\}, \{c\})$. The determinization results into the automaton depicted in Figure 2.3 which has 5 states.

Remark the construction produces some unusable transitions: for instance $(S_1, b, \perp, S_3)$ is unnecessary as $(S_1, \perp)$ is an unreachable configuration of the deterministic automaton. Furthermore, the automaton built is also not necessary a minimum-state VPA since the DVPA of Figure 2.2 recognizes the same language and has one state less.

## 2.2 Closure properties

We have already stated that the class of VPA is closed under union, intersection, complementation, renaming, concatenation and Kleene-$*$. These good closure properties make VPA a robust class comparable to *Regular* languages (see Table 2.1 for details) and much more robust than the class of (Deterministic) Context-Free Languages (DCFL and CFL). We will use explicitly later on the fact that we can compute efficiently the intersection of two VPA as well as the complement of a DVPA, so we will explain here how this can be done. We guide the reader toward [2] for details about the closure of the other applications.

**Proposition 2.2.1** *Let $\mathcal{A}$ and $\mathcal{B}$ be two* VPA *over $\hat{\Sigma}$. We can build in quadratic time the automaton $\mathcal{C}$ recognizing the language $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$.*

**Proof** Let $\mathcal{A} = (Q^{\mathcal{A}}, Q_I^{\mathcal{A}}, Q_F^{\mathcal{A}}, \Gamma^{\mathcal{A}}, \Delta^{\mathcal{A}})$ and $\mathcal{B} = (Q^{\mathcal{B}}, Q_I^{\mathcal{B}}, Q_F^{\mathcal{B}}, \Gamma^{\mathcal{B}}, \Delta^{\mathcal{B}})$ be two VPA over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. Let $\mathcal{C} = (Q^{\mathcal{A}} \times Q^{\mathcal{B}}, Q_I^{\mathcal{A}} \times Q_I^{\mathcal{B}}, Q_F^{\mathcal{A}} \times Q_F^{\mathcal{B}}, \Gamma^{\mathcal{A}} \times \Gamma^{\mathcal{B}}, \Delta^{\mathcal{C}})$ be a VPA, such that :

-$((q_1, q_2), c, (q_1', q_2'), (\gamma_1, \gamma_2)) \in \Delta_c^{\mathcal{C}} \Leftrightarrow (q_1, c, q_1', \gamma_1) \in \Delta_c^{\mathcal{A}} \wedge (q_2, c, q_2', \gamma_2) \in \Delta_c^{\mathcal{B}}$
-$((q_1, q_2), r, (\gamma_1, \gamma_2), (q_1', q_2')) \in \Delta_r^{\mathcal{C}} \Leftrightarrow (q_1, r, \gamma_1, q_1') \in \Delta_r^{\mathcal{A}} \wedge (q_2, r, \gamma_2, q_2') \in \Delta_r^{\mathcal{B}}$
-$((q_1, q_2), l, (q_1', q_2')) \in \Delta_l^{\mathcal{C}} \Leftrightarrow (q_1, l, q_1') \in \Delta_l^{\mathcal{A}} \wedge (q_2, l, q_2') \in \Delta_l^{\mathcal{B}}$

|  | Closure under |  |  |  |  |
|---|---|---|---|---|---|
|  | $\cup$ | $\cap$ | Complement | Concatenation | Kleene-$*$ |
| Regular | ✓ | ✓ | ✓ | ✓ | ✓ |
| VPL | ✓ | ✓ | ✓ | ✓ | ✓ |
| DCFL | × | × | ✓ | × | × |
| CFL | ✓ | × | × | ✓ | ✓ |

Table 2.1: Closure Properties for VPL and their surrounding classes in the Chomsky hierarchy.

One can notice $\mathcal{C}$ accepts a word $w$ if and only if both $\mathcal{A}$ and $\mathcal{B}$ accept $w$, and thus $\mathcal{C}$ recognizes exactly $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$.

Note there is a simple algorithm in order to build $\mathcal{C}$ by running through all transitions of $\mathcal{A}$ and $\mathcal{B}$, which one takes $\mathcal{O}(||A|| \cdot ||B||)$ time and obviously $||A|| \cdot ||B|| \leq (||A|| + ||B||)^2$. □

**Proposition 2.2.2** *Let $\mathcal{A}$ be a* DVPA *over $\hat{\Sigma}$. We can build in linear time a* DVPA *$\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$.*

**Proof** Given a DVPA $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta)$, one can simply build a DVPA $\mathcal{B} = (Q, q_i, Q \setminus Q_F, \Gamma, \Delta)$. Obviously, the set of words recognized by $\mathcal{B}$ is exactly the set of words that are rejected by $\mathcal{A}$. □

Please note that the above proposition does not hold for NVPA, as the acceptance condition for NVPA is an existence condition. The complementation of NVPA can be computed though by first determinizing them (see Theorem 2.1.1), before computing their complement.

**Theorem 2.2.3 ([2])** *The class of* VPL *is closed under union, intersection, complementation, concatenation and Kleene-$*$.*

## 2.3 Decision problems

In the previous section, we have seen that the class of VPL is closed under all boolean set operations and usual language operations such as concatenation and Kleene-$*$. But the VPA is also a powerful model when dealing with decision problems. We know for instance that most decision problems are very easy to solve for DFA (Table 2.2 sums up the main decision problems for models of automata that are related to VPA). If we add a little non-determinism to DFA, then decision problems start to become more difficult (for instance NFA and Unambiguous NFA or UFA). Even if decision problems for VPA lie in an upper complexity class (i.e. EXPTIME), they stay decidable unlike inclusion for DPDA and both universality/equivalence and inclusion for PDA. We refer to [2] for details about the proof of hardness of the different decision problems for VPA.

However, when we consider only deterministic VPA, the decision problems become rather easy to solve as stated by the following theorem:

| | Decision problems | | |
|---|---|---|---|
| | Emptiness | Universality and Equivalence | Inclusion |
| DFA | NLOGSPACE | NLOGSPACE | NLOGSPACE |
| UFA | NLOGSPACE | PTIME | PTIME |
| NFA | NLOGSPACE | PSPACE | PSPACE |
| DVPA | PTIME | PTIME | PTIME |
| VPA | PTIME | EXPTIME | EXPTIME |
| DPDA | PTIME | Decidable | Undecidable |
| PDA | PTIME | Undecidable | Undecidable |

Table 2.2: Main decision problems for VPA and some close automaton models.

**Theorem 2.3.1** *Let $\mathcal{A}$ and $\mathcal{B}$ be two* DVPA *over $\hat{\Sigma}$. We can decide in polynomial time whether $\mathcal{L}(\mathcal{A}) = \emptyset$, $\mathcal{L}(\mathcal{A}) = \Sigma^*$, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ and $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.*

**Proof** Let $\mathcal{A}$, $\mathcal{B}$ be two DVPA over $\hat{\Sigma}$.

As VPA are also PDA, we can decide in time $\mathcal{O}(|\mathcal{A}|^3)$ if $\mathcal{L}(\mathcal{A}) = \emptyset$.

We can decide also easily if $\mathcal{L}(\mathcal{A}) = \Sigma^*$, by checking if $\mathcal{L}(\mathcal{A}^c) = \emptyset$, where $\mathcal{A}^c$ is the complement of $\mathcal{A}$ (obtained using Proposition 2.2.2).

Moreover, using Propositions 2.2.2 and 2.2.1 we can compute a VPA $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}^c)$. Since, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \Leftrightarrow \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}^c) = \emptyset$, we can also decide in $\mathcal{O}(|\mathcal{C}|^3)$ time (i.e. $\mathcal{O}(n^6)$, with $n = max(||A||, ||B||)$) whether the language of $\mathcal{A}$ is included in the language of $\mathcal{B}$ or not.

Finally, the equivalence is obtained by testing if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ and $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$.

□

Hence, the class of VPL can be considered tractable when we compare it to other classes of stack automata. At last Theorem 2.3.1 implies that the determinization problem is also EXPTIME-hard since equivalence, universality and inclusion are PTIME on deterministic input but EXPTIME-complete otherwise.

## 2.4 Applications

Let us conclude this introductory chapter by giving an idea on how to use the VPA model in applications. In the introduction, we cite several domains where VPA appeared and present an algorithmic solution to various program analysis questions. We will describe in further details here two of their applications, on boolean programs and on semantics of programming languages.

### Boolean Programs

One of the diverse applications of VPA is checking properties on **boolean programs** (that is programs where all variables have finite types)[5].

[5]This application is initially described in [2].

Boolean programs can be obtained by using *predicate abstraction* on general software. They are usually constructed by over-approximating the behaviour of a program and several tools already exist to translate general code into boolean programs (e.g. SLAM and SATABS).

Let $P$ denote a boolean program with procedures that can call each other. We can easily find a pushdown alphabet $(\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ that maps every call of $P$ to symbols of $\Sigma_{call}$, returns to $\Sigma_{ret}$ and other statements to $\Sigma_{loc}$. Then a generator for a visibly pushdown language $L(P)$ can be built by constructing a VPA that acts as $P$ does. We can check that $P$ satisfies some specification by checking $L(P) \subseteq S$, where $S$ is a specification given as a VPL (recall inclusion is decidable in polynomial time for deterministic VPA and exponential time for non-deterministic ones, cf Table 2.2).

In this manner, we can naturally verify all regular properties but also non-regular ones like:

**Partial correctness:** If some property $p$ holds when a procedure $P$ is invoked, then if $P$ returns, some property $q$ must hold thereafter.

**Total correctness:** Same as partial correctness but $P$ is also required to always return.

**Local properties:** In a procedure $P$, if we skip all sub-computations due to methods invocation, then some regular property $q$ must hold.

**Access control:** We can invoke a procedure $P$ only from a procedure $Q$.

**Interrupt stack limits:** A property $p$ must hold whenever the number of interrupts in the stack is under a given constant.

Note that these requirements can generally be verified faster when we have a minimal automaton that accepts the VPL $S$ given by the specification. This is an important motivation to investigate the problem of minimization, treated in the next chapter.

### 2.4.1 Semantics of programs

In [11], the authors construct VPA that describe the semantics of a program expression given in *Idealized Algol*, a language introduced by Reynolds in order to synthesize functional and imperative languages.

Then, the VPA are used to check program expression equivalence (remark testing equivalence of VPA is as hard as inclusion, see Table 2.2). Their construction is based on building intermediate automata in order to translate a program expression. If we can achieve efficiently the minimization of VPA, the feasibility of their construction will be greatly improved.

In other contexts, such as XML streaming [4, 5] and model-checking [9], a minimization algorithm for VPA would be also highly appreciated. This is another motivation in order to focus on this particular problem.

# 3 Minimization and Congruences

In the previous section, we have set the model of VPA and stated why this model presents both the advantage of being tractable and robust but still preserve enough expressiveness to solve interesting verification problems. Nevertheless there are still some open questions about VPA that could have great impacts on potential applications.

In automata theory, a well known question is the minimization problem, which asks whether a given automaton is the minimal (usually regarding its number of states) automaton that recognizes a language. The functional variant of this problem is to find such a minimal automaton, usually given as input an ordinary automaton that recognizes the desired language.

Two standard questions arise from the minimization problem: first, we can wonder if a unique model exists for a given language (i.e. a canonical model for a formal language) which one guarantees an efficient algorithm for checking equivalence of two automata; second, a minimal automaton model for a given language can be desired, which offers the best memory consumption for a given language, as well as providing fewer computation for all operations depending of the size of an automaton. Regular languages have the good property that for each language one can find a unique minimal DFA, that answers simultaneously both of the two aforementioned questions.

Concerning VPA, we know that there does not exist a unique minimal model for a given VPL [8]. However, several canonical candidates have been introduced (through regular tree automata [2], congruences [8], or variants of VPA [14]), but still all these canonical model have been shown to not be minimal and furthermore to potentially lead to an exponential blow-up in comparison to the minimal automaton (see e.g. [14]). Since no polynomial approximation within polynomial time is known, the problem is assumed to be hard to solve [14].

We will first describe in Section 3.1 the minimization problem and especially what are the parameters that matter the most in the case of VPA. We shall focus in Section 3.2 on the minimization of non-deterministic VPA. Then in Section 3.3, we will investigate the minimization problem for DVPA, show that the minimal automaton is not canonical and give some arguments for a hardness result. At last, we will conclude by describing the standard congruences on VPA which lead directly to a canonical model for any VPL in Section 3.4 and show why these congruences are not well suited for minimization.

## 3.1 The minimization problem

The minimization of a VPA will lead to a minimal model, but still we have first to decide what should be the input and the output of the problem, i.e. what is the most useful parameter of a VPA that should be minimized.

First, we are more inclined to use as input and output only deterministic VPA. The reason behind this choice is due to the complexity of non-deterministic automata. In fact, even for the simplest abstract machine model e.g. finite state automata, minimization of non-deterministic models is hard. Precisely, when the minimization of DFA is seen as a simple problem (i.e. the problem is in NLOGSPACE) the minimization of NFA is undoubtedly a harder problem (i.e. the problem has been shown to be PSPACE-complete, see [15]). Thus even if non-deterministic automata offer potentially exponentially smaller models, their minimization is hard enough that it is not practically realizable in many cases. We show indeed that minimization of non-deterministic VPA is EXPTIME-complete in Section 3.2. When nothing is precised we will assume in the rest of this chapter that we are only manipulating deterministic automata.

Now we can wonder what parameters have to be minimize, i.e. in a VPA $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta)$, there are three parameters providing some kind of size: the number of states $|Q|$, the size of the stack alphabet $|\Gamma|$ and the number of transitions $|\Delta|$. Note the alphabet on which $\mathcal{A}$ works, has almost no importance with regards to its size, and thus we will usually consider that the size of the input alphabet is a constant. This is a safe assumption as anyway the minimal automaton for a given visibly pushdown language has to share the input alphabet with any other VPA recognizing the same language. We will show in this section why $|Q|$ is the most relevant parameter of a VPA, and therefore it is indeed very profitable to minimize it.

Traditionally in formal language theory, it is often the number of states of an automaton model that is intended to be minimize. This is due to several reasons: first the number of transitions is bounded by some function of the number of states, second the number of states is more often used as a complexity bound for typical problems on the automaton model, and third minimization of transitions is a harder problem. In fact, like the minimization of non-deterministic automata, the minimization of transitions is already a hard problem for DFA. Mainly for this reason and since it is bounded by some functions of the number of states (and both the stack alphabet and the input alphabet for VPA), we will not consider here any kind of minimization on the number of transitions.

So, we can still want to minimize the stack alphabet, the number of states or some wise combinations of both parameters. Investigating which is the most relevant parameter for measure complexity of deterministic pushdown automata has already been studied, and recently [16] shows that there does not exist any combination of the number of states and the size of the alphabet that is suitable for DPDA. However, VPA has the particularity that the stack alphabet is bounded by the number of states times the push alphabet (noticed by [14]), and thus it is of lesser importance than the state set size. We will show also that a similar claim can be maid for NVPA.

## A uniform stack

Let us prove in this section our last claim about the necessity of a stack alphabet. Afterwards, we will see what are the consequences of such a statement over the other measurement parameters of minimal Vpa.

