ENS CACHAN

Realizability of Concurrent Recursive Programs

Benedikt Bollig, Manuela-Lidia Grindei, and Peter Habermehl

Research Report LSV-08-29

October 2008

Laboratoire Spécification et Vérification

CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Realizability of Concurrent Recursive Programs*

Benedikt Bollig[1], Manuela-Lidia Grindei[1], and Peter Habermehl[1,2]

[1] LSV, ENS Cachan, CNRS, INRIA, France
email: {bollig,grindei}@lsv.ens-cachan.fr
[2] LIAFA, CNRS and University Paris Diderot, France
email: haberm@liafa.jussieu.fr

Abstract. We define and study an automata model of concurrent recursive programs. An automaton consists of a finite number of pushdown systems running in parallel and communicating via shared actions. Actually, we combine multi-stack visibly pushdown automata and Zielonka's asynchronous automata towards a model with an undecidable emptiness problem. However, a reasonable restriction allows us to lift Zielonka's Theorem to this recursive setting and permits a logical characterization in terms of a suitable monadic second-order logic. Building on results from Mazurkiewicz trace theory and recent work by La Torre, Madhusudan, and Parlato, we thus develop a framework for the specification, synthesis, and verification of concurrent recursive processes.

1 Introduction

The analysis of a concurrent recursive program where several recursive threads access a shared memory is a difficult task due to the typically high complexity of interaction between its components. One general approach is to run a verification algorithm on a finite-state abstract model of the program. As the model usually preserves recursion, this amounts to verifying multi-stack pushdown automata. Unfortunately, even if we deal with a boolean abstraction of data, the control-state reachability problem in this case is undecidable [24]. However, as proved in [23], it becomes decidable if only those states are taken into consideration that can be reached within a bounded number of context switches. A context switch consists in a transfer of control from one process to another. This result allows for the discovery of many errors, since they typically manifest themselves after a few context switches [8, 18, 23]. Other approaches to analyzing multi-threaded programs restrict the kind of communication between processes [18, 26], or compute over-approximations of the set of reachable states [5].

All these works have in common that they restrict to the analysis of an already existing system. A fundamentally different approach would be to synthesize a concurrent recursive program from a requirements specification, preferably automatically, so that that the inferred system can be considered "correct by construction". The general idea of synthesizing programs from specifications

* Partially supported by ARCUS, DOTS (ANR-06-SETIN-003), and P2R MODISTE-COVER/RNP Timed-DISCOVERI.

goes back to [12]. The particular case of non-recursive distributed systems is, e.g., dealt with in [7, 9, 19].

In this paper, we address the synthesis problem for finite-state concurrent *recursive* programs that communicate via shared variables. More precisely, we are interested in transforming a given global specification in terms of a context-sensitive language into a design model of a distributed implementation thereof. The first step is to provide an automata model that captures both asynchronous procedure calls and shared-variable communication. To this aim, we combine visibly pushdown automata [2] and asynchronous automata [29], which, seen individually, constitute robust automata classes with desirable closure properties and decidable verification problems.

Visibly pushdown automata were introduced in [2] by Alur and Madhusudan. One such automaton reads words over an alphabet partitioned in three sets: the call alphabet, the return alphabet and the internal alphabet. The automaton is supposed to push exactly one symbol onto its stack if it reads a call letter, to pop exactly one symbol from its stack if it reads a return letter, and to leave the stack unchanged if it reads an internal letter. The good news about this class is that it is closed under boolean operations and has decidable basic decision problems (in particular, one can decide if the intersection of two visibly pushdown languages is empty, which is an undecidable problem for ordinary pushdown automata).

Asynchronous automata can be traced back to Zielonka, who defined this model in the framework of the partial-order model of Mazurkiewicz traces [29]. Herein, local processes synchronize by executing certain actions (e.g., writing a variable) simultaneously, whereas others may be taken autonomously. Roughly speaking, asynchronous automata have the same nice language-theoretical properties as finite automata.

Merging visibly pushdown automata and asynchronous automata, we obtain *concurrent visibly pushdown automata* (CVPA). They serve as our model of an implementation and are actually a special case of multi-stack visibly pushdown automata (MVPA), which were introduced in [17]. For MVPA, the reachability problem is again undecidable. To counteract this drawback, a reasonable restriction has been considered in [17]: The domain of input words is restricted to *k-phase words*, which can be decomposed into k subwords where k is a natural number. One such subword is supposed to contain return symbols for the same stack, if any. This model leads to a decidable emptiness problem and to a class of languages closed under boolean operations. A particularly nice feature in our context is that one can decide whether an error state can be reached within k phases, having all processes able to evolve in each phase but only one process can return from a procedure, which is less restrictive than the notion of bounded context switches that we discussed above.

