# The Language Theory of Bounded Context-Switching

S. La Torre[1], P. Madhusudan[2], and G. Parlato[1,2]

[1] Università degli Studi di Salerno, Italy
[2] University of Illinois at Urbana-Champaign, USA

**Abstract.** Concurrent compositions of recursive programs with finite data are a natural abstraction model for concurrent programs. Since reachability is undecidable for this class, a restricted form of reachability has become popular in the formal verification literature, where the set of states reached within $k$ context-switches, for a fixed small constant $k$, is explored. In this paper, we consider the language theory of these models: concurrent recursive programs with finite data domains that communicate using shared memory and work within $k$ round-robin rounds of context-switches, and where further the stack operations are made visible (as in visibly pushdown automata). We show that the corresponding class of languages, for any fixed $k$, forms a robust subclass of context-sensitive languages, closed under all the Boolean operations. Our main technical contribution is to show that these automata are *determinizable* as well. This is the first class we are aware of that includes non-context-free languages, and yet has the above properties.

## 1 Introduction

Concurrent threads with recursive procedures that communicate using shared memory is a natural and attractive model, as it models concurrent imperative programs naturally. While message-passing is more common in the distributed computing world where processes run on different machines, the advent of multi-core computing has led to an increased interest in shared-memory programs.

In the methodology of model-checking for program verification, a common paradigm is to analyze a program by verifying a model of it, where the model is obtained by abstracting or simplifying the *data-domain* used by a program, but preserving control flows accurately. Many program analysis frameworks, such as data-flow analysis or predicate abstraction, fall under this category. Hence concurrent recursive programs where variables range over a finite data-domain are an attractive model of study.

The automata-theoretic model of a concurrent program with recursion and shared-memory communication is simply an automaton with multiple stacks. (Note that in such a model, the shared memory, and hence the communication between processes, resides in the control state of the automaton.) Since Turing machines can be simulated by an automaton with two stacks, the *emptiness* problem for these automata, and hence the model-checking problem, is undecidable.

A. López-Ortiz (Ed.): LATIN 2010, LNCS 6034, pp. 96–107, 2010.

In order to overcome this barrier, a recent proposal is to search only the space reached by these automata using a bounded number of context-switches. In other words, we view the computation as occurring in $k$ consecutive stages (for a fixed constant $k$), where in each stage only one of the concurrent threads is active. This restriction in the automaton model translates to restricting the computation to $k$ stages, where in each stage, only one of the stacks is manipulated. It turns out that in this model the reachability problem is decidable (for any fixed $k$) [16].

The idea of checking and testing concurrent programs under a context-switching bound has gained attention in recent years for several reasons. First, there is an intuitive appeal that one expects most concurrency errors to manifest themselves within a few context-switches. This has been argued fairly effectively in recent experimental studies (see [14]). Second, the model-checking problem for bounded context-switching is decidable [16], thus yielding exact algorithmic methods to solve the reachability problem. And third, checking concurrent programs under a context-bound can be done *compositionally*— we can search the state space by avoiding to build explicitly the product of local states of all automata, and instead work with only the space defined by a single thread and $k$ copies of shared variables. The last aspect is in fact a very appealing (and the least articulated) aspect of bounded context-switching that has been exploited in recent work: model-checking tools for concurrent Boolean programs have been developed [8,12,17], and translations of concurrent programs to sequential programs have been developed that reduce bounded context-switching reachability to sequential reachability, even for general C-programs [9,12]. In recent work, the above translation has even been used to verify concurrent programs under a context-bound using deductive verification tools for sequential software [11].

In this paper, we undertake a language and automata-theoretic study of the concurrent programs with recursion under a context-switching bound. While research has so far concentrated on the computation of reachable states, we instead look at the *class of languages* accepted by these automata. In doing so, we make the calls and returns to procedures in the concurrent program *visible*— for sequential programs this yields the class of *visibly pushdown automata* [1], which has been shown to define a robust class of context-free languages, and has led to a flurry of research (see [19] for a list of papers in this area).

We consider automata with $n$ stacks, where an execution goes through $k$ *round-robin schedules* of computation, i.e., a round is a fixed sequence of exactly $n$ contexts, one for each stack. In a *context* for a stack $i$, for $i = 1, \ldots, n$, the automaton can only read letters pertaining to stack $i$ and manipulate stack $i$.

The visibility of actions on the stack immediately implies that the class of languages is closed under union and intersection. Surprisingly, we show that the nondeterministic and deterministic versions of these automata are equivalent. The determinization construction is the key technical theorem in this paper, and crucially uses the compositional reasoning of the automata using interfaces of tuples of global states that we alluded to earlier. Determinizability gives us closure under complement as well, and hence shows that the class of automata

with $k$-rounds of round-robin scheduling is a robust class closed under all Boolean operations.

The class of languages accepted by visibly multi-stack pushdown automata with a bounded number of context-switching rounds is the only class we are aware of that includes non-context-free languages, has a decidable emptiness (and membership) problem, is closed under all Boolean operations, and is further determinizable. (It is easy to see that this class is a subclass of context-sensitive languages, i.e. languages accepted by nondeterministic Turing machines with linear space). Note that standard *complexity* classes defined using resource constraints seldom have a decidable emptiness problem (one can prove undecidability by padding input).