**Proposition 3.1.1** *Let $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta)$ be a* Vpa *over $(\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. Then there exists an equivalent* Vpa *$\mathcal{A}' = (Q, q_i, Q_F, \Gamma', \Delta')$ with $\Delta' \subseteq \Delta$ and $|\Gamma'| \leq |Q| \cdot |\Sigma_{call}|$.*

**Proof** Since $\mathcal{A}$ is deterministic there are at most $|Q| \cdot |\Sigma_{call}|$ call transitions (the control state and the input letter determine directly both the target state and the stack symbol). In the worst case, each such transition use a different push symbol, and we can set $\Delta' \subseteq \Delta$ in order to contain only return transitions that pop one of these symbols, thus we get $|\Gamma'| \leq |Q| \cdot |\Sigma_{call}|$. □

**Proposition 3.1.2** *Let $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta)$ be a* Vpa *over $(\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. Then we can build in linear time an equivalent* Vpa *with stack alphabet $Q \times \Sigma_{call}$.*

**Proof** Let $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta^{\mathcal{A}})$ be a Vpa over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. We construct an equivalent Vpa $\mathcal{B} = (Q, q_i, Q_F, Q \times \Sigma_{call}, \Delta^{\mathcal{B}})$. Note $|\mathcal{A}| = |\mathcal{B}|$. We first set $\Delta^{\mathcal{B}}_l = \Delta^{\mathcal{A}}_l$, then $\Delta^{\mathcal{B}}_c = \{(q, c, q', (q, c)) | \exists \gamma \in \Gamma, (q, c, q', \gamma) \in \Delta^{\mathcal{A}}_c\}$.

Now for each $\gamma \in \Gamma$ let $\alpha_\gamma = \{(q, c) | \exists q' \in Q, (q, c, q', \gamma) \in \Delta^{\mathcal{A}}_c\}$. Then we can define $\Delta^{\mathcal{B}}_r = \{(q, r, (q'', c), q') | \exists \gamma \in \Gamma, (q'', c) \in \alpha_\gamma \wedge (q, r, \gamma, q') \in \Delta^{\mathcal{A}}_r\}$.

One can notice that since $\mathcal{A}$ is deterministic and each $\alpha_\gamma$ is disjoint from the others, $\mathcal{B}$ is also deterministic.

Let us finally check that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. Consider a run $(q_1, \sigma_1), ..., (q_{|w|}, \sigma_{|w|})$ for $\mathcal{A}$ on a word $w \in \Sigma^*$, and similarly another run $(q'_1, \sigma'_1), ..., (q'_{|w|}, \sigma'_{|w|})$ for $\mathcal{B}$ on the same word.

By induction let us prove that $\forall 1 \leq i \leq |w|$, $q_i = q'_i$. Obviously $(q_1, \sigma_1) = (q'_1, \sigma'_1) = (q_i, \perp)$. Assume $\forall 1 \leq i < k$, $q_i = q'_i$ and consider the transition used for $w[k]$.

If $w[k] \in \Sigma_{loc}$, then $(q_k, w[k], q_{k+1}) \in \Delta^{\mathcal{A}}_l$ and $(q'_k, w[k], q'_{k+1}) \in \Delta^{\mathcal{B}}_l$ and as $q_k = q'_k$ and $\Delta^{\mathcal{B}}_l = \Delta^{\mathcal{A}}_l$, $q_{k+1} = q'_{k+1}$.

Similarly, if $w[k] \in \Sigma_{call}$ then $q_{k+1} = q'_{k+1}$ as the stack content is also not examined.

If $w[k] \in \Sigma_{ret}$, then let $\gamma$ denote the top of the stack $\sigma_k$, i.e. $\sigma_k = \sigma_{k+1}\gamma$ if $\sigma_k \neq \perp$ or $\sigma_k = \sigma_{k+1} = \gamma = \perp$ otherwise. In the latter case, then trivially $q_{k+1} = q'_{k+1}$ for the same arguments as above. Otherwise, let $j$ be the step when $\gamma$ has been pushed. We have $(q_j, \sigma_j) \overset{w[j]}{\rightarrow}_{\mathcal{A}} (q_{j+1}, \sigma_{j+1})$ with $\sigma_{j+1} = \sigma_j \gamma$, and in the same fashion we have $(q'_j, \sigma'_j) \overset{w[j]}{\rightarrow}_{\mathcal{B}} (q'_{j+1}, \sigma'_{j+1})$ with $\sigma'_{j+1} = \sigma'_j(q'_j, w[j])$. So we have $(q'_j, w[j]) \in \alpha_\gamma$ and moreover by definition there exists $(q'_k, r, (q'_j, w[j]), q'_{k+1}) \in \Delta^{\mathcal{B}}_r$ iff $(q_k, r, \gamma, q_{k+1}) \in \Delta^{\mathcal{A}}_r$ with $q'_{k+1} = q_{k+1}$.

Eventually every run is leading to the same states, and therefore $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. □

Assuming the input alphabet is a constant, we can find polynomial approximation of the number of transitions using only the minimum number of states as the following proposition states:

**Proposition 3.1.3** *Let $L$ be a* VPL *over $(\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$, $\mathcal{A}_{min}$ a minimum-state* VPA *recognizing $L$ and $m$ the minimum number of transitions that a* VPA *needs to recognize $L$. Then $|\mathcal{A}_{min}| \leq m \leq \mathcal{O}(|\mathcal{A}_{min}|^2)$.*

**Proof** Let $\mathcal{A}_{min} = (Q, q_i, Q_F, \Gamma, \Delta)$ be a minimum-state VPA that recognizes $L$. We can assume $\mathcal{A}_{min}$ to use only stack symbol from $Q \times \Sigma_{call}$, using Proposition 3.1.2. We have by definition $|\Delta| \leq |Q| \cdot |\Sigma_{call}| + |Q| \cdot |\Sigma_{ret}| \cdot (|\Sigma_{call}| \cdot |Q|) + |Q| \cdot |\Sigma_{loc}|$. Hence $|\Delta| \leq 2 \cdot |\Sigma| \cdot |Q| + |Q|^2$ and as $|\Sigma|$ is considered constant, we get $|\Delta| \leq \mathcal{O}(|Q|^2)$.

Now let $\mathcal{A'}_{min} = (Q', q'_i, Q'_F, \Gamma', \Delta')$ be a minimum-transition VPA that recognizes $L$. We can assume every state is reachable, and therefore there is at least as much transitions as states in $\mathcal{A'}_{min}$, i.e. $|\Delta'| \geq |Q'|$.

Since $\mathcal{A}_{min}$ is a minimum-state VPA, we have $|Q'| \geq |Q|$, hence $|Q| \leq |Q'| \leq |\Delta'| \leq |\Delta| \leq \mathcal{O}(|Q|^2)$. Thus at worst $|\Delta'| = |Q|$, and the minimum-state VPA has only *approximately* quadratically more transitions than the minimum-transitions VPA.

□

To conclude, although the number of stack symbols can be inferred from the size of an automaton and thus a minimum-state VPA gives a VPA with relatively few stack symbols, the opposite is not always true. Indeed, the minimum-stack symbol VPA can be much bigger than the minimum-state VPA:

**Proposition 3.1.4** *There are some* VPL *$L$ where a minimum-stack symbol* VPA *has exponentially more states than the minimum-state* VPA*.*

**Proof** Let $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ with $\Sigma_{call} = \{c_a, c_b\}$, $\Sigma_{ret} = \{r_a, r_b\}$ and $\Sigma_{loc} = \emptyset$ be a visibly pushdown alphabet and let $k$ be an integer. Consider the VPL $L = \{w\widetilde{w} \mid w \in \Sigma_{call}^k, \widetilde{w} \in \Sigma_{ret}^k, \text{s.t. } \forall 1 \leq i \leq k, w[i] = c_a \Leftrightarrow \widetilde{w}[2 \cdot k - i + 1] = r_a\}$, i.e. $L$ contains all the words of length $2 \cdot k$ where the second half is the mirror of the first half, considering only subscript letters $a$ and $b$.

It is possible to build a VPA using only 1 stack symbol as $L$ is indeed a regular language. If we use only one stack symbol, after reading $w$ with $|w| = k$ the only information that is on the stack is the number of calls read and it is useless in the construction of the VPA. Since we have to know exactly what was the symbol at position $i$, we need to memorize this information in the control state of the automaton, and thus any 1-stack symbol VPA needs at least $\mathcal{O}(2^k)$ states to memorize $w$.

Now, if two stack symbols are allowed we can put on the stack exactly the first $k$ letters (either $c_a$ or $c_b$). Then we just have to read transitions that read the corresponding $r_a$ or $r_b$. In fact, the minimum-state VPA needs only $k$ states, either checking the *constant* length of the word using the call symbols or the return symbols.

□

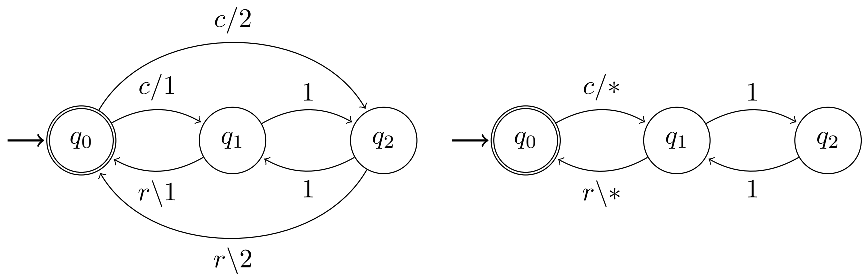

Figure 3.1: On the left a NVPA recognizing the VPL $L = \{(c(11)^*r)^*\}$ over $(\{c\}, \{r\}, \{1\})$. On the right, a VPA that recognizes the same language.

## 3.2 Minimization of Nvpa

As stated in the introduction of this chapter, the minimization of *non-deterministic* VPA is a computationally hard problem. In this section, we will prove that the minimization of NVPA is EXPTIME-complete. We need first to introduce an argument similar to the one used in the deterministic case, i.e. stack alphabet can be either bounded or replaced by a generic one without increasing the number of states of a NVPA. Note we cannot apply directly Proposition 3.1.2 in order to replace the stack symbols of NVPA, since the stack symbol can be used to remember what was the target state of a push transition. This case is illustrated by Figure 3.1, where we cannot apply the proposition since there are two outgoing transitions labelled $c$ from $q_0$. However in the deterministic case (right automaton of Figure 3.1), we can safely replace the stack symbol $*$ by $(q_0, c)$.

**Proposition 3.2.1** *For any* NVPA *over* $(\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$*, we can build in linear time an equivalent* NVPA *of same size that uses only the set* $Q \times Q \times \Sigma_{call}$ *as its stack alphabet.*

**Proof** The idea of the proof relies mainly on similar arguments used in the proof of Proposition 3.1.2, i.e. we will replace a stack symbol by a subset of $Q \times Q \times \Sigma_{call}$.

Let $\mathcal{A} = (Q, Q_I, Q_F, \Gamma, \Delta^{\mathcal{A}})$ be a NVPA over $\hat{\Sigma}$. We will build the NVPA $\mathcal{B} = (Q, Q_I, Q_F, Q \times Q \times \Sigma_{call}, \Delta^{\mathcal{B}})$ over the same pushdown alphabet in such a fashion that both automata will recognize the same language (note $\mathcal{A}$ and $\mathcal{B}$ differ only by stack alphabet and transition set). We set $\Delta_l^{\mathcal{B}} = \Delta_l^{\mathcal{A}}$, and $\Delta_c^{\mathcal{B}} = \{(q, c, q', (q, q', c)) \mid \exists \gamma \in \Gamma, (q, c, q', \gamma) \in \Delta_c^{\mathcal{A}}\}$.

Now for each $\gamma \in \Gamma$ let $\alpha_\gamma = \{(q, q', c) \mid (q, c, q', \gamma) \in \Delta_c^{\mathcal{A}}\}$. Then we define $\Delta_r^{\mathcal{B}} = \{(q, r, (q_1, q_2, c), q') \mid \exists \gamma \in \Gamma, (q, r, \gamma, q') \in \Delta_r^{\mathcal{A}} \wedge (q_1, q_2, c) \in \alpha_\gamma\}$.

In order to prove $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ we will show that for every run $(q_1, \sigma_1)$, $(q_2, \sigma_2)$, ..., $(q_n, \sigma_n)$ of $\mathcal{A}$ on $w$, there exists $\sigma'_1, \sigma'_2, ..., \sigma'_n$ such that the sequence $(q_1, \sigma'_1)$, $(q_2, \sigma'_2)$, ..., $(q_n, \sigma'_n)$ is a run of $\mathcal{B}$ over $w$ (with $q_1 \in Q_I$ and $\sigma_1 = \sigma'_1 = \perp$). Remark this shows $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ since there exists an accepting run of $\mathcal{A}$

implies there exists an accepting run of $\mathcal{B}$, and the opposite can be deduced easily.

Fix $\rho = (q_1, \sigma_1), (q_2, \sigma_2), ..., (q_n, \sigma_n)$ a run of $\mathcal{A}$ over $w$ and let us prove by induction that $\forall 1 \leq i \leq n$, there exists $\sigma'_1, \sigma'_2, ..., \sigma'_i$ s.t. $(q_1, \sigma'_1)$, $(q_2, \sigma'_2)$, ..., $(q_i, \sigma'_i)$ is a run of $\mathcal{B}$ over $w[0..i-1]$ (i.e. the beginning of a run of $\mathcal{B}$ over $w$).

The basis is obvious, $(q_1, \sigma'_1)$ is a run of $\mathcal{B}$ over $\varepsilon$ (since $q_1 \in Q_I$ and $\sigma'_1 = \bot$).

Assume the property holds until some $1 \leq k < n$, i.e. there exists $\sigma'_1, \sigma'_2, ..., \sigma'_k$ such that $\rho' = (q_1, \sigma'_1)$, $(q_2, \sigma'_2)$, ..., $(q_k, \sigma'_k)$ is a run of $\mathcal{B}$ over $w[0..k-1]$.

If $w[k] \in \Sigma_{loc}$, then $\rho'(q_{k+1}, \sigma'_{k+1})$ with $\sigma'_{k+1} = \sigma'_k$ is a run of $\mathcal{B}$ over $w[0..k]$ as $\Delta^{\mathcal{B}}_l = \Delta^{\mathcal{A}}_l$.

Similarly if $w[k] \in \Sigma_{call}$, then there exists $\gamma \in \Gamma$ s.t. $\sigma_{k+1} = \sigma_k\gamma$ as $\rho$ is a run of $\mathcal{A}$, and thus $\rho'(q_{k+1}, \sigma'_k(q_k, q_{k+1}, c))$ is a run of $\mathcal{B}$ on $w[0..k]$.

Finally consider the case where $w[k] \in \Sigma_{ret}$, we have that if $\sigma_{k+1} = \sigma_k = \bot$ then trivially $\rho'(q_{k+1}, \bot)$ is a run of $\mathcal{B}$ on $w[0..k]$. Otherwise, let $j$ denote the step of the matching call of $w[k]$, and let $c = w[j] \in \Sigma_{call}$ (note $j < k$).