Let us turn to the main contributions of our paper. In the context of k-phase words, we will provide a decidable sufficient criteria when an MVPA, which represents a context-sensitive specification, can be transformed into a CVPA. In doing so, we lift Zielonka's well-known theorem to a recursive setting. Recall that it was argued in [23] that system errors typically manifest themselves within few

context switches. This suggests that the global state space of a concurrent recursive program is often the closure of a set of k-phase words under permutation rewriting of independent events. We will actually show that the closure of an MVPA language that is *represented* (in a sense that will be made clear) by its k-phase executions can be realized as a CVPA. The problem with MVPA as specifications is, however, that they do not necessarily possess a closure property that CVPA naturally have. We therefore propose, in another section of this paper, to use monadic-second order (MSO) logic as a specification language. In fact, we can show that, under the assumption of a k-phase restriction, any formula from this logic can be effectively transformed into a CVPA. It should be noted that this constitutes an extension of the classical connection between finite (asynchronous) automata and MSO logic [6, 27, 28].

Organization Section 2 provides basic definitions and introduces MVPA and CVPA. Section 3 considers the task of synthesizing a distributed system in terms of a CVPA from an MVPA specification. In doing so, we give two extensions of Zielonka's Theorem to concurrent recursive programs. In Section 4, we provide a logical characterization of CVPA in terms of MSO logic. We conclude with Section 5, in which we suggest several directions for future work.

2 Definitions

The set $\{0, 1, 2, \ldots\}$ of natural numbers is denoted by $\mathbb{N}$. We call any finite set an *alphabet*. Its elements are called *letters* or *actions*. For an alphabet Σ, Σ^* is the set of finite words over Σ; the empty word is denoted by ε. The concatenation uv of words $u, v \in \Sigma^*$ is denoted by $u \cdot v$. For a set X, we let $|X|$ denote its size and 2^X its powerset.

2.1 Concurrent Pushdown Alphabets

The architecture of a system is constituted by a *concurrent (visibly) pushdown alphabet*. To define it formally, we fix a nonempty finite set *Proc* of *process names* or, simply, *processes*. Now consider a collection $\widetilde{\Sigma} = ((\Sigma_p^c, \Sigma_p^r, \Sigma_p^{int}))_{p \in Proc}$ of alphabets. The triple $(\Sigma_p^c, \Sigma_p^r, \Sigma_p^{int})$ associated with process p contains the supplies of actions that can be executed by process p. More precisely, the alphabets contain its call, return, and internal actions, respectively. We call $\widetilde{\Sigma}$ a *concurrent pushdown alphabet* (over *Proc*) if

- for every $p \in Proc$, the sets Σ_p^c, Σ_p^r, and Σ_p^{int} are pairwise disjoint, and
- for every $p, q \in Proc$ with $p \neq q$, $(\Sigma_p^c \cup \Sigma_p^r) \cap (\Sigma_q^c \cup \Sigma_q^r) = \emptyset$.

For $p \in Proc$, let Σ_p refer to $\Sigma_p^c \cup \Sigma_p^r \cup \Sigma_p^{int}$, the set of actions that are available to p. Thus, $\Sigma = \bigcup_{p \in Proc} \Sigma_p$ is the set of all actions. Furthermore, for $a \in \Sigma$, let $proc(a) = \{p \in Proc \mid a \in \Sigma_p\}$. The intuition behind a concurrent pushdown alphabet is as follows: An action $a \in \Sigma$ is executed simultaneously by every process from $proc(a)$. In doing so, a process $p \in proc(a)$ can access the current

state of any other process from $proc(a)$. The only restriction is that p can access and modify only its own stack, provided $a \in \Sigma_p^c \cup \Sigma_p^r$. However, in that case, the stack operation can be "observed" by some other process q if $a \in \Sigma_q^{int}$.

We introduce further useful abbreviations and let $\Sigma^c = \bigcup_{p \in Proc} \Sigma_p^c$, $\Sigma^r = \bigcup_{p \in Proc} \Sigma_p^r$, and $\Sigma^{int} = (\bigcup_{p \in Proc} \Sigma_p^{int}) \setminus (\Sigma^c \cup \Sigma^r)$.

Example 1. Let $Proc = \{p, q\}$ and let $\widetilde{\Sigma} = ((\{a\}, \{\overline{a}\}, \{b\}), (\{b\}, \{\overline{b}\}, \emptyset))$ be a concurrent pushdown alphabet where the triple $(\{a\}, \{\overline{a}\}, \{b\})$ refers to process p and $(\{b\}, \{\overline{b}\}, \emptyset)$ belongs to process q. Thus, $\Sigma = \{a, \overline{a}, b, \overline{b}\}$, $\Sigma^c = \{a, b\}$, $\Sigma^r = \{b, \overline{b}\}$, and $\Sigma^{int} = \emptyset$. Note also that $proc(a) = \{p\}$ and $proc(b) = \{p, q\}$.

If not stated otherwise, $\widetilde{\Sigma}$ will henceforth be any concurrent pushdown alphabet.