We make several other observations regarding these automata. First, when the automata are generalized to have no bound on the number of round-robins of schedule, they are *not* determinizable. Second, it is well-known now that the emptiness problem restricted to only the words that can be accepted up to a bounded number of round-robins is decidable for these automata and in fact is NP-complete [13,16]. Thus, from the closure under boolean operations, it follows that universality and inclusion are decidable. Third, since these automata define a subclass of bounded-phase multi-stack pushdown automata [10], it follows that the Parikh theorem holds for the bounded context-switching class as well. Further, we show that the monadic second-order logic on $n$-nested words with $k$ rounds, where the logic has $n$ binary relations corresponding to the $n$ nesting relations on the word, corresponds exactly to the class of languages introduced in this paper.

Notice that our results show that the $k$-round-robin executions of multi-stack automata, which define a subclass of *context-sensitive languages* is determinizable. In [10], we show that even if there is *one phase* where a 2-stack automaton can push onto both stacks, followed by two phases where the automaton can pop from one stack only in each phase, is non-determinizable. This one example of non-determinizability rules out most natural extensions of multi-stack automata (where restrictions are based only on the patterns of pushes and pops) from being determinizable, and leads us to conjecture that considerably larger and determinizable sub-classes of context-sensitive languages defined using multi-stack nondeterministic automata are unlikely to exist.

In conclusion, the contribution of this paper is to exhibit a class of languages (those accepted by multi-stack automata with bounded rounds of context-switches), the first that we are aware of, that includes certain non-context-free languages, and has all the desirable properties that regular languages possess: closure under Boolean operations, decidable problems of membership, emptiness, and inclusion, determinizability and an MSO characterization.

**Related Work:** The classes of multi-stack automata studied in this paper are proper subclasses of the multi-stack automata which work in a bounded number of phases, where in each phase, symbols can be pushed into all stacks but popped only from one stack [10]. Though the class of languages accepted by bounded phase multi-stack automata is closed under all Boolean operations as well, such

automata are *not determinizable*, while the class of automata we consider here are. To the best of our knowledge, the class of automata we have introduced in this paper is the first extension of visibly pushdown automata with multiple stacks which is determinizable. [4] gives a wrong determinization construction for 2-stack visibly pushdown automata, which are indeed not determinizable (even if the stacks usage is constrained such that pop operations on the second stack are allowed only if the first stack is empty) as shown in [6]. Also, the class of 2-stack visibly pushdown automata is in general not closed under complement [2], while the answer is not known when such automata are constrained with an ordering on the usage of stacks (the proof of such closure property given in [4] relied on the determinizability of the model). Our determinization construction uses tuples of global states to capture the points at which context-switching occurs, similar to earlier papers [12,9] that reduce reachability in concurrent programs to reachability in sequential programs. However, the determinizability result does not follow from such conversions as the latter only preserve reachability, and not language equivalence (after all, sequential programs with finite data domains define only context-free languages). Besides the papers we have already cited, bounded context-switching has also been exploited for systems with heaps [3], systems communicating using queues [7], and weighted pushdown systems [13].

## 2 Multi-Stack Pushdown Automata

In this section we give the notation and definitions to introduce the model of automata we will use in the rest of the paper.

Given two positive integers $i$ and $j$, $i \leq j$, we denote with $[i, j]$ the set of integers $k$ with $i \leq k \leq j$, and with $[j]$ the set $[1, j]$.

An $n$-stack call-return alphabet is a tuple $\widetilde{\Sigma}_n = \langle \Sigma_c^i, \Sigma_r^i, \Sigma_{int}^i \rangle_{i=1}^n$ of pairwise disjoint finite alphabets. For any $i \in [n]$, $\Sigma_c^i$ is a finite set of *calls of stack i*, $\Sigma_r^i$ is a finite set of *returns of stack i*, and $\Sigma_{int}^i$ is a finite set of *internal actions of stack i*. For any such $\widetilde{\Sigma}_n$, let

- $\Sigma^i = \Sigma_c^i \cup \Sigma_r^i \cup \Sigma_{int}^i$, for every $i \in [n]$;
- $\Sigma_c = \bigcup_{i=1}^n \Sigma_c^i$, $\Sigma_r = \bigcup_{i=1}^n \Sigma_r^i$, and $\Sigma_{int} = \bigcup_{i=1}^n \Sigma_{int}^i$;
- $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$.

A multi-stack visibly pushdown automaton over such an alphabet must push on stack $i$ exactly one symbol when it reads a call of the $i$-th alphabet, and pop exactly one symbol from stack $i$ when it reads a return of the $i$-th alphabet. Also, it cannot touch any stack when reading an internal symbol.