By induction hypothesis, we have that $\exists \sigma'_j, \sigma'_{j+1}$ s.t. $(q_j, \sigma'_j) \xrightarrow{c}_{\mathcal{B}} (q_{j+1}, \sigma'_{j+1})$. By definition of $\mathcal{B}$, $\sigma'_{j+1} = \sigma'_j(q_j, q_{j+1}, c)$. Moreover we know that $(q_j, \sigma_j) \xrightarrow{c}_{\mathcal{A}} (q_{j+1}, \sigma_{j+1})$ s.t. $\sigma_{j+1} = \sigma_j\gamma$ with $\gamma \in \Gamma$. Thus $(q_j, q_{j+1}, c) \in \alpha_\gamma$ and we have that $(q_k, r, \gamma, q_{k+1}) \in \Delta^{\mathcal{A}}_{ret}$ since $\rho$ is a valid run of $\mathcal{A}$ on $w$. Eventually, $\rho'(q_{k+1}, \sigma'_{k+1})$ with $\sigma'_k = \sigma'_{k+1}(q_j, q_{j+1}, c)$ is a run of $\mathcal{B}$ on $w[0..k]$ as $(q_k, r, (q_j, q_{j+1}, c), q_{k+1}) \in \Delta^{\mathcal{B}}_{ret}$.

□

The last proposition is useful in many ways as it gives an idea of the role played by the stack alphabet in a visibly pushdown automaton. Also it can be used to give a formal notion of completeness of such model. In fact, by definition a NVPA does not imply any bound on its stack alphabet and thus an unbounded number of transitions can lie between any two states (most of those transitions are certainly useless or redundant). Since we can replace the stack alphabet by one that is bounded by a function of the number of states and the call alphabet, we can assume any VPA to have this property (otherwise just apply Proposition 3.2.1). Moreover the proposition says that the symbol pushed by a call transition is determined directly by the rest of the transition, i.e. called transitions can be expressed only by a triplet of the form $(q, c, q')$ with $q, q' \in Q$, $c \in \Sigma_{call}$. We are now ready to give a good definition for a complete (non-deterministic) VPA using the previous generalization on the stack:

**Definition** A VPA $\mathcal{A} = (Q, Q_I, Q_F, \Gamma, \Delta)$ is said to be **complete** if its stack alphabet is $Q \times Q \times \Sigma_{call}$ and its transition set has the following properties $\forall q \in Q$:

- $\forall c \in \Sigma_{call}, \exists q' \in Q, (q, c, q', (q, q', c)) \in \Delta_{call}$
- $\forall r \in \Sigma_{ret}, \forall \gamma \in \Gamma \cup \{\bot\}, \exists q' \in Q, (q, r, \gamma, q') \in \Delta_{ret}$
- $\forall l \in \Sigma_{loc}, \exists q' \in Q, (q, l, q') \in \Delta_{loc}$

Note that we explicitly require that all potential push symbols have to appear in a return transition from any state. This way, we ensure that all potential transitions are *written* inside the VPA. Of course, some such transitions are certainly impossible to use inside any valid run, but they give a simple completeness definition for VPA. For instance, in the VPA illustrated on the right of Figure 3.1, the stack symbol[1] $*$ cannot be read in the initial state $q_0$.

**Remark** One can complete any VPA $\mathcal{A}$ by first applying Proposition 3.2.1, then adding a **sink** state that is not final. Then we can add to $\mathcal{A}$ all transitions needed to satisfy the definition to end in this sink state. The resulting automaton has only one more state and recognizes exactly the same language as $\mathcal{A}$.

**Proposition 3.2.2** *For every pushdown alphabet* $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$*, there are only two (up to isomorphism) complete single-state* VPA *called* $\mathcal{V}_{\Sigma^*}$ *and* $\mathcal{V}_{\emptyset}$ *that recognizes respectively* $\Sigma^*$ *and* $\emptyset$*.*

**Proof** First note that a complete automaton should have the set $Q \times Q \times \Sigma_{call}$ as its stack alphabet, and in the case that the VPA is reduced to a single state, this set is equivalent to $\Sigma_{call}$.

Also an automaton that has only one state $q$ can have only one set of transitions that satisfies the definition of a complete VPA, i.e. the set $\Delta = \Delta_{call} \cup \Delta_{ret} \cup \Delta_{loc}$ such that $\Delta_{call} = \{(q, c, q, c) \mid c \in \Sigma_{call}\}$, $\Delta_{ret} = \{(q, r, c, q) \mid r \in \Sigma_{ret}, c \in \Sigma_{call}\} \cup \{(q, r, \perp, q) \mid r \in \Sigma_{ret}\}$ and $\Delta_{loc} = \{(q, l, q) \mid l \in \Sigma_{loc}\}$.

The only parameter of such an automaton is whether $q$ is final or not. In the former case, the VPA is $\mathcal{V}_{\Sigma^*}$ and recognizes obviously $\Sigma^*$ and in the later case the automaton is $\mathcal{V}_{\emptyset}$ and recognizes $\emptyset$. □

We can now define and show the hardness of the minimization problem for non-deterministic VPA.

**Definition** (MIN-NVPA)

INSTANCE: A NVPA $\mathcal{A}$ and a positive integer $k < |\mathcal{A}|$.

QUESTION: Is there a complete NVPA $\mathcal{B}$ of size $k$, such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ ?

**Theorem 3.2.3** *The decision problem* MIN-NVPA *is* EXPTIME*-complete.*

**Proof** First remark that MIN-NVPA is in EXPTIME as we can simply determinize $\mathcal{A}$, then enumerate all $k$ states NVPA (recall we can assume $\Gamma = Q^2 \times \Sigma_{call}$) and check each equivalence in EXPTIME.

To prove that MIN-NVPA is also EXPTIME-hard, we reduce the universality problem for NVPA to it (which is known to be EXPTIME-complete by [2]).

Precisely, if we execute MIN-NVPA with the input $(\mathcal{A}, 1)$, i.e. we ask if $\mathcal{A}$ can be reduced to a complete one state automaton. If the answer is no, then we can deduce that $\mathcal{A}$ does not recognize $\Sigma^*$ as $\Sigma^*$ can easily be recognized by a one state automaton (Proposition 3.2.2).

Moreover, if the minimal complete automaton recognizing $\mathcal{L}(\mathcal{A})$ has one state it is either $\mathcal{V}_{\Sigma^*}$ or $\mathcal{V}_{\emptyset}$ (Proposition 3.2.2) and since we can check in polynomial time if $\mathcal{L}(\mathcal{A}) = \emptyset$, we can know whether $\mathcal{L}(\mathcal{A}) = \Sigma^*$ or not. □

[1]or $\{(q_0, q_1, c)\}$ in a complete automaton.

## 3.3 Minimization of Vpa

### 3.3.1 Min-Vpa is in Np

In Section 3.1, we have seen the relevance to minimize the number of states of Vpa, and especially why it is more useful to minimize it rather than other size parameters of such automata. Since the minimization of Nvpa is computationally hard, we will now consider here the decision problem of finding a minimal deterministic automaton for a given Vpl, defined below.

**Definition** (Min-Vpa)
Instance: A Vpa $\mathcal{A}$ and a positive integer $k < |\mathcal{A}|$.
Question: Is there a Vpa $\mathcal{B}$ of size $k$, such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ ?

Whether Min-Vpa is a computationally hard question is an open problem, but it is suspected to be hard to solve as no polynomial time algorithm is known. Yet contrary to their non-deterministic versions, we can test in polynomial time the equivalence of Vpa. Thanks to this test, we can easily show that Min-Vpa is in Np.

**Proposition 3.3.1** *The problem* Min-Vpa *is in* Np*.*

**Proof** One can guess an automaton of size $k$ and check in polynomial time using Proposition 2.3.1 (page 13) that it is equivalent to the input automaton.

Note since the stack alphabet is bounded by the product of the call alphabet and the state set, there are *only* $\mathcal{O}(2^{k^3})$ distinct Vpa (up to isomorphism) when the size of the input alphabet is considered as constant. □

**Remark** The above proof shows also that Min-Vpa is Fixed Parameter Tractable as we can decide in time $\mathcal{O}(|\mathcal{A}|^6 \cdot 2^{k^3})$ what is the correct answer (recall $k < |\mathcal{A}|$ and Theorem 2.3.1, page 13).

### 3.3.2 Uniqueness

As previously stated, a Vpa $\mathcal{A}$ is said to be **minimal** when its number of states is the minimal number of states of all other Vpa that recognize the same language as $\mathcal{A}$. Unfortunately, such minimal automaton is not unique for a given Vpl and several minimal non-isomorphic Vpa can exist. Let us prove formally our claim:

**Proposition 3.3.2** *Some* Vpl *do not have a minimal unique* Vpa *recognizing them.*

**Proof** Let us consider for instance the language $L = \{(c_1 1(11)^* r + c_2(11)^* r)^*\}$ over the pushdown alphabet $\hat{\Sigma} = (\{c_1, c_2\}, \{r\}, \{1\})$. First we can show that $L$ is effectively a Vpl, as some Vpa recognizes it. A simple argument is that $L$ is a regular language and the class of regular languages is embedded inside the class of Vpl.

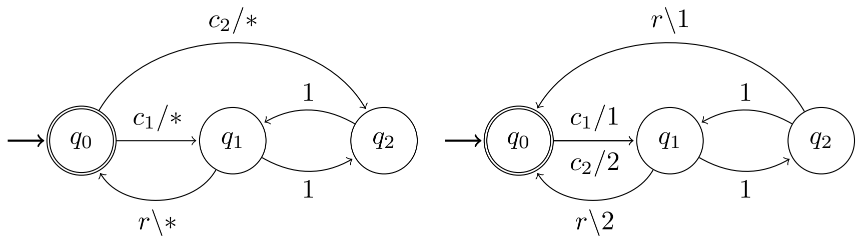

Figure 3.2: Two non-isomorphic minimal VPA for the VPL $L = \{(c_1 1(11)^* r + c_2(11)^* r)^*\}$. On the left $\mathcal{A}_1$, and on the right $\mathcal{A}_2$.

Let us now consider the VPA $\mathcal{A}_1$ and $\mathcal{A}_2$ illustrated in Figure 3.2. One can easily check that both are recognizing $L$. Let us prove that any VPA needs at least 3 states in order to recognize $L$. Let $\mathcal{B}$ be a 2-states automaton recognizing $L$. Since $1 \in \Sigma_{loc}$, the automaton cannot *touch* the stack when reading 1's but still has to count their parity. In order to do so it needs two states, let us say $q_1$ and $q_2$ such that $(q_1, 1, q_2) \in \Delta_{loc}$ and $(q_2, 1, q_1) \in \Delta_{loc}$. One of them must be initial, and one them final. Thus $1 \in \mathcal{L}(\mathcal{B})$ or $11 \in \mathcal{L}(\mathcal{B})$. Contradiction.

Finally, the two automata of Figure 3.2 are not isomorphic, since $\mathcal{A}_2$ requires that all transitions outgoing from $q_0$ go to the same state, and this is definitively not compatible with the other transitions of $\mathcal{A}_1$. □

The fact that minimal VPA is not unique has great consequences on their minimization. Indeed, one cannot merge states in order to find the minimal automaton, as it is made for DFA minimization. For instance, consider the automaton of Figure 2.3, page 11. It is clearly not a minimal automaton, and one can think that by merging $S_0$ and $S_1$, the automaton could become minimal (see Figure 2.2, page 9, for a minimal automaton). This guess is wrong as this *reduced* automaton recognizes an extra word $w = b.c$ which is not in the desired language. This implies there is not always a homomorphism between a VPA and a minimal equivalent one, thus the minimization of VPA cannot follow similar techniques as in the finite state case.

Even if the minimal VPA for a VPL is not unique, the minimal automaton has still some great advantages. One possible application of minimizing VPA is to use VPA as a model in order to recognize regular languages, but using significantly less states thanks to their stack. For instance, there are some regular languages where the minimal VPA can be exponentially smaller that the minimal DFA. One such example is the regular language $L = \{w\widetilde{w} \mid \widetilde{w}$ is the mirror of $w$ and $|w| = k\}$ over a finite alphabet $\Sigma$ where $k$ is a constant (the language that we have used during the proof of Proposition 3.1.4). The minimal DFA that recognizes $L$ must remember $w$ in its state as there is a unique $\widetilde{w}$ which belongs to $L$, and thus such minimal DFA must have at least $\mathcal{O}(|\Sigma|^k)$ states. But the minimal VPA recognizing $L$ only requires $k$ states (the alphabet has just to be slightly modified to contain push and pop symbols).

| | **Complexity of Minimization** | | Uniqueness |
|---|---|---|---|
| | Membership | Hardness | |
| **String Automata** | | | |
| DFA [17] | in PTIME | NLOGSPACE-hard | ✓ |
| UFA [15] | in NP | NP-hard | × |
| NFA [18] | in PSPACE | PSPACE-hard | × |
| **Tree Automata** | | | |
| DTA [19, 20] | in PTIME | PTIME-hard | ✓ |
| DUTA [21] | in NP | NP-hard | × |
| UTA/UUTA [21] | in NP | NP-hard | × |
| NTA/NUTA [21] | in EXPTIME | EXPTIME-hard | × |
| **Pushdown Automata** | | | |
| DVPA | in NP | ? | × |
| NVPA (Theorem 3.2.3) | in EXPTIME | EXPTIME-hard | × |
| DPDA | $\leq$ Primitive Recursive | | × |

Table 3.1: The complexity of the minimization problem and uniqueness of a minimal automaton for different models of automata that are related to the VPA.

### 3.3.3 Insights into hardness

We can wonder if the minimization problem for VPA can be suspected to be hard and why. We know (see Table 3.1 for a full sum up of the complexity of minimizing different models of automata) that only DFA and DTA (Deterministic Tree Automata) can be minimized within polynomial time if P $\neq$ NP. Since DVPA are deterministic and share several aspects with DTA, one could hope that their minimization is still possible in polynomial time.

But we can see that a little non-determinism makes the minimization problem harder (for instance the problem is NP-complete for UFA and Unambiguous (Unranked) Tree Automata - UTA/UUTA) and full non-determinism makes the problem even harder (it is EXPTIME-complete for Nondeterministic (Unranked) Tree Automata - NTA/NUTA and NVPA). Although we do not understand yet where is the combinatorial problem when trying to minimize VPA, we can see the stack influences in many way the semantic of the states and we can relate this to non-determinism in some aspects.

Another argument that convinces us of the hardness of the minimization of VPA is the uniqueness of a minimal machine. The only other models that are known to be minimizable in polynomial time (i.e. DFA and DTA) accept a unique (up to isomorphism) minimal automaton for each recognized language. We have already shown that there are VPL that have no unique minimal automaton, and thus we may think that the minimization problem for VPA is computationally hard to solve. We failed to prove the hardness of minimizing VPA although we suspect the problem to be NP-hard. In fact, we do not know how to minimize even one of the simplest subclasses of VPL, see Section 5.3, page 43, for details.

## 3.4 Congruences

Visibly pushdown languages have different formulations: based on VPA, MSO with logic, Regular Tree Automata, Visibly pushdown grammar (see [2] for details about these formulations) and nested word automata [12]. We can also define VPL based upon congruences characterization. We shall see in this section congruences for VPL that lead to a canonical machine for every VPL. Unfortunately, this automaton can be exponentially bigger than a minimal VPA.

Precisely, we shall define three congruences for a given VPL (based on [8]). The first one $\sim$ will be for words when the stack is empty and is equivalent to the traditional Myhill-Nerode right congruence. The second one $\approx$ will be for words that need not see the stack to be distinguished. The third one $\equiv$ will be used for well-matched words.