2.2 Multi-Stack Visibly Pushdown Automata

Before we introduce our new automata model, we recall multi-stack visibly pushdown automata, as recently introduced by La Torre, Madhusudan, and Parlato [17]. Though this model will be parameterized by $\widetilde{\Sigma}$, it is not distributed yet. The concurrent pushdown alphabet only determines the number of stacks (which equals $|Proc|$) and the actions operating on them. In the next subsection, an element $p \in Proc$ will then actually play the role of a process.

Definition 2. *A* multi-stack visibly pushdown automaton (MVPA) *over $\widetilde{\Sigma}$ is a tuple $\mathcal{A} = (S, \Gamma, \delta, \iota, F)$ where*

- *S is its finite set of* states*,*
- *$\iota \in S$ is the* initial state*,*
- *$F \subseteq S$ is the set of* final states*,*
- *Γ is the finite* stack alphabet *containing a special symbol $\bot$, and*
- *$\delta \subseteq S \times \Sigma \times \Gamma \times S$ is the set of* transitions*.*

Consider a transition $(s, a, A, s') \in \delta$. If $a \in \Sigma_p^c$, then we deal with a push-transition meaning that, being in state s, the automaton can read a, push the symbol $A \in \Gamma \setminus \{\bot\}$ onto the p-stack, and go over to state s'. Transitions $(s, a, A, s') \in \delta$ with $a \in \Sigma^c$ and $A = \bot$ are discarded. If $a \in \Sigma_p^r$, then the transition allows us to pop $A \neq \bot$ from the p-stack when reading a, while the control changes from state s to state s'; if, however, $A = \bot$, then the a can be executed provided the stack of p is empty, i.e., $\bot$ is never popped. Finally, if $a \in \Sigma^{int}$, then an internal action is applied, which does not involve a stack operation. In that case, the symbol A is simply ignored.

Let us formalize the behavior of the MVPA $\mathcal{A}$. A *stack contents* is a nonempty finite sequence from $Cont = (\Gamma \setminus \{\bot\})^* \cdot \{\bot\}$. The leftmost symbol is thus the top symbol of the stack contents. A configuration of $\mathcal{A}$ consists of a state and a stack contents for each process. Hence, it is an element of $S \times Cont^{Proc}$. Consider a word $w = a_1 \dots a_n \in \Sigma^*$. A *run* of $\mathcal{A}$ on w is a sequence $\rho = (s_0, (\sigma_p^0)_{p \in Proc}) \dots (s_n, (\sigma_p^n)_{p \in Proc}) \in (S \times Cont^{Proc})^*$ such that $s_0 = \iota$, $\sigma_p^0 = \bot$ for all $p \in Proc$, and, for all $i \in \{1, \dots, n\}$, the following hold:

[Push] If $a_i \in \Sigma_p^c$ for $p \in \mathit{Proc}$, then there is a stack symbol $A \in \Gamma \setminus \{\bot\}$ such that $(s_{i-1}, a_i, A, s_i) \in \delta$, $\sigma_p^i = A \cdot \sigma_p^{i-1}$, and $\sigma_q^i = \sigma_q^{i-1}$ for all $q \in \mathit{Proc} \setminus \{p\}$.

[Pop] If $a_i \in \Sigma_p^r$ for $p \in \mathit{Proc}$, then there is a stack symbol $A \in \Gamma$ such that $(s_{i-1}, a_i, A, s_i) \in \delta$, $\sigma_q^i = \sigma_q^{i-1}$ for all $q \in \mathit{Proc} \setminus \{p\}$, and either $A \neq \bot$ and $\sigma_p^{i-1} = A \cdot \sigma_p^i$, or $A = \bot$ and $\sigma_p^i = \sigma_p^{i-1} = \bot$.

[Internal] If $a_i \in \Sigma^{int}$, then there is $A \in \Gamma$ such that $(s_{i-1}, a_i, A, s_i) \in \delta$ and $\sigma_p^i = \sigma_p^{i-1}$ for every $p \in \mathit{Proc}$.

The run ρ is *accepting* if $s_n \in F$. A word $w \in \Sigma^*$ is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The set of accepted words forms the *language* of $\mathcal{A}$, which is denoted by $L(\mathcal{A})$.

It is easy to see that the emptiness problem for MVPA is undecidable. Moreover, it has been shown that MVPA can in general not be complemented [4]. We can remedy this situation by restricting our domain to k-phase words [17]. Let $k \in \mathbb{N}$. A word $w \in \Sigma^*$ is called a *k-phase word* over $\widetilde{\Sigma}$ if it can be written as $w_1 \cdot \ldots \cdot w_k$ where, for all $i \in \{1, \ldots, k\}$, we have $w_i \in (\Sigma^c \cup \Sigma^{int} \cup \Sigma_p^r)^*$ for some $p \in \mathit{Proc}$. The set of k-phase words over $\widetilde{\Sigma}$ is denoted by $\mathrm{W}_k(\widetilde{\Sigma})$. Note that $\mathrm{W}_k(\widetilde{\Sigma})$ is regular. The language of $\mathcal{A}$ relative to k-phase words, denoted by $L_k(\mathcal{A})$, is defined to be the set $L(\mathcal{A}) \cap \mathrm{W}_k(\widetilde{\Sigma})$. Even if we restrict to k-phase words, a deterministic variant of MVPA is strictly weaker, unless we deal with simple concurrent pushdown alphabets where $\Sigma = \Sigma^{int}$ [17, 29].