**Definition 1.** (Multi-stack visibly pushdown automaton) *A* multi-stack visibly pushdown automaton *(*Mvpa*) over the n-stack call-return alphabet* $\widetilde{\Sigma}_n = \langle \Sigma_c^i, \Sigma_r^i, \Sigma_{int}^i \rangle_{i=1}^n$*, is a tuple* $M = (Q, Q_I, \Gamma, \delta, Q_F)$ *where* $Q$ *is a finite set of states,* $Q_I \subseteq Q$ *is the set of initial states,* $\Gamma$ *is a finite stack alphabet that contains a special bottom-of-stack symbol* $\perp$*,* $\delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\perp\})) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_{int} \times Q)$*, and* $Q_F \subseteq Q$ *is the set of final states.*

*Moreover, $M$ is* deterministic *if $|Q_I| = 1$, and $|\{(q,a,q') \in \delta\} \cup \{(q,a,q',\gamma) \in \delta\} \cup \{(q,a,\gamma',q') \in \delta\}| \leq 1$, for every given $q \in Q$, $a \in \Sigma$ and $\gamma' \in \Gamma$.*

Let us fix an $n$-stack alphabet $\widetilde{\Sigma}_n$ for the rest of the paper.

A transition $(q,a,q',\gamma)$, for $a \in \Sigma_c^i$ and $\gamma \neq \perp$, is a push-transition where on input $a$, $\gamma$ is pushed onto stack $i$ and the control changes from $q$ to $q'$. Similarly, $(q,a,\gamma,q')$ for $a \in \Sigma_r^i$ is a pop-transition where on input $a$, $\gamma$ is read from the top of stack $i$ and popped (except for $\gamma = \perp$, which is never popped), and the control changes from $q$ to $q'$. A transition $(q,a,q')$, for $a \in \Sigma_{int}$, is an internal transition where on input $a$ the control changes from $q$ to $q'$.

A *stack content* $\sigma$ is a nonempty finite sequence over $\Gamma$ where the bottom-of-stack symbol $\perp$ appears always in the end, i.e., $\sigma \in (\Gamma \setminus \{\perp\})^*.\{\perp\}$. A *configuration* of an MVPA $M$ over $\widetilde{\Sigma}_n$ is a tuple $C = \langle q, \sigma_1, \ldots, \sigma_n \rangle$, where $q \in Q$ and each $\sigma_i$ is a stack content. Moreover, $C$ is *initial* if $q \in Q_I$ and $\sigma_i = \perp$ for every $i \in [n]$, and *accepting* if $q \in Q_F$. *Transitions* between configurations are defined as follows: $\langle q, \sigma_1, \ldots, \sigma_n \rangle \xrightarrow{a}_M \langle q', \sigma'_1, \ldots, \sigma'_n \rangle$ if one of the following holds ($M$ is omitted whenever it is clear from the context):

**[Push]** If $a \in \Sigma_c^i$ (i.e., $a$ is a call of stack $i$), then $\exists \gamma \in \Gamma \setminus \{\perp\}$ such that $(q,a,q',\gamma) \in \delta$, $\sigma'_i = \gamma \cdot \sigma_i$, and $\sigma'_h = \sigma_h$ for every $h \in ([n] \setminus \{i\})$.

**[Pop]** If $a \in \Sigma_r^i$ (i.e., $a$ is a return of stack $i$), then $\exists \gamma \in \Gamma$ such that $(q,a,\gamma,q') \in \delta$, $\sigma'_h = \sigma_h$ for every $h \in ([n] \setminus \{i\})$, and either $\gamma \neq \perp$ and $\sigma_i = \gamma \cdot \sigma'_i$, or $\gamma = \sigma_i = \sigma'_i = \perp$.

**[Internal]** If $a \in \Sigma_{int}$, then $(q,a,q') \in \delta$, and $\sigma'_h = \sigma_h$ for every $h \in [n]$.

For a word $w = a_1 \ldots a_m$ in $\Sigma^*$, a *run* of $M$ on $w$ from $C_0$ to $C_m$, denoted $C_0 \xrightarrow{w}_M C_m$, is a sequence of transitions $C_{i-1} \xrightarrow{a_i} C_i$ for $i \in [m]$ where each $C_i$ is a configuration. A word $w \in \Sigma^*$ is accepted by an MVPA $M$ if there is an initial configuration $C$ and an accepting configuration $C'$ such that $C \xrightarrow{w}_M C'$. The language accepted by $M$ is denoted with $L(M)$.

A visibly pushdown automaton [1] is an MVPA with just one stack.

**Definition 2.** (VISIBLY PUSHDOWN AUTOMATON) *A* visibly pushdown automaton*, denoted* VPA*, is an* MVPA *over $\widetilde{\Sigma}_n$ with $n = 1$. A language over $\Sigma$ accepted by a* VPA *is a* visibly pushdown language*. With* VPL *we denote the class of visibly pushdown languages.*

### 2.1 Restricting to a Bounded Number of Rounds

A *context* over $\Sigma^i$, with $i \in [n]$, is a word in $(\Sigma^i)^*$. A *round* over $\widetilde{\Sigma}_n$ is a word $w$ of $\Sigma^*$ of the form $w_1 w_2 \ldots w_n$ where for each $i \in [n]$, $w_i$ is a context over $\Sigma^i$. A *$k$-round word* over $\widetilde{\Sigma}_n$ is a word of $\Sigma^*$ that can be obtained as the concatenation of $k$ rounds over $\widetilde{\Sigma}_n$. Let $Round(\widetilde{\Sigma}_n, k)$ denote the set of all the $k$-round words over $\widetilde{\Sigma}_n$.