**Definition** We define the visibly pushdown congruences $\sim, \approx, \equiv$ for a given VPL $L$ over a pushdown alphabet $\hat{\Sigma}$ as:
For $u_1, u_2 \in MC(\hat{\Sigma})$, $u_1 \sim u_2$ iff $\forall v \in \Sigma^*, u_1.v \in L \Leftrightarrow u_2.v \in L$
For $u_1, u_2 \in \Sigma^*$, $u_1 \approx u_2$ iff $\forall v \in MR(\hat{\Sigma}), u_1.v \in L \Leftrightarrow u_2.v \in L$
For $w_1, w_2 \in WM(\hat{\Sigma})$, $w_1 \equiv w_2$ iff $\forall u, v \in \Sigma^*, u.w_1.v \in L \Leftrightarrow u.w_2.v$

The finiteness of the number of the equivalence classes for $\sim, \approx$ and $\equiv$ is in fact a characterization of VPL, as stated by the next theorem.

**Theorem 3.4.1 ([8])** *A language $L$ is a* VPL *over $\hat{\Sigma} \Leftrightarrow \sim, \approx$ and $\equiv$ are of finite index.*

This characterization results in a canonical machine for visibly pushdown languages, in contrast to a minimal VPA which is not necessarily unique (see Proposition 3.3.2).

**Proposition 3.4.2 ([8])** *For every* VPL*, there is a canonical* VPA *over $\hat{\Sigma}$ based upon $\sim, \approx$ and $\equiv$.*

**Proof** Let $L$ be a well-matched language over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. Theorem 3.4.1 claims that $\sim, \approx$ and $\equiv$ have finitely many equivalence classes.

The idea is to build an automaton that has equivalence classes of $\sim$ (for the empty stack) and couple of equivalence classes of $\approx$ and $\equiv$ as states. We will keep as an invariant that after reading a word $u \in \Sigma^*$ the configuration of the automaton is:

- $([u]_\sim, \perp)$ if $u \in MC(\hat{\Sigma})$.
- $(([u]_\approx, [w_n]_\equiv), \sigma)$ otherwise, s.t.:
  - $u = u'c_n w_n$ with $u' \in \Sigma^*$, $c_n \in \Sigma_{call}$ and $w_n \in WM(\hat{\Sigma})$
  - $u' = vc_1w_1...c_{n-1}w_{n-1}$ with $v \in MC(\hat{\Sigma})$, $w_1, ..., w_{n-1} \in WM(\hat{\Sigma})$, $c_1, ..., c_{n-1} \in \Sigma_{call}$
  - $\sigma = \perp([v]_\sim, c_1)(([vc_1w_1]_\approx, [w_1]_\equiv), c_2)...(([u']_\approx, [w_n]_\equiv), c_n)$

Formally, we construct the VPA $\mathcal{A} = (Q, q_i, Q_F, Q \times \Sigma_{call}, \Delta)$, where $Q = \{[u]_\sim \mid u \in MC(\hat{\Sigma})\} \cup \{([u]_\approx, [w]_\equiv) \mid u \in \Sigma^* \wedge w \in WM(\hat{\Sigma})\}$, $q_i = [\varepsilon]_\sim$, $Q_F = \{[u]_\sim \mid u \in L\} \cup \{([u]_\approx, [w]_\equiv) \mid u \in L \wedge w \in WM(\hat{\Sigma})\}$ and with the following transitions[2]:

**Empty Stack**

- for all $c \in \Sigma_{call}$, $([u]_\sim, c, ([uc]_\approx, [\varepsilon]_\equiv)) \in \Delta_{call}$
- for all $r \in \Sigma_{ret}$, $([u]_\sim, r, \bot, [ur]_\sim) \in \Delta_{ret}$
- for all $l \in \Sigma_{loc}$, $([u]_\sim, l, [ul]_\sim) \in \Delta_{loc}$

**Non-empty Stack**

- for all $c \in \Sigma_{call}$, $(([u]_\approx, [w]_\equiv), c, ([uc]_\approx, [\varepsilon]_\equiv)) \in \Delta_{call}$
- for all $r \in \Sigma_{ret}$,
  - $(([u]_\approx, [w]_\equiv), r, (([u']_\approx, [w']_\equiv), c), ([u'cwr]_\approx, [w'cwr]_\equiv)) \in \Delta_{ret}$
  - $(([u]_\approx, [w]_\equiv), r, ([u']_\sim, c), [u'cwr]_\sim) \in \Delta_{ret}$
- for all $l \in \Sigma_{loc}$, $(([u]_\approx, [w]_\equiv), l, ([ul]_\approx, [wl]_\equiv)) \in \Delta_{loc}$

One can check that our previous invariant is indeed correct and thus $\mathcal{A}$ recognizes exactly $L$.

□

Proposition 3.4.2 allows us to construct a canonical automaton for a given VPL and once this automaton has been computed, we can check equivalence of VPA efficiently. Besides the constructed automaton might not be the minimal VPA, it can also be exponentially bigger.

For instance, consider the language $L_k = \{a_i.c.L_{a_i}.r \mid i \in [1..k]\}$ over the pushdown alphabet $\hat{\Sigma} = (\{c\}, \{r\}, \Sigma_{loc})$ with $k$ being a fixed integer and where $\Sigma_{loc} = \{a_1, a_2, ..., a_k\}$ and $L_{a_i} = \{w \in \Sigma_{loc}^* \mid \text{the number of } a_i \text{ in } w \text{ is even}\}$. After reading $a_i$, the automaton built using the above proof will go to state $[a_i]_\sim$, which are obviously different states for each $i \in [1..k]$.

In state $[a_i]_\sim$, when reading $c$ the automaton will go to state $([a_ic]_\approx, [\varepsilon]_\equiv)$. The problem is $[a_1c]_\approx = [a_2c]_\approx = ... = [a_kc]_\approx$, as for all matched-return[3] words $v$ and $i \in [1..k]$, $a_i.c.v \notin L$. Thus after reading $c$, the automaton has completely forgotten which was the initial local action read (i.e. the initial $a_i$) and it has to wait until reading the last return symbol before checking the stack. Hence, the automaton needs at least $2^k$ states to memorize all possible combinations of parity of $a_i$'s.

Remark also that we can build a trivial automaton that uses only $\mathcal{O}(k)$ states. The automaton has just to go to different states after reading $c$ and it can check the parity of $a_i$ with only two states. An example of such automaton is depicted in Figure 5.1, page 36 (where $k = 3$).

[2]Call transitions push the current state and the call symbol, so we omit them for clarity.

[3]In fact, the only words $v \in \Sigma^*$, such that $a_icv \in L$ are $v \in L_{a_i}r$, which is trivially not a matched-return word.

# 4 Block Visibly Pushdown Automata

*Block Visibly Pushdown Automata* (Bvpa) is an automaton model introduced recently by Chevret and Walukiewicz [14] in order to combine the expressiveness of Vpa and still conserve some good minimization properties. A Bvpa is a Vpa where the states can be partitioned into *modules* (or *blocks*), such that when reading a local action the current module is unchanged and if a call is read in module $m$, then when we read its matching return the automaton come back to module $m$. Another way to express this, is that two positions in a word with the same stack level $l$, such that all intermediate positions are on levels greater than $l$, will be within the same module[1]. Moreover, each module has only one *entry-state*, i.e. transitions can enter the module by only a single state.

An interesting property of Bvpa is that for each Vpa, one can build an equivalent Bvpa of quadratic size. Thus minimizing such automaton gives a Bvpa that is no more than quadratically bigger than the minimal Vpa for the desired visibly pushdown language, and therefore it gives a quadratic approximation of the minimal Vpa. Bvpa is the only model where a minimization algorithm is known that achieves a polynomial approximation for Vpa.

Chevret and Walukiewicz proposed a minimization algorithm for Bvpa relative to so-called *associated partition* (see Section 4.2 for definition). Thus if one finds a Bvpa with an optimal associated partition, we can use the minimization algorithm in order to find the minimal Bvpa for a given Vpl, hence a quadratic approximation of the minimal Vpa. Unfortunately a method to find such a partition is not known yet and moreover there exist partitions that lead to an exponentially larger minimal Bvpa compared to the minimal Vpa.

We shall first introduce in Section 4.1 the model of Bvpa and useful notions about it. Then, we will describe in Section 4.2 associated partitions and known results about Bvpa minimization. In Section 4.3, we will prove the hardness of a *module-based* minimization of Bvpa. We will conclude by focusing on the general Bvpa minimization in Section 4.4.

## 4.1 Definitions

Bvpa are deterministic Vpa with a partition on their states which separates the states into **modules**. First, Bvpa assume the stack alphabet to be the cross product of its states and call alphabet. Recall every Vpa can be *transformed* to match this requirement without increasing its number of states [2].

[1]For this reason the starting module always corresponds to an empty stack.
[2]See Section 3.1, page 17, or Figure 4.1 for an example.

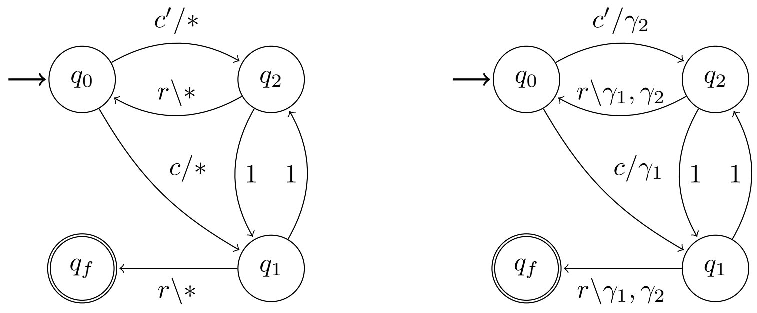

Figure 4.1: Two VPA that recognize the visibly pushdown language $L = (c1(11)^*r|c'(11)^*r)^* \boldsymbol{.} (c(11)^*r|c'1(11)^*r)$. On the left, using a single stack-symbol $*$; on the right, using a *uniform* stack with $\gamma_1 = (c, q_0)$ and $\gamma_2 = (c', q_0)$.

Intuitively, a module is a set of states that has a single entry and every local transition must stay inside the same module. Also for each call transition $(q, c, q', (q, c))$, going from module $m$ to $m'$, every return transition going from module $m'$ and reading $(q, c)$ as stack symbol must end in module $m$. Finally, we forbid for technical reasons to re-use the module of the empty stack, i.e. no call transitions end in the initial state module. The formal definition is as follows:

**Definition** A VPA $\mathcal{A} = (Q, q_i, Q_F, Q \times \Sigma_{call}, \Delta)$ over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ is a BVPA, if there exists a partition of the states into modules such that if $[q]$ denotes the module of state $q$, the following properties hold:

> **Empty stack:** for all $(q, c, q', (q, c)) \in \Delta_{call}$, $[q_i] \neq [q']$
>
> **Single entry:** for all $(q_1, c_1, q'_1, (q_1, c_1)), (q_2, c_2, q'_2, (q_2, c_2)) \in \Delta_{call}$, $[q'_1] = [q'_2] \to q'_1 = q'_2$
>
> **Return to entry module:** for all $(q, r, (q', c), q'') \in \Delta_{ret}$, $[q'] = [q'']$
>
> **Local actions preserve module:** for all $(q, l, q') \in \Delta_{loc}$, $[q] = [q']$

Note that we will usually not mention the stack when defining BVPA as it can be directly inferred from the state set and the pushdown alphabet. For the same reasons, push transitions will be noted $(q, c, q')$ since the push symbol is implicitly $(q, c)$. Now, we can see how to build a BVPA from a VPA:

**Proposition 4.1.1** *For every* VPA $\mathcal{A}$*, we can compute in quadratic time an equivalent* BVPA *of size* $\mathcal{O}(|\mathcal{A}|^2)$*.*

**Proof** Let $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta)$ be a VPA. Recall that we assume that $\Gamma = Q \times \Sigma_{call}$ and the automaton pushes $(q, c)$ when using a transition from control state $q$ and reading call symbol $c$.

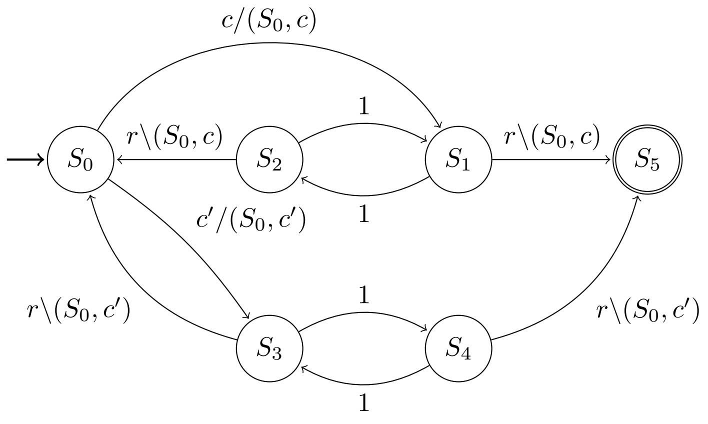

Figure 4.2: BVPA that recognizes the same language as the VPA of Figure 4.1 (right), i.e. $\mathcal{L}(\mathcal{B}) = (c1(11)^*r|c'(11)^*r)^*(c(11)^*r|c'1(11)^*r)$. The BVPA is built using Proposition 4.1.1. We have $S_0 = (I, q_0)$, $S_1 = (q_1, q_1)$, $S_2 = (q_1, q_2)$, $S_3 = (q_2, q_2)$, $S_4 = (q_2, q_1)$ and $S_5 = (I, q_f)$.

The idea of the proof is to build a BVPA $\mathcal{B}$ based on $\mathcal{A}$ by simulating a copy $\mathcal{A}$ for each module of $\mathcal{B}$.

Precisely, each module will be associated with a single state of $\mathcal{A}$, and it will be accessed by any transition *pointing* to this state (hence we only need a module per state with ingoing call transitions). Then the module will simulate $\mathcal{A}$ until a call transition that targets another state of $\mathcal{A}$ is read.

Formally, let $M = \{q \mid \exists(q', c, q, (q', c)) \in \Delta_{call}\} \cup \{I\}$ and $\mathcal{B} = (Q', q'_i, Q'_F, \Delta')$ be a BVPA defined as follows. We set $Q' = \{(m, q) \mid m \in M\ ,\ q \in Q\}$, $q'_i = (I, q_i)$ and $Q'_F = \{(m, q) \in Q' \mid q \in Q_F\}$. The transitions are as follows for all $m, m' \in M$:

- for each $(q, c, q') \in \Delta_{call}$, $((m, q), c, (q', q')) \in \Delta'_{call}$
- for each $(q, r, (q', c), q'') \in \Delta_{ret}$, $((m, q), r, ((m', q'), c), (m', q'')) \in \Delta'_{ret}$
- for all $(q, l, q') \in \Delta_{loc}$, $((m, q), l, (m, q')) \in \Delta'_{loc}$

We can easily check that $\mathcal{B}$ is a BVPA by setting the module of $(m, q) \in Q'$ to be $m$ (note the size of $\mathcal{B}$ is indeed $\mathcal{O}(|\mathcal{A}|^2)$). Finally, it recognizes exactly the same language of $\mathcal{A}$ as after reading a word $w$ if $\mathcal{B}$ is in state $(m, q)$ then the current state of $\mathcal{A}$ is $q$.