In this paper, we will exploit the following two theorems concerning MVPA. The first states that emptiness of MVPA is decidable wrt. k-phase words. The second establishes that an MVPA recognizing a set of k-phase words can be complemented.

Theorem 3 (La Torre-Madhusudan-Parlato [17]). *The following problem is decidable:*

INPUT*: Concurrent pushdown alphabet $\widetilde{\Sigma}$; $k \in \mathbb{N}$;* MVPA *$\mathcal{A}$ over $\widetilde{\Sigma}$.*
QUESTION*: Does $L_k(\mathcal{A}) \neq \emptyset$ hold?*

The problem is decidable in doubly exponential time wrt. $|S|$, $|\mathit{Proc}|$, and k, where S is the set of states of $\mathcal{A}$.

Theorem 4 (La Torre-Madhusudan-Parlato [17]). *Let $k \in \mathbb{N}$ and let $\mathcal{A}$ be an* MVPA *over $\widetilde{\Sigma}$. One can effectively construct an* MVPA *$\mathcal{A}'$ over $\widetilde{\Sigma}$ such that $L(\mathcal{A}') = \overline{L_k(\mathcal{A})}$, where $\overline{L_k(\mathcal{A})}$ is defined to be $\Sigma^* \setminus L_k(\mathcal{A})$.*

2.3 Concurrent Visibly Pushdown Automata

We let $I_{\widetilde{\Sigma}} = \{(a, b) \in \Sigma \times \Sigma \mid \mathit{proc}(a) \cap \mathit{proc}(b) = \emptyset\}$ contain the pairs of actions that are considered *independent*. Moreover, $\sim_{\widetilde{\Sigma}} \subseteq \Sigma^* \times \Sigma^*$ shall be the least congruence that satisfies $ab \sim_{\widetilde{\Sigma}} ba$ for all $(a, b) \in I_{\widetilde{\Sigma}}$. The equivalence class of a representative $w \in \Sigma^*$ wrt. $\sim_{\widetilde{\Sigma}}$ is denoted by $[w]_{\sim_{\widetilde{\Sigma}}}$. We canonically extend $[.]_{\sim_{\widetilde{\Sigma}}}$ to sets $L \subseteq \Sigma^*$ and let $[L]_{\sim_{\widetilde{\Sigma}}} = \{w \in \Sigma^* \mid w \sim_{\widetilde{\Sigma}} w' \text{ for some } w' \in L\}$.

Based on Definition 2, we now introduce our model of a concurrent recursive program, which will indeed produce languages that are closed under $\sim_{\widetilde{\Sigma}}$.

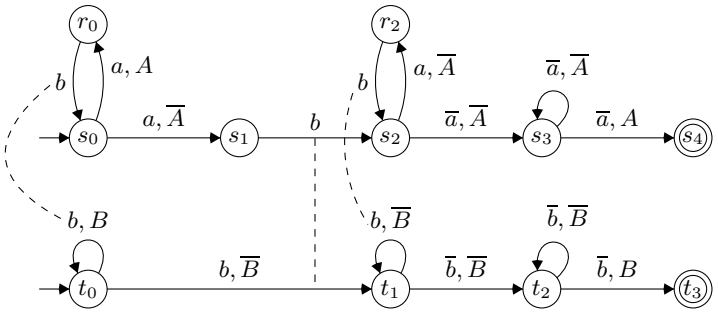

Fig. 1. A concurrent visibly pushdown automaton

Definition 5. *A* concurrent visibly pushdown automaton (CVPA) *over* $\widetilde{\Sigma}$ *is an* MVPA $(S, \Gamma, \delta, \iota, F)$ *over* $\widetilde{\Sigma}$ *such that there exist*

- *a family* $(S_p)_{p \in Proc}$ *of sets of* local states *and*
- *relations* $\delta_a \subseteq \left(\prod_{p \in proc(a)} S_p\right) \times \Gamma \times \left(\prod_{p \in proc(a)} S_p\right)$ *for* $a \in \Sigma$

satisfying the following properties:

- $S = \prod_{p \in Proc} S_p$ *and*
- *for every* $s, s' \in S$, $a \in \Sigma$, *and* $A \in \Gamma$, *we have* $(s, a, A, s') \in \delta$ *iff*
  - $((s_p)_{p \in proc(a)}, A, (s'_p)_{p \in proc(a)}) \in \delta_a$ *and*
  - $s_q = s'_q$ *for every* $q \in Proc \setminus proc(a)$

  where s_p *denotes the* p*-component of state* s.