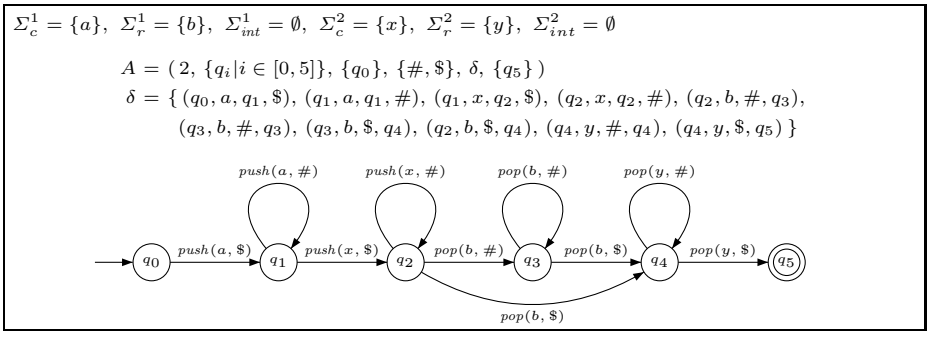


**Fig. 1.** A 2–round MVPA recognizing the language $\{\, a^t\, x^s\, b^t\, y^s \mid t, s \geq 1 \,\}$

**Definition 3.** (MULTI-STACK VISIBLY PUSHDOWN LANGUAGES WITH $k$-ROUNDS) *For any $k$, a* $k$-round multi-stack visibly pushdown automaton *($k$-round* MVPA*) over $\widetilde{\Sigma}_n$ is a tuple $A = (k, Q, Q_I, \Gamma, \delta, Q_F)$ where $M = (Q, Q_I, \Gamma, \delta, Q_F)$ is an* MVPA *over $\widetilde{\Sigma}_n$. Moreover, $A$ is* deterministic *iff $M$ is deterministic. The language accepted by $A$ is $L(A) = L(M) \cap Round(\widetilde{\Sigma}_n, k)$ and is called a* $k$-round multi-stack visibly pushdown language*. The class of $k$-round multi-stack visibly pushdown languages is denoted with* $k$-RVPL*. The set $\bigcup_{k\geq 1} k$-RVPL is denoted with* RVPL *(the class of* multi-stack visibly pushdown languages with a bounded number of rounds*).*

*Example 1.* Figure 1 gives a formal definition of a 2–round MVPA $A$ over $\widetilde{\Sigma}_2$ that accepts the language $\{a^t x^s b^t y^s \,|t, s \geq 1\}$. (Note that this language is not context-free and $A$ is deterministic.) The automaton $A$ checks whether the input word has the form $a^+x^+b^+y^+$ using its control states. $A$ starts in the control state $q_0$. When it reads the first call symbol $a$ it pushes the symbol \$ onto the stack $S_1$; for all the remaining $a$'s $A$ pushes the symbol # onto $S_1$. Stack $S_1$ will contain as many symbols as the number of read $a$'s. When the first call symbol $x$ of stack 2 is read a \$ symbol is pushed onto stack $S_2$, for the remaining $x$'s the symbol # is pushed onto stack $S_2$. As in the previous case, stack $S_2$ will contain as many symbols as the $x$'s which are read. Stack $S_1$ is then popped for each return symbol $b$ until $S_1$ is empty (read the symbol \$). Then only return symbols $y$ can be read. Stack $S_2$ is popped for each read $y$ until it gets empty (popping the symbol \$). After that $A$ moves into the accepting state $q_5$. □

The main result on MVPAs with a bounded number of rounds, that we show in this paper, is that the class of languages accepted by the deterministic and the nondeterministic models coincide. Notice that the boundedness of the number of rounds is crucial in our proof. In fact, determinizability does not hold in general for MVPAs. To see this consider the language $L = \{(ab)^i c^j d^{i-j} x^j y^{i-j} | i \in \mathbb{N}, j \in [i]\}$ over $\widetilde{\Sigma}_2 = (\Sigma_c^1, \Sigma_r^1, \emptyset, \Sigma_c^2, \Sigma_r^2, \emptyset)$, with $\Sigma_c^1 = \{a\}$, $\Sigma_r^1 = \{c, d\}$, $\Sigma_c^2 = \{b\}$, $\Sigma_r^2 = \{x, y\}$. First, observe that since the number of occurrences of $ab$ is unbounded in $L$ and $a$ and $b$ are from different stacks, this language contains words with an unbounded number of rounds and thus cannot be accepted by a

$k$-round MVPA for any fixed $k$. Further, this language is accepted by a nondeterministic MVPA which guesses nondeterministically the index $j$ when pushing symbols on reading $a$ and $b$. However, it is not accepted by any deterministic MVPA, since a deterministic MVPA would need an unbounded number of control states to store the index $j$ (see [10]). Nondeterminizability of MVPA also follows from the non-complementability of MVPA [2]. Therefore, we have:

**Theorem 1.** *The class of* MVPA *is not closed under determinization.*

## 3 Determinization of $k$-Round MVPAs

In this section we prove the main result of this paper: if $A$ is a $k$-round MVPA over $\widetilde{\Sigma}_n$, then there exists a deterministic $k$-round MVPA $A^D$ over $\widetilde{\Sigma}_n$ such that $L(A) = L(A^D)$.