$\square$

**Example** Consider the VPA of Figure 4.1 (right). It is not a BVPA as $q_1$ and $q_2$ must belong to the same module (due to local transitions) but this particular module possesses two entry-states ($q_1$ and $q_2$ because of transitions $(q_0, c, q_1)$ and $(q_0, c', q_2)$). However, we can build a BVPA using Proposition 4.1.1 (illustrated in Figure 4.2) that has only two more states. In this Bvpa, we can clearly see the states $S_1$ and $S_2$ (resp. $S_3$ and $S_4$) simulating $\mathcal{A}$ when the last transition used was $(q_0, c, q_1)$ (resp. $(q_0, c', q_2)$).

**Remark** When a Bvpa has $k$ modules, we will usually refer as a **k-module** Bvpa. Also, we name the module of the initial state of a Bvpa the **empty stack module**, and the other modules are the **call** modules.

## 4.2 Associated partition minimization

In the last section, we have introduced how to build a Bvpa of quadratic size from any Vpa and thus the approximative minimization of Vpa can be achieved through minimization of Bvpa. Unfortunately, we do not yet know how to minimize efficiently Bvpa, however [14] presents a way to minimize any Bvpa under the assumption to keep the same *module partition* (the minimization applies to each module separately). More precisely, for any Bvpa $\mathcal{B}$ there is a unique (up to isomorphism) minimal Bvpa $\mathcal{B}_{min}$ that enters the same module as $\mathcal{B}$ at any call. In particular, $\mathcal{B}$ and $\mathcal{B}_{min}$ have the same number of modules.

**Definition** Let $\mathcal{B} = (Q, q_i, Q_F, \Delta)$ be a k-module Bvpa over the visibly pushdown alphabet $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ and $M$ the set of its *call* modules ($|M| = k - 1$). The **associated partition** of $\mathcal{B}$ is the partition of $L_M = \{u.c \mid u \in MR(\hat{\Sigma}), c \in \Sigma_{call}, u.c \in Pref(L)\}$ into $|M|$ disjoint sets $(L_m)_{m \in M}$ such that:

$$\forall m \in M, L_m = \{u.c \mid u.c \in L_M \wedge [q] = m \text{ s.t. } q_i \xrightarrow{uc}_{\mathcal{B}} q\}$$

**Theorem 4.2.1 ([14])** *Given a* Bvpa *$\mathcal{B}$, we can compute in cubic time the unique (up to isomorphism) minimal equivalent* Bvpa *that has the same associated partition.*

The idea behind the proof of the previous theorem is first to build an Ecda $\mathcal{E}$ (another variant of Vpa defined in Section 5.1.1) based on the original Bvpa $\mathcal{B}$. Then we can minimize $\mathcal{E}$ and convert back to Bvpa $\mathcal{B}_{min}$ in order to obtain a minimal unique automaton with the same module partitioning as $\mathcal{B}$. Also, Theorem 4.2.1 is a promising first step in the minimization of Vpa. If we start from a Bvpa that has the optimal associated partition, we can compute a good approximation of the minimal Vpa. The actual problem is to find such a Bvpa to start our minimization, and finding a good partition seems as hard as the general minimization of Bvpa (see next section).

Chevret and Walukiewicz also showed that some Bvpa have a really unefficient associated partition, and their minimization leads to an exponential blow-up if we compare to the minimal Bvpa. For instance, consider the visibly pushdown language[3] $L_k$. In Section 3.4, we have stated that there exists a Vpa of size $\mathcal{O}(2^k)$ which recognizes $L_k$ ($k \in N$). This Vpa $\mathcal{A}_k$ is in fact a Bvpa

[3]The definition of $L_k$ is given in Section 3.4, page 26. An example of $L_k$ for a fixed integer is illustrated in Figure 5.1, page 36 (where $k = 3$).

that has two modules: the empty stack module and a single call module $m_c$. Its associated partition is $L_{m_c} = \{a_1c, ..., a_kc\}$ and $\mathcal{A}_k$ is already the smallest BVPA with such associated partition. However, there exists obviously a BVPA that has $k$ modules (one for each $a_ic$, $i \in [1..k]$) and only $\mathcal{O}(k)$ states (see page 26 for the construction).

## 4.3 Module minimization

The minimization problem for BVPA can be stated in different ways, and an interesting one is when we want to minimize both the number of modules and the size of them. Precisely, we do not want to minimize the total number of states of the automaton, but rather the number and the size of the modules. In practice, this is motivated when each module represents a subroutine of a recursive program, and we want to bound both the number of subroutines and their maximum size.

Let us formally define the aforementioned decision problem:

**Definition** (K-S-MIN-BVPA)

INSTANCE: A BVPA $\mathcal{B}$ and two integers $k$ and $s$ given in unary.

QUESTION: Is there a $k$-module BVPA $\mathcal{A}$, such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ and the size of each module is bounded by $s$?

**Proposition 4.3.1** K-S-MIN-BVPA *is in* NP.

**Proof** K-S-MIN-BVPA is indeed in NP as one can guess a VPA $\mathcal{A}$ and a partition of its states[4], then check if the automaton $\mathcal{A}$ with this partition is a $k$-module BVPA and that each module does not exceed $s$ states. Finally, we can check in polynomial time if two BVPA are equivalent, using Proposition 2.3.1 since BVPA are naturally also VPA.

□

Such a minimization of BVPA is in fact computationally hard and we can prove it using a reduction to a variant of BIN-PACKING that we call SET-UNION-PACKING. Recall bin packing is the problem of finding a partition of a set of weighted objects into bins that have a limited capacity, such that each bin does not overfill and the total number of bins used is minimal. For the NP-completeness of (unary) BIN-PACKING, we guide the reader toward [7].

**Definition** (BIN-PACKING)

INSTANCE: A finite set $U$ of items, a size $v(u) \in \mathbb{N}^+$ for each $u \in U$ and two integers $k$ and $s$, all given in unary.

QUESTION: Is there a partition of $U$ into $k$ disjoint sets $U_1, U_2, ..., U_k$ such that for each $1 \leq j \leq k$, $\sum\limits_{u \in U_j} v(u) \leq s$.

[4]Remark there are at most $\mathcal{O}(2^K)$ possible BVPA and partitions with $K = k \cdot s$.

**Definition** (SET-UNION-PACKING)

INSTANCE: A collection $S_1, S_2, ..., S_n$ of subsets of a finite set $U$ and two integers $k$ and $s$, all given in unary.

QUESTION: Is there a partition of $\{1, 2, ..., n\} = \bigcup_{i \in [1..k]} X_i$ such that for each $1 \leq j \leq k$, $|\bigcup_{i \in X_j} S_i| \leq s$?

**Lemma 4.3.2** SET-UNION-PACKING *is* NP-*complete.*

**Proof** First, we can see that SET-UNION-PACKING is in NP as one can guess a partition of $\{1, 2, ..., n\}$ then check in linear time that the partition satisfies the size condition on each set.

Let $I = (U, v, k, s)$ with $U = u_1, u_2, ..., u_n$ be an instance of BIN-PACKING. Let us build an instance $I' = (U', (S_i)_{1 \leq i \leq n}, k', s')$ of SET-UNION-PACKING. We set $U' = \{u^i \mid u \in U \wedge 1 \leq i \leq n\}$ and we set for each $1 \leq i \leq n$, $S_i = \{u_i^1, ..., u_i^{v(u)}\}$ (recall that the weights $v(u)$ are given in unary). Finally we set $k' = k$ and $s' = s$.

Let us prove now that $I$ has a solution for BIN-PACKING if and only if $I'$ has a solution for SET-UNION-PACKING.

Assume we can find a $k$ partition of $U$ into disjoint sets $U_1, U_2, ..., U_k$ such that for each $1 \leq j \leq k$, $\sum_{u \in U_j} v(u) \leq s$. Then the partition $X_1, X_2, ..., X_k$ of $\{1, 2, ..., n\}$ s.t. $1 \leq j \leq k$, $X_j = \{i \mid u_i \in U_j\}$ is a solution of $I'$. Indeed, for each $1 \leq j \leq k$, $\sum_{u \in U_j} v(u) \leq s$ we have that for each $1 \leq j \leq k$, $|\bigcup_{i \in X_j} S_i| \leq s$ by definition of $I'$ (note that the $S_i$ are disjoint). Moreover, since $U_1, U_2, ..., U_k$ is a partition of $U$, $X_1, X_2, ..., X_k$ is effectively a partition of $\{1, 2, ..., n\}$.

Now assume there is a partition $X_1, X_2, ..., X_k$ of $\{1, 2, ..., n\}$ such that $1 \leq j \leq k$, $|\bigcup_{i \in X_j} S_i| \leq s$. In the same fashion as previously, we build the solution $\Gamma = U_1, U_2, ..., U_k$ for $I$, such that for each $1 \leq j \leq k$, $U_j = \{u_i \mid i \in X_j\}$. Since $X_1, X_2, ..., X_k$ is a partition $\{1, 2, ..., n\}$, every $u \in U$ should belong to a set $U_i$, and thus $\Gamma$ is indeed a partition of $U$. Finally, since for every $1 \leq j \leq k$, $|\bigcup_{i \in X_j} S_i| \leq s$ we get that for every $1 \leq j \leq k$, $\sum_{u \in U_j} v(u) \leq s$. □

**Lemma 4.3.3** K-S-MIN-BVPA *is* NP-*complete.*

**Proof** First, recall that K-S-MIN-BVPA is in NP as stated by Proposition 4.3.1.

The idea of the proof is a reduction from decision problem SET-UNION-PACKING that has been shown to be NP-complete in Lemma 4.3.2. Let $I = (S_1, S_2, ..., S_n, U, k, s)$ be an instance of SET-UNION-PACKING. We will build a BVPA that recognizes the language $L_I = \{c_i.u.r_u \mid u \in S_i\}$ over the visibly pushdown alphabet $\hat{\Sigma} = (\{c_i \mid 1 \leq i \leq n\}, \{r_u \mid u \in U\}, U)$.

Let $\mathcal{B} = (Q, q_0, Q_F, \Delta)$ be a BVPA over $\hat{\Sigma}$. We set $Q = \{q_0, q_1\} \cup \{q_u \mid u \in U\} \cup \{q_f\}$, with $Q_F = \{q_f\}$. We use the following transitions:

- for each $1 \leq i \leq n$, $(q_0, c_i, q_1) \in \Delta_{call}$

- for each $u \in U$, $(q_1, u, q_u) \in \Delta_{loc}$
- for each $u \in U$, $(q_u, r_u, (q_0, c_i), q_f) \in \Delta_{ret} \Leftrightarrow u \in S_i$

It is straightforward that $\mathcal{B}$ recognizes $L_I$, and also that $\mathcal{B}$ is a Bvpa (it has only two modules $\{q_0, q_f\}$ for the empty stack and the rest of the states).

Now let $(\mathcal{B}, k+1, s+1)$ be an instance of K-s-min-bvpa and let us prove it has a solution if and only if $I$ has a solution.

Assume there exists a partition $X = X_1, X_2, ..., X_k$ of $\{1, 2, ..., n\}$ such that $X$ is a solution for $I$. Then we build the following Bvpa $\mathcal{C} = (Q', q'_0, Q'_F, \Delta')$ such that $Q' = \{q'_0\} \cup \{q'_j \mid 1 \leq j \leq k\} \cup Q'_J \cup \{q'_f\}$ with $Q'_J = \{q'_{j,u} \mid \exists i \in X_j, u \in S_i\}$ and $Q'_f = \{q'_f\}$. $\mathcal{C}$ has the following transitions:

- for each $1 \leq i \leq n$, $(q'_0, c_i, q'_j) \in \Delta_{call} \Leftrightarrow i \in X_j$
- for each $1 \leq j \leq n$, $q'_{j,u} \in Q'_J$, $(q'_j, u, q'_{j,u}) \in \Delta_{loc}$
- for each $q'_{j,u} \in Q'_J$, $(q'_{j,u}, r_u, (q'_0, c_i), q'_f) \in \Delta_{ret} \Leftrightarrow i \in X_j \wedge u \in S_i$

There is a simple partition of $Q'$ into modules (in fact there is only one possible) by letting the empty stack module $m_I$ to be $\{q'_0, q'_f\}$, and then the $k$ other modules named $m_j$ for each $1 \leq j \leq k$ are the union of the state $q'_j$ and the states $q'_{j,u} \in Q'_J$. Let us check that $\mathcal{C}$ is indeed a Bvpa against this partitioning. There is no call transitions that end in the empty stack module, all return transitions come back to the original module (i.e. $\{q'_0, q'_f\}$), local transitions stay in the same module and finally there is only a single entry state by module ($q'_j$ for each module $m_j$). Moreover each module $m_j$ is of size bounded by $s+1$ as $m_j = \{q'_j\} \cup \{q'_{j,u} \in Q'_J\} = \{q'_j\} \cup \{q'_{j,u} \mid u \in \bigcup_{i \in X_j} S_i\}$, i.e. $|m_j| \leq |\bigcup_{i \in X_j} S_i| + 1$.

Eventually we have to verify that $\mathcal{C}$ recognizes exactly $L_I$. Consider a word $w = c.u.r$ and let examine when it is accepted by looking at possible ways to reach $q'_f$: $w$ is accepted by $\mathcal{C}$ iff $c = c_i \in \Sigma_{call}$ for some $i \in [1..n]$ and $(q'_j, (q'_0, c_i)) \xrightarrow{u.r}_{\mathcal{C}} (q'_f, \perp)$ s.t. $X_j$ contains[5] $i$, i.e. if $u \in \{u \in S_i \mid i \in X_j\}$ and $(q'_{j,u}, (q'_0, c_i)) \xrightarrow{r}_{\mathcal{C}} (q'_f, \perp)$ since $(q'_j, (q'_0, c_i)) \xrightarrow{u}_{\mathcal{C}} (q'_{j,u}, (q'_0, c_i))$, and thus $r = r_u \in \Sigma_{ret}$ and $u \in S_i$ as $(q'_{j,u}, (q'_0, c_i)) \xrightarrow{r_u}_{\mathcal{C}} (q'_f, \perp)$ is the only transition which reaches $q'_f$ from $q'_{j,u}$. Hence $w \in \mathcal{L}(\mathcal{C}) \Leftrightarrow w \in L_I$.

Now assume that K-s-min-bvpa has a solution for $(\mathcal{B}, k+1, s+1)$ and name $\mathcal{A}$ such a solution, i.e. $\mathcal{A} = (Q, q_0, Q_F, \Delta)$ is a $(k+1)$-modules Bvpa with each module size bounded by $s$ and it recognizes $L_I$. Since $\varepsilon \notin L_I$, $\mathcal{A}$ needs at least two states for the empty stack module, an initial state $q_0$ and a final one $q_f$. Also there cannot be any loop on any states, so we can split $Q$ into three sets $Q_0, Q_1, Q_2$ s.t. $Q_0 = \{q_0, q_f\}$ (we can assume there is only one final state). $Q_1$

[5]Note that $j$ exists and is unique since $X$ is a partition of $[1..n]$.

is the set of states reached after having read the first letter and $Q_2$ the second letter. Between $Q_1$ and $Q_2$ there can be only local transitions, so states in $Q_2$ reachable from a state $q \in Q_1$ belong to the same module as $q$. Thus every state in $Q_1$ determines a different module, every state in $Q_2$ belongs to one of these modules, and $Q_0$ is the empty stack module. This directly implies $|Q_1| \leq k$, and thus we can define $Q_1 = \{q_1, q_2, ..., q_k\}$.