To make local states and their transition relations explicit, we may consider a CVPA to be a structure $((S_p)_{p \in Proc}, \Gamma, (\delta_a)_{a \in \Sigma}, \iota, F)$.

Note that, if $\Sigma = \Sigma^{int}$ (i.e., $\widetilde{\Sigma} = ((\emptyset, \emptyset, \Sigma_p))_{p \in Proc}$), then a CVPA can be seen as a simple asynchronous automaton [9, 29]. It is straightforward to show that the language $L(\mathcal{C})$ of a CVPA $\mathcal{C}$ is closed under $\sim_{\widetilde{\Sigma}}$ meaning that $L(\mathcal{C}) = [L(\mathcal{C})]_{\sim_{\widetilde{\Sigma}}}$.

Lemma 6. *Let* $\mathcal{C}$ *be a* CVPA *over* $\widetilde{\Sigma}$. *For every* $u, v \in \Sigma^*$ *with* $u \sim_{\widetilde{\Sigma}} v$, *we have* $u \in L(\mathcal{C})$ *iff* $v \in L(\mathcal{C})$.

Example 7. Consider the concurrent pushdown alphabet $\widetilde{\Sigma}$ from Example 1. Assume $\mathcal{C} = (S, \Gamma, \delta, \iota, F)$ to be the CVPA depicted in Figure 1 where S is the cartesian product of $S_p = \{s_0, \ldots, s_4, r_0, r_2\}$ and $S_q = \{t_0, \ldots, t_3\}$. Actions from $\{a, \overline{a}, \overline{b}\}$ are exclusive to a single process so that corresponding transitions are local. For example, the relation δ_a, as required in Definition 5, is given by $\{(s_0, A, r_0), (s_0, \overline{A}, s_1), (s_2, \overline{A}, r_2)\}$. Thus, $((s_0, t_i), a, A, (r_0, t_i)) \in \delta$ for all $i \in \{0, \ldots, 3\}$. In contrast, executing b involves both processes, which is indicated by the dashed lines depicting δ_b. For example, $((s_1, t_0), \overline{B}, (s_2, t_1)) \in \delta_b$. Furthermore, $((r_0, t_0), b, B, (s_0, t_0))$, $((s_1, t_0), b, \overline{B}, (s_2, t_1))$, and $((r_2, t_1), b, \overline{B}, (s_2, t_1))$ are the global b-transitions contained in δ. Note that $L_1(\mathcal{C}) = \emptyset$, since at least two phases are needed to reach the final state (s_4, t_3). Moreover,

– $L_2(\mathcal{C}) = \{(ab)^n w \mid n \geq 2,\ w \in \{\overline{a}^m \overline{b}^m, \overline{b}^m \overline{a}^m\} \text{ for some } m \in \{2, \ldots, n\}\}$ and
– $L(\mathcal{C}) = \{(ab)^n w \mid n \geq 2,\ w \in \{\overline{a}, \overline{b}\}^*,\ |w|_{\overline{a}} = |w|_{\overline{b}} \in \{2, \ldots, n\}\} = [L_2(\mathcal{C})]_{\sim_{\widetilde{\Sigma}}}$.

As $L(\mathcal{C}) = [L_2(\mathcal{C})]_{\sim_{\widetilde{\Sigma}}}$, the language $L_2(\mathcal{C})$ can be viewed as an incomplete description of $L(\mathcal{C})$. The next section deals with how to derive CVPA from both incomplete specifications and those that are closed under $\sim_{\widetilde{\Sigma}}$.

3 Realizability of Concurrent Recursive Programs

From now on, we consider an MVPA $\mathcal{A}$ to be a specification of a system, and we are looking for a *realization* or *implementation* of $\mathcal{A}$, which is provided by a CVPA $\mathcal{C}$ such that $L(\mathcal{C}) = L(\mathcal{A})$. Actually, specifications often have a "global" view of the system, and the difficult task is to *distribute* the state space onto the processes, which henceforth communicate in a restricted manner that conforms to the predefined system architecture $\widetilde{\Sigma}$. If, on the other hand, $\mathcal{A}$ is not closed under $\sim_{\widetilde{\Sigma}}$, it might yet be considered as an incomplete specification so that we ask for a CVPA $\mathcal{C}$ such that $L(\mathcal{C}) = [L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$.

In this section, we make use of two well-known theorems from Mazurkiewicz trace theory, which we recall in the following. The first one, Zielonka's celebrated theorem, applies to simple concurrent pushdown alphabets, and it will later be lifted to general concurrent pushdown alphabets.