Fix a $k$-round MVPA $A = (k, Q, Q_I, \Gamma, \delta, Q_F)$ over $\widetilde{\Sigma}_n = \langle \Sigma_c^j, \Sigma_r^j, \Sigma_{int}^j \rangle_{j=1}^n$ and a word $w \in \mathit{Round}(\widetilde{\Sigma}_n, k)$.

For ease of presentation, in the rest of this section we assume that each context of $w$ is not empty. Also, we denote with $w[i, j]$ the $j$-th context of the $i$-th round in $w$, and with $A_j$, $j \in [n]$, the VPA $(Q, Q_I, \Gamma, \delta', Q_F)$ over $\langle \Sigma_c^j, \Sigma_r^j, \Sigma_{int}^j \rangle$ where $\delta' \subseteq \delta$ is the set of all moves of $\delta$ on symbols of $\Sigma^j$ (i.e., the VPA which equals $A$ on the $j$-th stack).

The main idea behind our construction of $A^D$ is to look at the executions of $A$ on $w$ as shown in Fig. 2. The automaton $A$ is seen as a composition of the $A_j$'s. Initially $A$ is in control state $q_{\langle 1,1 \rangle}$. Then, it starts the computation by passing $q_{\langle 1,1 \rangle}$ to $A_1$. $A_1$ reads $w[1, 1]$ and reaches a control state $q'_{\langle 1,1 \rangle}$ with stack content $\sigma_{\langle 1,1 \rangle}$. At this point, $A$ stops $A_1$ and passes $q_{\langle 1,2 \rangle} = q'_{\langle 1,1 \rangle}$ to $A_2$. $A_2$ reads $w[1, 2]$ and reaches a state $q'_{\langle 1,2 \rangle}$ with stack content $\sigma_{\langle 1,2 \rangle}$. And so on, from $A_3$ through $A_n$, until $q'_{\langle 1,n \rangle}$ is reached. Now, $A$ passes $q_{\langle 2,1 \rangle} = q'_{\langle 1,n \rangle}$ to $A_1$. Since this is the first time $A_1$ is re-activated after reading $w[1, 1]$, its stack (i.e., the first one) has not changed in the meantime. Thus $A_1$ starts now from control state $q_{\langle 2,1 \rangle}$ and stack content $\sigma_{\langle 1,1 \rangle}$. Then, again by reading $w[2, 1]$, $A_1$ reaches a control state $q'_{\langle 2,1 \rangle}$ and $A_2$ is started from control state $q_{\langle 2,2 \rangle} = q'_{\langle 2,1 \rangle}$ and stack content $\sigma_{\langle 1,2 \rangle}$, and so on, until completion of the whole run.

The salient aspect in the above description is that each run of $A$ on $w$ can be computed by running each $A_j$ individually on $w[1, j], \ldots, w[k, j]$, provided that $q_{\langle 1,j \rangle}, \ldots, q_{\langle k,j \rangle}$ are fed. Also, note that $A_j$ computes a relation of state pairs $\langle q_{\langle i,j \rangle}, q'_{\langle i,j \rangle} \rangle$ which are connected by a run of $A_j$ over words $w[i, j]$ for $i \in [k]$, and thus, a relation of tuples $\langle q_{\langle i,j \rangle}, q'_{\langle i,j \rangle} \rangle_{i=1}^k$ corresponding to words $\langle w[i, j] \rangle_{i=1}^k$. We call such tuples *switching vectors*. Note that the switching vectors store all the information we need to stitch together the local runs of all $A_j$'s in order to build a global run of $A$.

This suggests the following scheme to construct $A^D$. First, for each $A_j$, construct a VPA $A'_j$ that computes the switching vectors corresponding to $\langle w[i, j] \rangle_{i=1}^k$ when reading $w[1, j] \# \ldots \# w[k, j] \#$, where $\#$ is a fresh internal symbol (computed switching vectors are stored in the final states). Construct for each $A'_j$ an

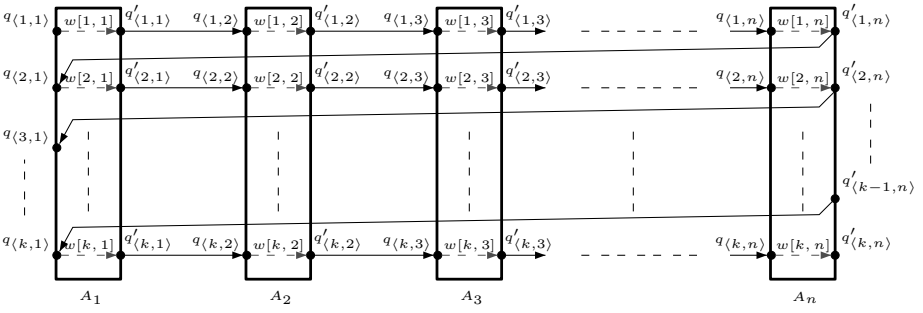


**Fig. 2.** Decomposition of a $k$-round MVPA

equivalent deterministic VPA $A_j^D$, and then, $A^D$ by composing them such that: (1) the states of $A^D$ are the cross product of the states of the $A_j^D$'s; (2) in each context over $\Sigma^j$, except for the first symbol, only $A_j^D$ is executed, and on reading the first symbol $a$ of a context $j+1$ of a round $i$, $A_j^D$ is executed on input # and $A_{j+1}^D$ is executed on input $a$; (3) a word $w$ is accepted if the computed switching vectors for each $A_j^D$ can be composed according to a scheme such as in Fig. 2, i.e., they form a *sequence of compatible tuples*.