Let $X = X_1, X_2, ..., X_k$ be a partition of $[1..n]$ such that $\forall 1 \leq j \leq k$, $X_j = \{i \mid (q_0, \perp) \xrightarrow{c_i}_{\mathcal{A}} (q_j, (q_0, c_i))\}$. Remark this partition is well defined as $\mathcal{A}$ is deterministic and of course each $c_i \in \Sigma_{call}$ has to follow a transition of $\mathcal{A}$ as they all appear in at least one word of $L_I$. There remains one last thing to prove, that $X$ is a solution for $I$, formally that $\forall j \in [1..k]$, $|\bigcup_{i \in X_j} S_i| \leq s$.

Assume $\exists j \in [1..k]$ s.t. $|\bigcup_{i \in X_j} S_i| > s$. As the modules of $\mathcal{A}$ are bounded by $s+1$, $\exists u_1, u_2 \in \bigcup_{i \in X_j} S_i$ such that $u_1 \neq u_2$, $(q_j, (q_0, c_i)) \xrightarrow{u_1}_{\mathcal{A}} (q', (q_0, c_i))$ and $(q_j, (q_0, c_i)) \xrightarrow{u_2}_{\mathcal{A}} (q', (q_0, c_i))$ with $q' \in Q_2$, $[q'] = [q_j]$ and $i \in X_j$.

Also the words $c_i.u_1.r_{u_1}$ and $c_i.u_2.r_{u_2}$ have to be recognized so we must have $(q', (q_0, c_i)) \xrightarrow{u_1}_{\mathcal{A}} (q_f, \perp)$ and $(q', (q_0, c_i)) \xrightarrow{u_2}_{\mathcal{A}} (q_f, \perp)$. This implies that the word $c_i.u_1.r_{u_2}$ (as well as $c_i.u_2.r_{u_1}$) is also recognized, thus $\mathcal{L}(\mathcal{A}) \neq L_I$ and it leads to a contradiction.

$\square$

## 4.4 General minimization

The minimization problem introduced in the last section can be seen as a rather difficult problem compared to the general minimization. Indeed, the traditional minimization asks only to minimize the number of states of an automaton, contrary to K-S-MIN-BVPA that forces both the number of modules and the size of each module to be minimal. This problem is in fact much harder that the normal minimization as we can reduce one to the other problem, but the converse is not known yet. Let us show here this reduction.

**Definition** (MIN-BVPA)

INSTANCE: A BVPA $\mathcal{B}$ and an integer $k$.

QUESTION: Is there a BVPA $\mathcal{A}$, such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ and $|\mathcal{A}| \leq k$ ?

**Proposition 4.4.1** MIN-BVPA $\leq$ K-S-MIN-BVPA.

**Proof** Let $(\mathcal{A}, k)$ be an instance of MIN-BVPA. $(\mathcal{A}, k)$ is a positive instance iff there exists a BVPA using between 1 and $k$ modules that recognizes $\mathcal{L}(\mathcal{A})$. If such a BVPA uses $i \in [1..k]$ modules, then each module cannot have more than $\frac{k}{i}$ states.

Hence, we can call $k$ times K-S-MIN-BVPA with input $(\mathcal{A}, i, \frac{k}{i})$ for $1 \leq i \leq k$. If one of those calls answers positively, then we ensure that a $k$-states BVPA exists. If we get only negative answers, then no $k$-states BVPA exist.

$\square$

# 5 Minimization of variants of VPA and subclasses of VPL

We have previously seen that it is computationally hard to minimize Bvpa. Although Bvpa are close in succinctness to Vpa, their minimization seems as hard as in the case of Vpa. Apart from Bvpa, several other variants of Vpa have been introduced in order to achieve a polynomial-time minimization. Among those variants, Single-Entry Vpa [8] and Multi-Entry Vpa [3] have been introduced to model different kind of recursive programs. Both these variants have been shown to be special cases of a larger model: the Call Driven Automata (Cda) introduced in [14]. All these variants share with Bvpa some module partitioning of the states, and for instance in the Cda case each module represents a set of call symbols and each time the automaton reads some call symbol $c$, it has to go inside the corresponding module, hence the name of *Call Driven*.

The minimization of Cda is obtained through the minimization of a simpler pushdown machine: Expanded Call Driven Automaton (Ecda). This model has been introduced by Chevret and Walukiewicz in [14] to serve as a simple basic Vpa model in order to show minimization of more complex models such as Sevpa, Mevpa and Cda. Unfortunately, although Ecda can be minimized within polynomial time, they are exponentially less succinct that both Bvpa and Vpa, i.e. there exist Vpl such that a minimum Ecda recognizing it is of size $\mathcal{O}(2^k)$ whereas minimum Bvpa and Vpa are of size $\mathcal{O}(k)$.

All these variants have shown to be either difficult to minimize (e.g. finding the optimal partition of Bvpa) or easy to minimize[1] but the minimal automaton can be exponentially larger than the minimal Vpa (e.g. Cda, Sevpa, Mevpa and Ecda). We investigate in Section 5.2, models of automata that are simultaneously easy to minimize and do not show any blow-up against minimal Vpa at the cost of expressiveness, i.e. automata that recognize subclasses of Vpl.

In Section 5.1, we will describe one of the general variants of Vpa which is Ecda and we will introduce a direct translation from Vpa to Ecda. This translation can be used to find a canonical representative of a Vpl as well as giving a translation between these two classes that does not use congruences. Then, we will extend this translation to the general case of Cda. In Section 5.2, we will focus on several subclasses of Vpl in order to explore where the minimization of Vpa seems to start to be intractable. We will conclude with Section 5.3 by giving a reduction from one subclass of Vpa to a minimization problem on finite state machines. This reduction is intended to show the actual difficulty in establishing the tractability frontier for Vpa minimization.

[1]We use here the traditional computability analysis sense of *easy*, i.e. within polynomial time.

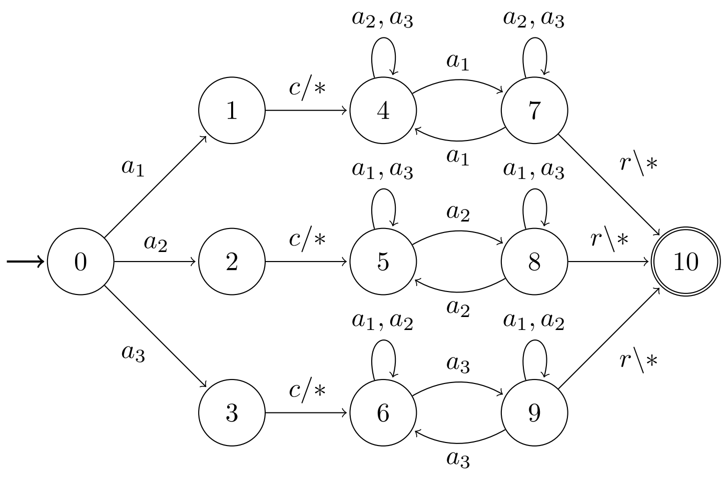

Figure 5.1: The minimal VPA recognizing the VPL $L_3 = \{a_i.c.L_{a_i}.r \mid 1 \leq i \leq 3\}$ over the pushdown alphabet $\hat{\Sigma} = (\{c\}, \{r\}, \{a_1, a_2, ..., a_n\})$, where $L_{a_i}$ is the regular language over $\Sigma_{loc} = \{a_1, a_2, ..., a_n\}$ that contains all words with an even number of $a_i$.

## 5.1 Direct translations

### 5.1.1 From Vpa to Ecda

An Expanded Call Driven Automaton is a BVPA where each call symbol has its own dedicated module. Traditionally, in BVPA a module can be characterized by a set of call-transitions (i.e. pairs of the form $(q, c) \in Q \times \Sigma_{call}$) that end in the entry-state of the module. In ECDA, the call symbol determines uniquely which is the module to invoke and therefore the automaton has always $|\Sigma_{call}|+1$ modules: one for each call symbol, and one for the empty stack.

**Definition** A BVPA $\mathcal{A} = (Q, q_i, Q_F, \Delta)$ over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ is an ECDA if $\Delta$ has the following property:

- for all $(q_1, c_1, q_1'), (q_2, c_2, q_2') \in \Delta_{call}$, $q_1' = q_2' \Leftrightarrow c_1 = c_2$

Since each module can be accessed through a unique call symbol, we can refer to $m_c$ for the module of call symbol $c$, and $m_I$ for the empty stack module.

As stated before, ECDA can be efficiently minimized and moreover contrary to VPA, the minimum ECDA for a given VPL is unique, as the following theorems show.

**Theorem 5.1.1 ([14])** *For every* VPL $L \subseteq WM(\hat{\Sigma})$ *over a pushdown alphabet* $\hat{\Sigma}$*, there exists a unique (up to isomorphism) minimum-state* ECDA *recognizing* $L$.

**Proof *(Sketch)*** The idea of the proof is to define an equivalence relation for each module of the ECDA, then build a syntactic ECDA based on these relations. Precisely, we define the following equivalence relations for every $w_1, w_2 \in WM(\hat{\Sigma})$:

- $w_1 \sim_I w_2$ iff $\forall v \in \Sigma^*, w_1.v \in L \Leftrightarrow w_2.v \in L$
- for all $c \in \Sigma_{call}$, $w_1 \sim_c w_2$ iff $\forall u, v \in \Sigma^*, u.c.w_1.v \in L \Leftrightarrow u.c.w_2.v \in L$

Note that $\sim_I$ is exactly $\equiv$ (defined in Section 3.4, page 25) and all $\sim_c$ include $\equiv$, hence from Theorem 3.4.1 all these relations have finite index. We will write $[w]_I$ for the equivalence class of $w$ based on $\sim_I$ (resp. $[w]_c$ for $\sim_c$).

We can construct now an ECDA that uses as states the equivalence classes of these relations. Formally, we build the ECDA $\mathcal{A} = (Q, q_i, Q_F, \Delta)$ with $Q = \{(a, [w]_a) \mid a \in \Sigma_{call} \cup \{I\} \wedge w \in WM(\hat{\Sigma})\}$, $q_i = (I, [\varepsilon]_I)$, $Q_F = \{(I, [w]_I) \mid w \in L\}$ and $\Delta$ is defined as follows for all $a \in \Sigma_{call} \cup \{I\}$:

- for all $c \in \Sigma_{call}$, $((a, [w]_a), c, (c, [\varepsilon]_c)) \in \Delta_{call}$
- for all $r \in \Sigma_{ret}$, $c \in \Sigma_{call}$, $((c, [w]_c), r, ((a, [w']_a), c), (a, [w'.c.w.r]_a)) \in \Delta_{ret}$
- for all $l \in \Sigma_{loc}$, $((a, [w]_a), l, (a, [w.l]_a)) \in \Delta_{loc}$

We can easily check that $\mathcal{A}$ is well-defined and that $\mathcal{A}$ is an ECDA (the module of state $(a, [w]_a)$ is simply $a$). Moreover, this ECDA can be shown to be the unique (up to isomorphism) minimal automaton that recognizes $L$ as a homomorphism between any ECDA recognizing $L$ and $\mathcal{A}$ can be exhibited. $\square$

**Theorem 5.1.2 ([14])** *For any* ECDA $\mathcal{A}$*, we can compute the minimal* ECDA *recognizing the same language in cubic time.*

We can now describe how to explicitly build an ECDA from any VPA. Of course, the construction can lead to an exponential blow-up but this blow-up cannot be avoided in general. For instance consider the language $L_k$ (defined in Section 3.4, page 26), where if all transitions reading a $\boldsymbol{c}$ lead to the same state then the automaton needs $\mathcal{O}(2^k)$ states, whereas the minimal VPA is of size $\mathcal{O}(k)$. As the input can be any VPA, this construction allows to determinize in the same time as building an equivalent ECDA.

**Theorem 5.1.3** *Given a* (N)VPA $\mathcal{A}$ *over* $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ *recognizing a* VPL $L \subseteq WM(\hat{\Sigma})$*, we can build an* ECDA $\mathcal{E}$ *recognizing the same language such that* $|\mathcal{E}| \leq |\Sigma_{call} + 1| \cdot 2^{|A|^2}$.

**Proof** The main idea is to use a similar construction as for determinization, i.e. we will keep track of summary edges in order to simulate the automaton after reading a call.

Contrary to determinization, we have also to remember in the control state the last unmatched call symbol read (this will indicate what is the current module of the ECDA). We refer the reader to the proof of Theorem 2.1.1 (page 9) for the definitions of $Id$ and $\Pi$.

Let $\mathcal{A} = (Q, Q_I, Q_F, \Gamma, \Delta)$ be a (N)VPA over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. We build the ECDA $\mathcal{E} = (Q', q'_i, Q'_F, \Delta')$ with $Q' = \Sigma_{call} \cup \{m_I\} \times 2^{Q \times Q}$, $q'_i = (m_I, Id_{Q_I})$, $Q'_F = \{(m_I, S) \in Q' \mid \Pi_2(S) \cap Q_F \neq \emptyset\}$. $m_I$ is an extra symbol that designates the initial module. The transition set of the ECDA is as follows:

- $\forall c \in \Sigma_{call}$, $S \in Q'$, $(S, c, (c, Id_{Q_c})) \in \Delta'_{call}$
  $\Leftrightarrow Q_c = \{(q, q) \in Q^2 \mid \exists q' \in Q, \gamma \in \Gamma, (q', c, q, \gamma) \in \Delta_{call}\}$
- $\forall r \in \Sigma_{ret}$, $((c, s), r, ((m, s'), c), (m, s'')) \in \Delta'_{ret}$
  $\Leftrightarrow s'' = \{(q, q') \in Q^2 \mid \exists q_1,\ q_2,\ q_3 \in Q, \gamma \in \Gamma, (q, q_1) \in s' \wedge (q_1, c, q_2, \gamma) \in \Delta_{call} \wedge (q_2, q_3) \in s \wedge (q_3, r, \gamma, q') \in \Delta_{ret}\}$
- $\forall l \in \Sigma_{loc}$, $((m, s), l, (m, s')) \in \Delta'_{loc}$
  $\Leftrightarrow s' = \{(q, q'') \in Q^2 \mid \exists q' \in Q, (q, q') \in s \wedge (q', l, q'') \in \Delta_{loc}\}$

First of all, let us check that $\mathcal{E}$ is an ECDA. For that purpose, associate the state $(m, S) \in Q'$ to the module $m$. This way, no call transitions go to the empty stack module (module $m_I$), all call transition $\boldsymbol{c}$ go to the specific module $\boldsymbol{c}$, all return transitions come back to their original module and at last local transitions do not change the current module. Moreover, each module has a unique entry-state, i.e. $(m_I, Id_{Q_I})$ for the empty stack module and $(c, Id_{Q_c})$ for each module[2] $c$.

Now, we have to verify that $\mathcal{L}(\mathcal{E}) = \mathcal{L}(\mathcal{A})$. It is essentially due to the same reasons as in the determinization (we use the same invariant), but this time the summary-set $S$ in a state $(m, S)$ with $m \neq m_I$ does not contain reachable states. This is mainly due to the fact that we have lost all information when reading a call symbol $c$. But since we have restrained ourselves to well-matched words, we cannot finish in a state of the form $(m, S)$ with $m \neq m_I$.

The only final states are of the form $(m_I, S)$, and every time such a state is reached we know that the stack is empty. This means that the set of summary-edges for final states does only contain reachable states. Precisely, whenever $\mathcal{E}$ reaches a state $(m_I, S)$ after reading a word $w \in \Sigma^*$, then $\Pi_2(S)$ is the set of reachable states for $\mathcal{A}$ after reading $w$.