Theorem 8 (Zielonka [29]). *Suppose $\Sigma = \Sigma^{int}$. For every regular language $L \subseteq \Sigma^*$ that is $\sim_{\widetilde{\Sigma}}$-closed, there is a* CVPA $\mathcal{C}$ *over $\widetilde{\Sigma}$ such that $L(\mathcal{C}) = L$.*

In other words, a global specification in terms of a regular language can be realized as a distributed implementation if it is closed under $\sim_{\widetilde{\Sigma}}$.

Given an arbitrary concurrent pushdown alphabet $\widetilde{\Sigma}$, let us in the following assume an implicit lexicographic (i.e., strict total) ordering $<_{\text{lex}}$ on Σ. We say that a word $w \in \Sigma^*$ is in *lexicographic normal form* wrt. $<_{\text{lex}}$ if it is minimal wrt. $<_{\text{lex}}$ among all equivalent words. For a set $L \subseteq \Sigma^*$, we write $\text{Min}_{<_{\text{lex}}}(L)$ to denote the set of words from L that are in normal form wrt. $<_{\text{lex}}$. In particular, a word $w \in \Sigma^*$ is in normal form iff $w \in \text{Min}_{<_{\text{lex}}}([w]_{\sim_{\widetilde{\Sigma}}})$.

Theorem 9 (Ochmański [21]). *If $L \subseteq \Sigma^*$ is a regular set of words in lexicographic normal form wrt. $<_{\text{lex}}$, then $[L]_{\sim_{\widetilde{\Sigma}}}$ is regular.*

It will turn out to be useful to consider an MVPA $\mathcal{A} = (S, \Gamma, \delta, \iota, F)$ over $\widetilde{\Sigma}$ as a finite automaton reading letters over the alphabet $\Sigma \times \Gamma$. Recall that δ is a subset of $S \times \Sigma \times \Gamma \times S$. We will now simply interpret a transition $(s, a, A, s') \in \delta$ as the transition $(s, (a, A), s')$ of a finite automaton with state space S, reading the single letter $(a, A) \in \Sigma \times \Gamma$. In this manner, we obtain from $\mathcal{A}$ a finite automaton, denoted by $\mathcal{F}_{\mathcal{A}}$, which recognizes a regular word language $L(\mathcal{F}_{\mathcal{A}})$ over $\Sigma \times \Gamma$. Though $L(\mathcal{A})$ is in general not even context-free, we can provide a link between $L(\mathcal{A})$ and $L(\mathcal{F}_{\mathcal{A}})$. Indeed, $L(\mathcal{A})$ contains the projections of words from $L(\mathcal{F}_{\mathcal{A}})$ onto their first component if we restrict to *well-formed* words.

In a well-formed word, we take into account that the stack symbols from Γ must obey a pushdown-stack policy. Towards the definition of a well-formed word, we first call a word from Σ^* *p-well-matched* (wrt. $\widetilde{\Sigma}$), for some process $p \in \mathit{Proc}$, if it is generated by the grammar

$$\mathrm{N} ::= a\,\mathrm{N}\,b \mid \mathrm{N}\,\mathrm{N} \mid \varepsilon \mid c$$

where $a \in \Sigma_p^c$, $b \in \Sigma_p^r$, and $c \in \Sigma \setminus (\Sigma_p^c \cup \Sigma_p^r)$. Intuitively, we require that the restriction to symbols from $\Sigma_p^c \cup \Sigma_p^r$ is parenthesized correctly when interpreting pushs as opening and pops as closing brackets. Now suppose $w = a_1 \dots a_n \in \Sigma^*$. For $i, j \in \{1, \dots, n\}$, we call (i, j) a *matching pair* in w if $i < j$ and there is $p \in \mathit{Proc}$ such that $a_i \in \Sigma_p^c$, $a_j \in \Sigma_p^r$, and $a_{i+1} \dots a_{j-1}$ is p-well-matched. A position $i \in \{1, \dots, n\}$ is called *unmatched* in w if, for every $j \in \{1, \dots, n\}$, neither (i, j) nor (j, i) is a matching pair. We call a word $(a_1, A_1) \dots (a_n, A_n) \in (\Sigma \times \Gamma)^*$ well-formed if

- for each matching pair (i, j) in $a_1 \dots a_n$, we have $A_i = A_j$,
- for all $i \in \{1, \dots, n\}$ such that $a_i \in \Sigma^c$, we have $A_i \neq \bot$, and
- for all $i \in \{1, \dots, n\}$ such that $a_i \in \Sigma^r$ and i is unmatched in $a_1 \dots a_n$, we have $A_i = \bot$.

The notion of a well-formed word will always refer to the concurrent pushdown alphabet $\widetilde{\Sigma}$. We provide a projection mapping $\pi : 2^{(\Sigma \times \Gamma)^*} \to 2^{\Sigma^*}$, which filters from an argument $L \subseteq (\Sigma \times \Gamma)^*$ all the well-formed words and then abstracts away the symbols from Γ. Formally, $\pi(L) = \{w \mid (w, W) \in L \text{ is well-formed}\}$ (here and in the following, we may write a word $(a_1, A_1) \dots (a_n, A_n) \in (\Sigma \times \Gamma)^*$ as the pair $(a_1 \dots a_n, A_1 \dots A_n)$).