The above sketched construction is formally addressed in the rest of this section (more details are available in the Appendix). We start by defining the switching vectors, and then construct the VPA computing the switching vectors for a given VPA. We then define the concept of compatible tuples and prove that acceptance of $A$ can be checked by verifying the existence of a sequence of compatible switching vectors of $A_1, \ldots, A_n$. Finally, we construct $A^D$ by composing the VPAs computing the switching vectors of $A_1, \ldots, A_n$ and argue its soundness and completeness.

### 3.1 Visibly Pushdown Automata Computing Switching-Vectors

**Definition 4.** (SWITCHING VECTORS) *Let $M$ be a* VPA *over $\widetilde{\Sigma}_1$ with set of control states $Q$, and $u = \langle u_i \rangle_{i=1}^k$ be a tuple of $k$ words in $\Sigma^*$. The tuple $V = \langle (q_i, q'_i) \rangle_{i=1}^k \in (Q \times Q)^k$, is a* switching-vector *of $M$ with respect to $u$ if there exist $k$ pairs of $M$ configurations $\langle (C_i, C'_i) \rangle_{i=1}^k$, with $C_i = \langle q_i, \sigma_i \rangle$ and $C'_i = \langle q'_i, \sigma'_i \rangle$, such that (1) $\sigma_1 = \perp$, (2) $\sigma'_i = \sigma_{i+1}$, for every $i \in [k-1]$ (3) $C_i \xrightarrow{u_i}_M C'_i$, for every $i \in [k]$.*

In the next lemma we prove the existence of a VPA $T$, called *switching automaton*, that computes switching-vectors of a given VPA.

**Lemma 1.** (SWITCHING AUTOMATA) *Let $M$ be a* VPA *over $\langle \Sigma_c, \Sigma_r, \Sigma_{int} \rangle$, and let # be a fresh symbol not in $\Sigma$. Then, there exists a nondeterministic* VPA *$T$ over $\langle \Sigma_c, \Sigma_r, \Sigma_{int} \cup \{\#\} \rangle$ such that $V$ is a switching-vector of $M$ with respect to $\langle u_i \rangle_{i=1}^k \in \Sigma^k$ iff while reading the word $u_1 \# u_2 \# \ldots \# u_k \#$, $T$ enters a state which contains $V$ (denoted as $\langle V \rangle$).*

*Sketch of the Proof.* The idea behind the construction of the VPA $T$ is the following. $T$ nondeterministically guesses, in its initial state, a switching-vector $V = \langle (q_i, q'_i) \rangle_{i=1}^{k} \in (Q \times Q)^k$ and then simulates $M$ on all the non-# symbols. In doing this, besides the current control state of $M$, $T$ also keeps track of the index $i$ of the current word $u_i$ which it is reading. Whenever $T$ reads the symbol #, it changes the control state of $M$ according to the guessed $V$: if $T$ reads the $i$-th #, and $q'_i$ is the state of $M$ before reading #, then $T$ changes the state of $M$ from $q'_i$ to $q_{i+1}$, with an internal move on #, thus matching the guessed switching-vector. In the end, when $T$ reads the last # (the $k$-th one) and the control state of $M$ is $q'_k$, then $T$ moves into a final state $\langle V \rangle$. □

### 3.2 Compatible Tuples

**Definition 5.** (COMPATIBLE TUPLES) *Let* $V_j = \langle (q_{\langle i,j \rangle}, q'_{\langle i,j \rangle}) \rangle_{i=1}^{k}$ *for* $j \in [n]$, *a sequence of* compatible *tuples* $V_1, V_2, \ldots, V_n$ *is such that*

- $q'_{\langle i,j \rangle} = q_{\langle i,j+1 \rangle}$, *for every* $i \in [k], j \in [n-1]$, *and*
- $q'_{\langle i,n \rangle} = q_{\langle i+1,1 \rangle}$, *for every* $i \in [k-1]$.

The following lemma is used in the next section to argue soundness and completeness of the determinization construction. It relates the acceptance of a word by a $k$-round MVPA to the existence of a sequence of compatible switching-vectors.