At last, it is clear that $|\mathcal{E}| \leq |\Sigma_{call} + 1| \cdot 2^{|A|^2}$.

□

**Example** Consider the VPA of Figure 5.1. It is clearly not an ECDA as there exist several call transitions that lead to different states, e.g. $(1, c, 4, *)$, $(2, c, 5, *)$ and $(3, c, 6, *)$.

Once we use the previous construction, the states $0, 1, 2, 3, 10$ become $(m_I, 0)$, $(m_I, 1)$, $(m_I, 2)$, $(m_I, 3)$ and $(m_I, 10)$, the later is the only final state. When reading a $c$ in $(m_I, \{(0, 1)\})$, $(m_I, \{(0, 2)\})$ or $(m_I, \{(0, 3)\})$, we go to the state $(c, \{(0, 1), (0, 2), (0, 3)\})$. Then we have to memorize which are the $a_i$'s that are

[2]Remark that several modules can have the same starting summary-set, i.e. it is possible that $c \neq c'$ and $Q_c = Q_{c'}$. It is mainly for that purpose that we have to keep separate the module and the summary-set.

in even numbers, so we need 8 more states. Each of these states has as much as needed return transitions to $(m_I, 10)$, depending on the parity of the different $a_i$'s.

### 5.1.2 From Vpa to Cda

*Call Driven Automata* (CDA) are an extension of ECDA that generalize intermediate models such as SEVPA and MEVPA. The minimization of CDA can be done through the minimization of ECDA (see [14]) since from any CDA we can build an ECDA of size no more that $|\Sigma_{call}|$ times larger, where $\Sigma_{call}$ is the input call alphabet. Therefore, we can re-use the translation of Theorem 5.1.3 to build a CDA of size $\mathcal{O}(|\Sigma_{call}|^2 \cdot 2^{n^2})$ from a VPA of size $n$. However, we can build directly a CDA using a similar construction as previously and thus the factor $|\Sigma_{call}|$ can be skipped. This construction is more intended to show how to generalize the previous proposition than of practical use for minimization, since in order to minimize CDA the only known algorithm has to build an ECDA.

CDA are VPA that have a partition of its states into modules, in the same fashion as all variants of VPA seen so far, but rather that a module is specific to a unique call symbol as ECDA, a module can be accessed by a subset of the call alphabet. Moreover unlike BVPA, CDA can have several *entry-states* for each module, thus some CDA are not BVPA. The other rules on the structure of modules and transitions are similar to BVPA and ECDA.

**Definition** A VPA $\mathcal{A} = (Q, q_i, Q_F, Q \times \Sigma_{call}, \Delta)$ over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ is a CDA, if there exists a partition of the states into modules noted [] and the following properties hold:

**Empty stack:** for all $(q, c, q', (q, c)) \in \Delta_{call}$, $[q_i] \neq [q']$

**One target state per call:** for all $(q_1, c_1, q'_1, (q_1, c_1)), (q_2, c_2, q'_2, (q_2, c_2)) \in \Delta_{call}$, $c_1 = c_2 \to q'_1 = q'_2$

**Return to entry module:** for all $(q, r, (q', c), q'') \in \Delta_{ret}$, $[q'] = [q'']$

**Local actions preserve module:** for all $(q, l, q') \in \Delta_{loc}$, $[q] = [q']$

In the same fashion as BVPA, we will omit to write the push symbol in a call transition as it can be inferred from the other parameters.

**Example** The automaton of Figure 4.1, page 28, is a CDA. One can notice it is clearly not a BVPA (number of entry-states).

**Theorem 5.1.4** *Given a* VPA $\mathcal{A}$ *over* $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$ *recognizing a* VPL $L \subseteq WM(\hat{\Sigma})$*, we can build a* CDA $\mathcal{C}$ *recognizing the same language such that* $|\mathcal{C}| \leq 2 \cdot 2^{|A|^2}$.

**Proof** The construction is almost the same than in the case of ECDA but since one module can have several entry-states and represent several call symbols, we do not have to make $|\Sigma_{call}| + 1$ modules, two are enough: one for the empty stack and one for all the call symbols. Also now that the states do not contain the last unmatched call, we have to push it on the stack.

Let $\mathcal{A} = (Q, Q_I, Q_F, \Gamma, \Delta)$ be a NVPA over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. We build the CDA $\mathcal{C} = (Q', q'_i, Q'_F, \Delta')$ with $Q' = \{m_I, m_{call}\} \times 2^{Q\times Q}$, $q'_i = (m_I, Id_{Q_I})$, $Q'_F = \{(m_I, S) \in Q' \mid \Pi_2(S) \cap Q_F \neq \emptyset\}$ and the transition function is as follows:

- $\forall c \in \Sigma_{call}$, $S \in Q'$, $(S, c, (c, Id_{Q_c}), (S, c)) \in \Delta'_{call}$
  $\Leftrightarrow Q_c = \{(q, q) \in Q^2 \mid \exists q' \in Q, (q', c, q, \gamma) \in \Delta_{call}\}$.
- $\forall r \in \Sigma_{ret}$, $((m_0, s), r, ((m, s'), c), (m, s'')) \in \Delta'_{ret}$
  $\Leftrightarrow s'' = \{(q, q') \in Q^2 \mid \exists q_1, q_2, q_3 \in Q, \gamma \in \Gamma, (q, q_1) \in s' \wedge (q_1, c, q_2, \gamma) \in \Delta_{call} \wedge (q_2, q_3) \in s \wedge (q_3, r, \gamma, q') \in \Delta_{ret}\}$
- $\forall l \in \Sigma_{loc}$, $((m, s), l, (m, s')) \in \Delta'_{loc}$
  $\Leftrightarrow s' = \{(q, q'') \in Q^2 \mid \exists q' \in Q, (q, q') \in s \wedge (q', l, q'') \in \Delta_{loc}\}$

The modules of $\mathcal{C}$ are $m_I$ for the empty stack and $m_{call}$ for all call symbols. See the proof of Theorem 5.1.3 for the correctness of the construction. □

Note the well-matched constraint cannot be avoided in the construction since when reading a call symbol, the automaton has to move to a unique state whatever was read before. Thus the set of summary edges is *accurate* (i.e. for the set $S$, $\Pi_2(S)$ is exactly the set of reachable states) only when no more call have been postponed and so the stack is empty.

Also when the input VPA $\mathcal{A}$ is deterministic, we can reduce the module of the empty stack to the original states of $\mathcal{A}$ and thus the CDA would be of size $|A| + 2^{|A|^2}$. In fact, the corresponding CDA has just to remember what state is reached after reading a return symbol that comes back to the empty stack module. Since $\mathcal{A}$ is deterministic, a single state can be reached.

## 5.2 Stack-depth 1 subclass

Since a minimization algorithm is not yet known for VPA or any of its variants that are polynomially close in size, we shall try to focus on subclasses of VPL instead of trying to constrain the structure of VPA (for instance by assuming a modular structure). We will explore here one of the simplest subclass of VPA, the class of languages that use only a single space in the stack. All these languages are trivially regular, however the minimal VPA can be smaller than the minimal DFA recognizing such languages up to a factor $|\Gamma|$, where $\Gamma$ is the stack alphabet of the VPA.

**Proposition 5.2.1** *Let $A$ be a* DFA *over $\Sigma_{loc}$ and let $L_A = \{c.w.r \mid w \in \mathcal{L}(A)\}$ be a* VPL *over $\hat{\Sigma} = (\{c\}, \{r\}, \Sigma_{loc})$. We can find in polynomial time the smallest* VPA *recognizing $L_A$.*

**Proof** Let $A' = (Q, q_0, Q_F, \delta)$ be the minimal DFA for $L_A$. We can build the VPA $\mathcal{C} = (Q \cup \{q_i, q_f\}, q_i, \{q_f\}, \{*\}, \Delta)$ such that

- $(q_i, c, q_0, *) \in \Delta_{call}$
- $(q, l, q') \in \Delta \Leftrightarrow \delta(q, l) = q'$
- $\forall q \in Q_F,\ (q, r, *, q_f) \in \Delta_{ret}$

Obviously a word $w$ is accepted if and only if it is of the form $c.w'.r$ with $w' \in \mathcal{L}(A)$, and thus $\mathcal{C}$ recognizes $L_A$. This VPA can be computed in time $\mathcal{O}(|A| \cdot |\Sigma_{loc}| \cdot \log |A|)$ using a well known algorithm on DFA minimization [22].

Now let us consider a minimal VPA $\mathcal{C}' = (Q', q'_i, Q'_F, \Gamma', \Delta')$ for $L_A$. $\mathcal{C}'$ must first check that we read only a single $c$ and thus $\mathcal{C}'$ must have indeed a state $q'_0$ different than $q'_i$, and the transition[3] $(q'_i, c, q'_0, (q'_i, c))$. Also, note that we cannot allow a *back-transition* from $q'_0$ to $q'_i$ since it would allow $\mathcal{C}'$ to accept words that contain more than one symbol $c$. So $q'_i$ has only one outgoing transition, and we know that all states reachable from $q'_0$ should not have any outgoing call transitions, hence using Proposition 3.1.2 (page 17), we have that $\Gamma'$ is reduced to the symbol $(q'_i, c)$. Also, $\mathcal{C}'$ needs a final state different that $q'_i, q'_0$ for obvious reasons, let us call it $q'_f$. Let $Q'_F$ denote the set $\{q' \in Q' \mid (q', r, (q'_i, c), q'_f) \in \Delta'_{ret}\}$. Note $q'_f \notin Q'_F$, otherwise $r$ would not be the last symbol read, or several $r$ could have been read. Finally, $|\mathcal{C}'| \geq |A'| + 2$, as otherwise we can build a DFA that recognizes $\mathcal{L}(A)$ smaller than $|A'|$ using $\mathcal{C}'$ and $Q'_F$.

□

We can slightly improve Proposition 5.2.1 by increasing the number of call or return symbols, as long as we do not force any *relation* between calls and returns.

**Proposition 5.2.2** *Let $A$ be a* DFA *over $\Sigma_{loc}$ and let $L_A = \{c.w.r \mid w \in \mathcal{L}(A), c \in \Sigma_{call}, r \in \Sigma_{ret}\}$ be a* VPL *over $\hat{\Sigma} = (\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. We can find in polynomial time the smallest* VPA *recognizing $L_A$.*

**Proof** The proof is identical to the previous one, but in this case more stack symbols can be produced after reading the first call symbol (exactly $|\Sigma_{call}|$). Anyway, since when reading the last return letter all possible stack symbols have to lead to the final state, they are equivalent and one can build the resulting VPA such that it uses a unique call symbol.

□

So far, we have reduced automata where the stack symbol put on the stack was completely irrelevant, and thus a single *dummy* symbol was enough. An interesting question likely to arise is whether VPA and DFA are equivalent model in term of space consumption when the stack alphabet is reduced to 1. Another constraint can be that only well-matched words are considered, as otherwise we

[3]Precisely, if $\mathcal{C}'$ loops on $q'_i$ when reading $c$, the automaton would have no way to distinguish the case that it has read only one symbol, compare to the case where it has read several $c$'s.

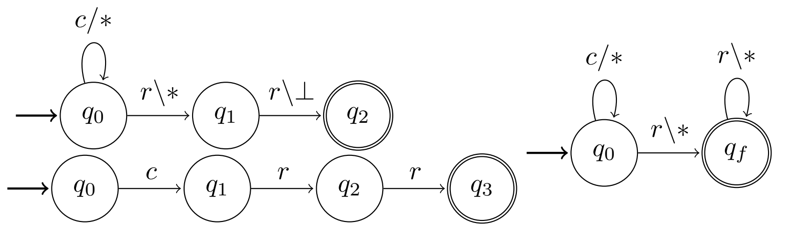

Figure 5.2: On the left, a minimal VPA recognizing $L = \{c.r^2\}$ over $(\{c\}, \{r\}, \emptyset)$ (above) and the minimal DFA that recognizes the same language (below). On the right, a minimal VPA recognizing $L = \{c^n\, r^m \mid n \in \mathbb{N}^+, m \in \mathbb{N}^+, m \leq n\}$.

can use the special bottom symbol to save some states compare to a DFA as illustrated in Figure 5.2 (left). Naturally, if the stack is not reduced to a single space, non-regular languages can be recognized (see Figure 5.2, right).

**Proposition 5.2.3** *Every* VPA $\mathcal{A}$ *over* $\hat{\Sigma}$ *s.t.* $\mathcal{L}(\mathcal{A}) \subseteq WM(\hat{\Sigma})$, $\mathcal{A}$ *uses only one stack symbol and one stack space is minimizable.*

**Proof** Let $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta)$ be a VPA with the same assumptions that in the proposition (i.e. $|\Gamma| = 1$ and $\forall v \in \mathcal{L}(\mathcal{A}), \forall u \in \text{pref}(v)$, $u \in MR(\hat{\Sigma})$ and the number of unmatched calls in $u$ is at most 1). W.l.o.g we assume that all states of $Q$ are reachable and co-reachable [4] and that the unique symbol of $\Gamma$ is $*$.

First notice that $\mathcal{A}$ cannot use transitions of the form $(q, r, \bot, q')$ since $\mathcal{L}(\mathcal{A})$ is well-matched. Second remark that the states can be divided into two sets $Q_0$ and $Q_1$ such that:

1. $q_i, q_f \in Q_0$
2. $\forall (q, c, q', *) \in \Delta_{call}$, $q \in Q_0$ and $q' \in Q_1$
3. $\forall (q, r, q', *) \in \Delta_{ret}$, $q \in Q_1$ and $q' \in Q_0$
4. $\forall (q, l, q') \in \Delta_{loc}$, $q \in Q_0 \Leftrightarrow q' \in Q_0$

The above statement is correct as we can always set $Q_0$ as the set of states reachable with an empty stack and $Q_1$ its complement against $Q$. This partition is sound as all states are reachable and co-reachable.

Now consider the DFA $A = (Q, q_i, Q_F, \delta)$ over $\Sigma^*$ defined as $\delta(q, a) = q' \Leftrightarrow a \in \Sigma_{call}, (q, a, q', *) \in \Delta_{call} \;\vee\; a \in \Sigma_{ret}, (q, a, *, q') \in \Delta_{ret} \;\vee\; a \in \Sigma_{loc}, (q, a, q') \in \Delta_{loc}$. $A$ recognizes exactly $\mathcal{L}(\mathcal{A})$ as whenever $A$ is in state $q$, if $q \in Q_0$ then the configuration of $\mathcal{A}$ must be $(q, \bot)$ and if $q \in Q_1$, the configuration of $\mathcal{A}$ must be $(q, \bot *)$.

Hence minimizing $A$ will keep this partition of the states, as the minimization will only merge the states $q$ and $q'$ if they are both in $Q_0$ or both in $Q_1$. Thus we can easily construct a VPA based on the minimal DFA using the visibly

[4] Otherwise we can eliminate unreachable states in polynomial time.

pushdown alphabet. This VPA is minimal as otherwise we can apply the same construction as above and find a smaller DFA than the minimal one.