One can easily establish the following link between MVPA and the associated finite automaton.

Proposition 10. *For every* MVPA $\mathcal{A}$ *over* $\widetilde{\Sigma}$*, we have* $L(\mathcal{A}) = \pi(L(\mathcal{F}_\mathcal{A}))$.

Proof. Let $\mathcal{A}$ be an MVPA over $\widetilde{\Sigma}$. For $w = a_1 \dots a_n \in L(\mathcal{A})$, there is an accepting run of $\mathcal{A}$ on w. Thus, every position $i \in \{1, \dots, n\}$ has to conform to a transition (s_{i-1}, a_i, A_i, s_i) such that the requirements **[Push]**, **[Pop]**, and **[Internal]** of the definition of an accepting run are met: If $a_i \in \Sigma^c$, then $A_i \neq \bot$; if i is an unmatched position such that $a_i \in \Sigma^r$, then $A_i = \bot$; if (i, j) is a matching pair in w, then $A_i = A_j \neq \bot$, as j is necessarily the position where the symbol A_i, which has been pushed at position i, is popped. Thus, $(a_1, A_1) \dots (a_n, A_n)$ fits in with the definition of a well-formed word. Moreover, $(a_1, A_1) \dots (a_n, A_n)$ is contained in $L(\mathcal{F}_\mathcal{A})$.

If, conversely, $(a_1, A_1) \dots (a_n, A_n) \in L(\mathcal{F}_\mathcal{A})$ is well-formed, then this gives rise to a run of $\mathcal{A}$ on w visiting the same states as $\mathcal{F}_\mathcal{A}$ does in its accepting run on $(a_1, A_1) \dots (a_n, A_n)$. In particular, as $A_i = A_j \neq \bot$ for all matching pairs (i, j) in w, the transition taken at j provides the stack symbol that is on top of the stack of the unique process p such that $a_j \in \Sigma_p^r$. □

Recall that the intersection of a regular and a context-free language is context-free. As emptiness of MVPA is undecidable, we obtain as a corollary that the set of well-formed words is not context-free. We note without giving the proof that it is, however, context-sensitive. Indeed, it is decidable if a given word over $\Sigma \times \Gamma$ is well-formed.

The following lemma is crucial for the task of translating MVPA into CVPA. It constitutes an extension of Theorem 9 to our recursive setting.

Lemma 11. *Let $\mathcal{A}$ be an* MVPA *over $\widetilde{\Sigma}$ (say with set of states S) satisfying $\mathrm{Min}_{<_{\mathrm{lex}}}([L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}) \subseteq L(\mathcal{A})$. There is a* CVPA *$\mathcal{C} = ((S_p)_{p\in Proc}, \Gamma, (\delta_a)_{a\in\Sigma}, \iota, F)$ over $\widetilde{\Sigma}$ such that $L(\mathcal{C}) = [L(\mathcal{A})]_{\widetilde{\Sigma}}$. Moreover, for all $p \in Proc$, the size of S_p is exponential in $|Proc|$, doubly exponential in $|S|$, and triply exponential in $|\Sigma|$.*

In other words: If $L(\mathcal{A})$ contains, for every word $w \in L(\mathcal{A})$, the normal form of w wrt. $<_{\mathrm{lex}}$, then there is some CVPA recognizing the closure of $L(\mathcal{A})$.

Proof. In the proof, we will basically interpret a given MVPA over $\widetilde{\Sigma}$ as an MVPA over a simplified concurrent pushdown alphabet so that Theorems 8 and 9 can be applied. In turn, the resulting automaton will be considered as a CVPA over $\widetilde{\Sigma}$ and will indeed have the desired property.

So let $\mathcal{A} = (S, \Gamma, \delta, \iota, F)$ be an MVPA over $\widetilde{\Sigma}$ such that $\mathrm{Min}_{<_{\mathrm{lex}}}([L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}) \subseteq L(\mathcal{A})$. We define a concurrent pushdown alphabet $\widetilde{\Omega} = ((\emptyset, \emptyset, \Sigma_p \times \Gamma))_{p\in Proc}$. In particular, we have $\Omega = \Sigma \times \Gamma$. Note that, for every $(a, A), (b, B) \in \Omega$, $(a, A) \sim_{\widetilde{\Omega}} (b, B)$ iff $a \sim_{\widetilde{\Sigma}} b$. Recall that we had fixed a lexicographic ordering $<_{\mathrm{lex}}$ on Σ. Now consider any lexicographic ordering $<'_{\mathrm{lex}} \subseteq \Omega \times \Omega$ such that, for every $(a, A), (b, B) \in \Omega$, $a <_{\mathrm{lex}} b$ implies $(a, A) <'_{\mathrm{lex}} (b, B)$. Let *LexNF* denote the set of all words $x \in \Omega^*$ that are in lexicographic normal form wrt. $<'_{\mathrm{lex}}$, i.e., such that $x \in \mathrm{Min}_{<'_{\mathrm{lex}}}([x]_{\sim_{\widetilde{\Omega}}})$. This set forms a regular word language [11, 16] so that the intersection $L(\mathcal{F}_{\mathcal{A}}) \cap \mathit{LexNF}$ is regular, too.