**Lemma 2.** *Let* $w \in Round(\widetilde{\Sigma}_n, k)$, $A = (k, Q, Q_I, \Gamma, \delta, Q_F)$ *be a* $k$*-round* MVPA *over* $\widetilde{\Sigma}_n$, *and* $w_j = \langle w[i,j] \rangle_{i=1}^{k}$, *for* $j \in [n]$. *The word* $w \in L(A)$ *iff for each* $j \in [n]$, *there exists a switching-vector* $V_j = \langle (q_{\langle i,j \rangle}, q'_{\langle i,j \rangle}) \rangle_{i=1}^{k}$ *of the* VPA $A_j$ *with respect to* $w_j$ *such that* $V_1, V_2, \ldots, V_n$ *is a sequence of compatible tuples,* $q_{\langle 1,1 \rangle} \in Q_I$, *and* $q_{\langle k,n \rangle} \in Q_F$.

### 3.3 Determinization of $k$-Round MVPAs

**Theorem 2.** *(*DETERMINIZABILITY*) If* $A$ *is a* $k$*-round* MVPA *over* $\widetilde{\Sigma}_n$, *then there exists a deterministic* $k$*-round* MVPA $A^D$ *over* $\widetilde{\Sigma}_n$ *such that* $L(A) = L(A^D)$. *Moreover, the size of* $A^D$ *is doubly exponential in the number of rounds, and singly exponential in the number of stacks and the number of states of* $A$.

*Proof.* For $j \in [n]$, let $T_j$ be a switching automaton which accepts the same language as $A_j$ and is constructed according to Lemma 1, and $S_j$ be the deterministic VPA such that $L(S_j) = L(T_j)$ which is obtained via the construction given in [1]. Thus, a state of $S_j$ contains the set of reachable control states of $T_j$. From the definition of $T_j$, it is easy to see that the $S_j$ state reached on input $u_1 \# u_2 \# \ldots \# u_k \#$ is the set of all the switching-vectors of $A_j$ with respect to $\langle u_i \rangle_{i=1}^{k} \in \Sigma^k$.

The idea behind the construction of $A^D$ is the following. $A^D$ simulates each $S_j$, by keeping track of the control state of every $S_j$. The entire simulation mimics the schema shown in Fig. 2. After the input word $w$ is completely read, $A^D$

reaches a state storing the set of all switching-vectors of each $A_j$. The states of $A^D$ which contain a sequence of compatible tuples are defined final. Thus, from Lemma 2, $A^D$ accepts the input word $w$ if and only if $A$ also does.

An issue that has to be addressed in the simulation of all $S_j$ is the following. Let $w$ be the input word of $A$. $S_j$ needs to read a symbol # when a context-switch happens, i.e., at the end of each context $j$ of each round. Thus, the idea is to simulate the move on # meanwhile $A^D$ processes the first symbol of the next context. This solves the problem for all the occurrences of # but the last one (no other context follows). Thus, for the simulation of $S_n$ in the last round, we keep a pair $(q, q_{\#})$ where $q$ is the current state of $S_n$ in the simulation, and $q_{\#}$ is the state computed from $q$ by applying the transition on #. Thus, $q$ is used to simulate the moves of $S_n$, and $q_{\#}$ is considered only for acceptance. □

## 4 Discussion

The class of languages studied in this paper is closely related to the class of *bounded-phase* MVPAs studied in [10]; using this relationship, we can derive many properties for RVPLs. Intuitively, a phase is a stage of computation of a multi-stack automaton where push actions are allowed on all stacks while pop actions are allowed only on one (hence phases generalize contexts).

MVPA ***with a bounded number of phases.*** Given a word $w \in \Sigma^*$, we denote with $Ret(w)$ the set of all returns in $w$. A word $w$ is a *phase* if $Ret(w) \subseteq \Sigma_r^i$, for some $i \in [n]$. For any $k$, a $k$-phase word is a word $w \in \Sigma^*$ such that $w$ can be factorized as $w = w_1 w_2 \dots w_{k'}$ where $k' \leq k$ and $w_h$ is a phase, for every $h \in [k']$. With $Phases(\widetilde{\Sigma}_n, k)$ we denote the set of all $k$-phase words over $\widetilde{\Sigma}_n$.

For any $k$, a *$k$-phase multi-stack visibly pushdown automaton* ($k$-phase MVPA) over $\widetilde{\Sigma}_n$ is a tuple $A = (k, Q, Q_I, \Gamma, \delta, Q_F)$ where $M = (Q, Q_I, \Gamma, \delta, Q_F)$ is an MVPA over $\widetilde{\Sigma}_n$. The language accepted by $A$ is $L(A) = L(M) \cap Phases(\widetilde{\Sigma}_n, k)$. The class of languages accepted by $k$-phase MVPAs is denoted with $k$-PVPL, and the set $\bigcup_{k>0} k$-PVPL is denoted with PVPL (the class of all languages accepted by a $k$-phase MVPA for some $k$).