$\square$

This technique starts to present some severe limitations when we start to generalize a little more. For instance, if the call symbol *influences* what should be the last return symbol, then the VPA recognizing such language is generally much smaller than a DFA, as it can uses its one memory space to remember what was such a symbol and reuse it at the end. An example of such a language can be the VPL $L = \{c_i\, r_i \mid i \in [1..k]\}$ for some integer $k$ over the pushdown alphabet $\hat{\Sigma} = (\{c_1, c_2, ..., c_k\}, \{r_1, r_2, ..., r_k\}, \emptyset)$. We can easily build a VPA of size 3 but any DFA has to remember what was the original call symbol in its state since it has no other kind of memory, and therefore its minimum size is $k + 2$. However, we can still minimize such automaton by carefully considering couple $(c, r) \in \Sigma_{call} \times \Sigma_{ret}$ such that $c.L.r \subseteq L_A$, with $L$ being a fixed regular language.

**Proposition 5.2.4** *Let $A$ be a* DFA *over $\Sigma_{loc}$ and let $L_A = \bigcup_{(c,r)\in X} L_{c,r}$ with $X \subseteq \Sigma_{call} \times \Sigma_{ret}$ and $L_{c,r} = \{c\, w\, r \mid w \in \mathcal{L}(A)\}$ be a* VPL *over $(\Sigma_{call}, \Sigma_{ret}, \Sigma_{loc})$. We can find in polynomial time the smallest* VPA *recognizing $L_A$.*

**Proof** The proof is similar to the one of Proposition 5.2.1 and 5.2.2, but this time several stack symbols are needed. The idea is to minimize the original DFA $A$ (let $A_{min} = (Q, q_0, Q_F, \Delta)$ be a minimal DFA for $\mathcal{L}(A)$) then build a VPA $\mathcal{A} = (Q', q_i, Q'_F, \Delta)$ such that $Q' = Q \cup \{q_i, q_f\}$ and $Q'_F = \{q_f\}$. $\mathcal{A}$ has the same transitions as $A$, plus for all $c \in \Sigma_{call}$ $(q_i, c, q_0, c)$ and for all $q \in Q_F$, $r \in \Sigma_{ret}$, $(q, r, c, q_f)$ s.t. $L_{c,r} \subseteq L$.

For the same reasons as previously, any automaton needs at least two extra states compare to the minimal DFA.

$\square$

## 5.3 Difficult subclasses

In the previous section, we have seen some simple subclasses of VPL, where we can exhibit a polynomial time minimization algorithm. When we try to generalize a bit more, we face minimization problems that we do not know how to solve efficiently. So far, languages considered were VPL that includes a fixed regular language on the local alphabet. If the regular language depends on the call read, then the minimization problem starts to be difficult.

For instance, consider languages $L$ such that $L \subseteq \Sigma_{call}\Sigma^*_{loc}\Sigma_{ret}$, where $\Sigma_{ret}$ can be reduced to a single return symbol $r$. Such languages are regular and are included in the class of stack depth 1 VPL, but still we do not know how to find a minimal VPA that recognizes them[5]. Let us reformulate this problem into a finite automaton problem.

[5]Note if $|\Sigma_{call}| = 1$, the problem becomes straightforward as stated in Proposition 5.2.2.

**Definition** A $n$ Sets Deterministic Finite Automaton ($n$-Sdfa) $\mathcal{S}$ over a finite alphabet $\Sigma$ is a tuple $(Q, \alpha_i, \alpha_f, \delta)$, such that $Q$ is a finite set of states, $n \in \mathbb{N}$, $\alpha_i : [1..n] \to Q$ is the initial state function, $\alpha_f : [1..n] \to 2^Q$ is the final state function and $\delta : Q \times \Sigma \to Q$ is the (partial) transition function of $\mathcal{S}$.

Intuitively, such automaton recognizes $n$ different regular languages with the same set of states. A tuple $(u, k) \in \Sigma^* \times [1..n]$ is accepted by an $n$-Sdfa if the word $u$ is accepted by the Dfa obtained by applying $\alpha_i$ and $\alpha_f$ on $k$, i.e. the Dfa $A_k = (Q, \alpha_i(k), \alpha_f(k), \delta)$. Analogously to other classes of automata, $\mathcal{L}(\mathcal{S})$ is the set of such accepted tuples. For any $n$-Sdfa $\mathcal{S}$, we denote $\mathcal{L}_k(\mathcal{S}) = \{u \in \Sigma^* \mid (u, k) \in \mathcal{L}(\mathcal{S})\}$ the $k^{th}$ language recognized by $\mathcal{S}$.

This kind of automaton can be used in practice to store efficiently large number of regular languages, especially when these languages share *common* subautomata. A concrete example can be when one has to check the type of a natural word $u$ (and secondarily that $u$ is well spelled). For instance, two words such as *antidisestablishmentarianism* and *antiestablishment* will share a lot of states and transitions in a Sdfa, nevertheless the former belongs to $L_{noun}$ and the latter to $L_{adjective}$.

Now, we want to exhibit that minimizing some of the simplest subclasses of Vpl is an equivalent problem to the minimization of Sdfa. Hence, either one can find an efficient minimization algorithm for Sdfa that provides directly a minimization algorithm for such subclasses of Vpa, or minimizing Sdfa is computationally hard thus minimization of all Vpa is hard.

**Definition** (Min-Simple-Vpa)

Instance: A Vpa $\mathcal{A}$ over $(\Sigma_{call}, \{r\}, \Sigma_{loc})$ with $\mathcal{L}(\mathcal{A}) \subseteq \Sigma_{call}\Sigma_{loc}^*\Sigma_{ret}$ and a positive integer $k < |\mathcal{A}|$.

Question: Is there a Vpa $\mathcal{B}$ of size $k$, such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ ?

**Definition** (Min-Sdfa)

Instance: $n$ Dfa $A_1, A_2, ..., A_n$ and a positive integer $k < |\mathcal{S}|$.

Question: Is there a $n$-Sdfa $\mathcal{S}$ of size $k$, such that $\mathcal{L}_i(\mathcal{S}) = \mathcal{L}(A_i)$ for all $1 \leq i \leq n$ ?

**Proposition 5.3.1** *Given $n$ regular languages $L_1, ..., L_n \subseteq \Sigma_{loc}^*$, there exists an $n$-*Sdfa *$\mathcal{S}$ of size $k$, such that $\mathcal{L}_i(\mathcal{S}) = L_i$ for all $1 \leq i \leq n$, iff there exists a* Vpa *$\mathcal{A}$ over $(\{c_j \mid j \in [1..n]\}, \{r\}, \Sigma_{loc})$ of size $k+2$ that recognizes $\bigcup\limits_{1 \leq j \leq n} c_j L_j r$.*

**Proof** Let $L_1, ..., L_n \subseteq \Sigma_{loc}^*$ be $n$ regular languages.

$\Rightarrow$) Let $\mathcal{S} = (Q, \alpha_i, \alpha_f, \delta)$ be an $n$-Sdfa of size $k$ over $\Sigma_{loc}$ as in the proposition (i.e. $\mathcal{L}_i(\mathcal{S}) = \text{Ł}_i$ for $i \in [1..n]$). We build the following Vpa $\mathcal{A} = (Q', q_i', Q_F', \Gamma, \Delta)$ over $(\Sigma_{call}, \{r\}, \Sigma_{loc})$ with $\Sigma_{call} = \{c_j \mid j \in [1..n]\}$, such that $Q' = Q \cup \{q_i, q_f\}$, $Q_F' = \{q_f\}$, $\Gamma = [1..n]$ and its transitions are:

- for all $c_j \in \Sigma_{call}$, $(q_i, c_j, \alpha_i(j), j) \in \Delta_{call}$

- for all $q \in Q$, $j \in [1..n]$, if $q \in \alpha_f(j)$ then $(q, r, j, q_f) \in \Delta_{ret}$
- for all $q \in Q$, $l \in \Sigma_{loc}$, if $\delta(q, l) = q'$ then $(q, l, q') \in \Sigma_{loc}$

We have $|\mathcal{A}| = k + 2$, thus it remains just to show $\mathcal{L}(\mathcal{A}) = \bigcup\limits_{1 \leq j \leq n} c_j L_j r$. Consider a word $w \in \Sigma^*$. By looking at the transitions of $\mathcal{A}$ we have that $w \in \mathcal{L}(\mathcal{A}) \Leftrightarrow \exists w' \in \Sigma^*, j \in [1..n], q \in Q, w = c_j.w'.r$ and $(\alpha_i(j), j) \xrightarrow{w'}_{\mathcal{A}} (q, j)$ with $q \in \alpha_f(j)$. Since all states reached after reading $c_j$ and before reading $r$ are in $Q$, only transitions of $S$ can be applied, thus $w' \in \Sigma_{loc}$ and $\alpha_i(j) \xrightarrow{w'}_S q$. Finally, since $q \in \alpha_f(j)$, $w' \in L_j$.

$\Leftarrow$) Let $\mathcal{A} = (Q, q_i, Q_F, \Gamma, \Delta)$ be a VPA of size $k+2$ that recognizes $\bigcup\limits_{1 \leq j \leq n} c_j L_j r$. For the same arguments used in the earlier proof (see for instance Proposition 5.2.2), $\mathcal{A}$ must have two distinct states one initial $q_i$ with no ingoing transitions and one final $q_f$ with no outgoing transitions (we can also assume $Q_F = \{q_f\}$). Also we can assume $\Gamma$ to be restrained to the set $\Sigma_{call}$ as $q_i$ can be assumed to be the only state with outgoing call transitions.

We construct the following $n$-SDFA $\mathcal{S}' = (Q^{\mathcal{S}}, \alpha_i^{\mathcal{S}}, \alpha_f^{\mathcal{S}}, \delta^{\mathcal{S}})$, such that $Q' = Q \setminus \{q_i, q_f\}$, and

- for all $j \in [1..n]$, $\alpha_i^{\mathcal{S}}(j) = q$ s.t. $(q_i, c_j, q, c_j) \in \Delta_{call}$
- for all $j \in [1..n]$, $\alpha_f^{\mathcal{S}}(j) = \{q \mid \exists (q, r, c_j, q_f) \in \Delta_{ret}\}$
- for all $l \in \Sigma_{loc}$, $\delta(q, l) = q'$ iff $(q, l, q') \in \Sigma_{loc}$

A tuple $(u, j) \in \Sigma_{loc}^* \times [1..n]$ is accepted by $\mathcal{S}'$ iff $u \in \mathcal{L}_j(\mathcal{S}')$, i.e. $c_j\, u\, r \in \mathcal{L}(\mathcal{A})$. Since $\mathcal{L}(\mathcal{A}) = \bigcup\limits_{1 \leq j \leq n} c_j L_j r$, the word $c_j u r$ is accepted by $\mathcal{A}$ iff $u \in L_j$.

□

**Corollary 5.3.2** MIN-SIMPLE-VPA *and* MIN-SDFA *are equivalent problem.*

We do not know how to solve MIN-SDFA, but we suspect the problem to be NP-hard. If one shows the hardness of MIN-SDFA then we have a hardness proof for MIN-SIMPLE-VPA based on the previous corollary. This will have great consequences, as it implies that VPA are not minimizable within polynomial time unless P=NP. We leave this remark as a conjecture.

**Conjecture** MIN-SIMPLE-VPA is NP-complete.

# 6 Conclusion

We have studied the minimization problem for a recently introduced automaton model: *Visibly Pushdown Automata*. This abstract model is used in a wide range of areas and an efficient minimization algorithm would have direct consequences on several of their applications.

The minimization problem for VPA is suspected to be computationally hard but we failed to prove the hardness of the problem. Instead we have given several arguments to guide a future NP-hardness proof and have shown that the minimization problem for non-deterministic VPA is EXPTIME-complete, although it is a less interesting question than in the deterministic case. Also, we have investigated minimization on BVPA, a variant of VPA. We have demonstrated that modular minimization of BVPA is NP-complete, which can lead to a hardness proof for general BVPA.

Apart from hardness of minimization, we have introduced a direct translation between VPA and one of their only minimizable variants, i.e. ECDA. This translation has also been extended in order to suit to CDA, a more generic model. This translation can be used to compute a unique (up to isomorphism) minimal ECDA from any VPL directly from a VPA. At last, we have explored several subclasses of VPA and have proposed polynomial algorithms when possible. Also, since some subclasses seem hard to minimize, we have given a reduction to minimization of some finite state machines (called SDFA). This reduction can lead to a hardness result for VPA or extend the subclasses of VPA where polynomial time minimization algorithms are known.

Our work can be extended by several ways. First, our hardness result on BVPA can be generalized to the traditional minimization. Second, minimization of VPA can be shown to be computationally hard with further work on minimizing particular submodels. This work has also introduced a new model of finite automata, and the complexity of their minimization is an open question (it is in fact equivalent to minimizing a subclass of VPL). At last, we think that we can find more polynomial algorithms for subclasses of VPL by bounding some parameters of VPA, e.g. the number of call symbols used or the maximal stack depth.

# Bibliography

[1] P. Madhusudan. Visibly Pushdown Automata - Automata on Nested Words. `http://www.cs.uiuc.edu/~madhu/vpa/`.

[2] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM.

[3] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Minimization, learning, and conformance testing of boolean programs. In Proceedings of the 17th international conference on Concurrency Theory, CONCUR'06, pages 203–217, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] Corin Pitcher. Visibly pushdown expression effects for xml stream processing. In Programming Language Technologies for XML, pages 1–14, 2005.

[5] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly pushdown automata for streaming xml. In Proceedings of the 16th international conference on World Wide Web, WWW '07, pages 1053–1062, New York, NY, USA, 2007. ACM.

[6] John E. Hopcroft and Jeffrey D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[7] Michael R. Garey and David S. Johnson. Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA, 1990.

[8] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In Proceedings of the 32nd international conference on Automata, Languages and Programming, ICALP'05, pages 1102–1114, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. pages 113–130. Springer, 2000.

[10] Christof Löding, P. Madhusudan, and Olivier Serre. Visibly pushdown games. In Proceedings of the 24th international conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'04, pages 408–420, Berlin, Heidelberg, 2004. Springer-Verlag.

[11] Andrzej S. Murawski and Igor Walukiewicz. Third-order idealized algol with iteration is decidable. Theor. Comput. Sci., 390(2-3):214–229, January 2008.

[12] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. In In Developments in Language Theory, LNCS 4036, pages 1–13. Springer, 2006.

[13] Nguyen Van Tang. A tighter bound for determinization of visibly pushdown automata. In Proceedings of the 11th International Workshop on Verification of Infinite-State Systems, NFINITY'09, 2009.

[14] Patrick Chervet and Igor Walukiewicz. Minimizing variants of visibly pushdown automata. In MFCS, pages 135–146, 2007.

[15] Tao Jiang and Bala Ravikumar. Minimal nfa problems are hard. In Proceedings of the 18th International Colloquium on Automata, Languages and Programming, ICALP '91, pages 629–640, London, UK, UK, 1991. Springer-Verlag.

[16] Pavel Labath and Branislav Rovan. Simplifying dpda using supplementary information. In Proceedings of the 5th international conference on Language and automata theory and applications, LATA'11, pages 342–353, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages and Computability. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[18] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time(preliminary report). In Proceedings of the fifth annual ACM symposium on Theory of computing, STOC '73, pages 1–9, New York, NY, USA, 1973. ACM.

[19] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[20] Stephen A. Cook. An observation on time-storage trade off. J. Comput. Syst. Sci., 9(3):308–316, December 1974.

[21] Wim Martens and Joachim Niehren. On the minimization of xml schemas and tree automata for unranked trees. J. Comput. Syst. Sci., 73(4):550–583, June 2007.

[22] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.