**Theorem 3 ([10]).** *Let $k$ be any positive integer. $k$-PVPLs are closed under union, intersection, and complement. The membership, emptiness, inclusion, equivalence, and universality problems are decidable for $k$-PVPLs. $k$-PVPLs are not determinizable.*

The notion of phase is less restrictive than the notion of context, i.e., a context is a phase, and hence a round of context-switching can be simulated using a bounded number of phases. Hence:

**Lemma 3.** *Let the number of stacks be $n$. Then $k$-RVPL $\subset (k \cdot n)$-PVPL and* RVPL $\subset$ PVPL.

***Closure properties and decision problems.*** Closure under union and intersection of $k$-RVPL can be shown with standard constructions, and decidability

| | Closure properties | | | | Decision Problems | |
|---|---|---|---|---|---|---|
| | $\cup$ | $\cap$ | Compl. | Determ. | Emptiness | Univ./ Equiv./Incl. |
| Reg. | Yes | Yes | Yes | Yes | NLOG-C | PSPACE-C |
| VPL | Yes | Yes | Yes | Yes | PTIME-C | EXPTIME-C |
| CFL | Yes | No | No | No | PTIME-C | Undecidable |
| **Rvpl** | **Yes** | **Yes** | **Yes** | **Yes** | **NP-c** | **2Exptime** |
| PVPL | Yes | Yes | Yes | No | 2ETIME-C | 3EXPTIME<br>2EXPTIME-HARD |
| CSL | Yes | Yes | Yes | Unknown | Undecidable | Undecidable |

**Fig. 3.** Summary of main closure properties and decision problems

of decision problems such as membership and emptiness can be inherited from $k$-PVPL. Notice that since complementation of $k$-RVPL is defined with respect to words with bounded rounds of context-switching, closure under complement does not immediately follow from closure under complement for $k$-PVPL. However, closure under complement for $k$-RVPL follows from determinizability of the corresponding class of automata. Therefore, we get the following results:

**Theorem 4.** *Let $k$ be any positive integer. $k$-RVPLs are closed under union, intersection, and complement. The membership, emptiness, inclusion, equivalence, and universality problems are decidable for $k$-RVPLs.*

The table in Figure 3 summarizes the closure properties and decision problems for CSLs, CFLs, VPLs, PVPLs, RVPLs, and regular languages (see [1] for VPLs, and [5] for CSLs,CFLs and regular languages). In the table, NLOG-C stands for NLOG-complete, and so on.

***Parikh Theorem.*** The Parikh mapping $\Phi(w)$, introduced by Parikh [15], associates a word with the vector of natural numbers that reflect the number of occurrences of the symbols in the word. This mapping extends to languages in the natural way. Since a Parikh theorem holds for $k$-phase MVPAs [10], from Lemma 3 we get:

**Corollary 1.** *For every* RVPL $L$ *over $\widetilde{\Sigma}_n$, there exists a regular language $L'$ over $\Sigma$ such that $\Phi(L') = \Phi(L)$. Moreover, $L'$ can be effectively computed.*

***A Logical Characterization.*** Consider the *monadic second-order logic* ($\mathrm{MSO}_\mu$) over $\widetilde{\Sigma}_n$ defined by:

$$\varphi := P_a(x) | x \in X | x \leq y | \mu_j(x, y) | \neg\varphi | \varphi \vee \varphi | \exists x \varphi | \exists X \varphi$$

where $j \in [n]$, $a \in \Sigma$, $x, y$ are a first-order variables and $X$ is a set variable [10].

The models are words over $\Sigma$. Each of the $n$ binary relations $\mu_j$ ($j \in [n]$) is interpreted as the nested matching relation of calls and returns of $\Sigma^j$ . We denote with $R_k(\varphi)$ the set of all words of $Round(\widetilde{\Sigma}_n, k)$ that satisfy a sentence $\varphi$. By standard techniques that convert MSO to automata (given that the automata are closed under boolean operations and projection), we get (see [18]):

**Theorem 5.** *A language $L$ is a $k$-RVPL over $\widetilde{\Sigma}_n$ iff there is an $MSO_\mu$ sentence $\varphi$ over $\widetilde{\Sigma}_n$ with $R_k(\varphi) = L$.*

## References


1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC, pp. 202–211. ACM, New York (2004)
2. Bollig, B.: On the expressive power of 2-stack visibly pushdown automata. LMCS 4(4:16), 1–35 (2008)
3. Bouajjani, A., Fratani, S., Qadeer, S.: Context-bounded analysis of multithreaded programs with dynamic linked structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
4. Carotenuto, D., Murano, A., Peron, A.: 2-visibly pushdown automata. In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 132–144. Springer, Heidelberg (2007)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
6. La Torre, S., Madhusudan, P., Parlato, G.: 2-VPAs are not determinizable (2007), `http://www.cs.uiuc.edu/homes/~madhu/vpa/wrong-proof-CMP07.html`
7. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
8. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using fixed-point calculus. In: PLDI, pp. 211–222. ACM, New York (2009)
9. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
10. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS, pp. 161–170. IEEE Computer Society, Los Alamitos (2007)
11. Lahiri, S., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
12. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
13. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008) (See also Tech Report TR-1598, Comp. Sc. Dept., Univ. of Wisconsin, Madison)
14. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM, New York (2007)
15. Parikh, R.: On context-free languages. J. ACM 13(4), 570–581 (1966)
16. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
17. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic context-bounded analysis of multithreaded Java programs. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)
18. Thomas, W.: Languages, automata, and logic. In: Handbook of Formal Languages, vol. 3, pp. 389–455 (1997)
19. VPL, `http://www.cs.uiuc.edu/homes/~madhu/vpa/`