According to Theorem 9, $[L(\mathcal{F}_{\mathcal{A}}) \cap \mathit{LexNF}]_{\sim_{\widetilde{\Omega}}}$ is regular, and Theorem 8 tells us that there is a CVPA $\mathcal{C}$ over the concurrent pushdown alphabet $\widetilde{\Omega}$ such that $L(\mathcal{C}) = [L(\mathcal{F}_{\mathcal{A}}) \cap \mathit{LexNF}]_{\sim_{\widetilde{\Omega}}}$. From $\mathcal{C}$, we obtain an MVPA $\mathcal{C}'$ over $\widetilde{\Sigma}$ with stack alphabet Γ by transforming a transition $(s, (a, A), \mathrm{B}, s')$ into a transition (s, a, A, s') (recall that (a, A) is necessarily contained in Ω^{int} so that B can indeed be neglected). Observe that $\mathcal{C}'$ is actually a CVPA. As $L(\mathcal{F}_{\mathcal{C}'}) = L(\mathcal{C})$ and, by Proposition 10, $\pi(L(\mathcal{F}_{\mathcal{C}'})) = L(\mathcal{C}')$, we deduce $L(\mathcal{C}') = \pi(L(\mathcal{C}))$. So it remains to show that $[L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}} = \pi(L(\mathcal{C}))$.

Suppose $w \in [L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$. We chose the word $w' \in [w]_{\sim_{\widetilde{\Sigma}}}$ that is in lexicographic normal form wrt. $<_{\mathrm{lex}}$. As $\mathrm{Min}_{<_{\mathrm{lex}}}([L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}) \subseteq L(\mathcal{A})$, we have $w' \in L(\mathcal{A})$. Thus, there must be $W' \in \Gamma^*$ such that (w', W') is well-formed and contained in $L(\mathcal{F}_{\mathcal{A}})$ (Proposition 10). As w' is in lexicographic normal form wrt. $<_{\mathrm{lex}}$ and as $<'_{\mathrm{lex}}$ is kind of an extension of $<_{\mathrm{lex}}$, (w', W') is in lexicographic normal form wrt. $<'_{\mathrm{lex}}$ so that $(w', W') \in \mathit{LexNF}$. We can now reorder (w', W') in such a way that its first component becomes w. Formally, there is $W \in \Gamma^*$ such that $(w, W) \sim_{\widetilde{\Omega}} (w', W')$. As every word from $[(w', W')]_{\sim_{\widetilde{\Omega}}}$ is well-formed, so is (w, W), and we conclude $w \in \pi([L(\mathcal{F}_{\mathcal{A}}) \cap \mathit{LexNF}]_{\sim_{\widetilde{\Omega}}})$.

Now suppose $w \in \pi([L(\mathcal{F}_\mathcal{A}) \cap \mathit{LexNF}]_{\sim_{\widetilde{\Omega}}})$. We can find an extension $W \in \Gamma^*$ of w such that (w, W) is well-formed and contained in $[L(\mathcal{F}_\mathcal{A}) \cap \mathit{LexNF}]_{\sim_{\widetilde{\Omega}}}$. Thus, there is $(w', W') \in L(\mathcal{F}_\mathcal{A}) \cap \mathit{LexNF}$ such that $(w', W') \sim_{\widetilde{\Omega}} (w, W)$, which implies $w' \sim_{\widetilde{\Sigma}} w$. Note that (w', W') is well-formed, too, so that, with Proposition 10, $w' \in L(\mathcal{A})$. We conclude $w \in [L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$.

Let us analyze the size of $\mathcal{C}'$. For this, we need to introduce two notions concerning finite automata over Ω. A finite automaton is called *loop-connected* if, for every nonempty word $a_1 \dots a_n \in \Omega^*$ labeling a path from a state s back to state s, the graph (V, E) is connected, where $V = \{a_i \mid i \in \{1, \dots, n\}\}$ and $E = (V \times V) \setminus I_{\sim_{\widetilde{\Omega}}}$. It is said to be *$I$-diamond* if, for all $(\alpha, \beta) \in I_{\sim_{\widetilde{\Omega}}}$ and transitions $r \xrightarrow{\alpha} s \xrightarrow{\beta} t$, we have transitions $r \xrightarrow{\beta} s' \xrightarrow{\alpha} t$ for some state s'. From [16], we know that there is a deterministic loop connected finite automaton $\mathcal{B}_1$ over Ω with $(|\Sigma| + 1)!$ many states that recognizes the set LexNF. The set of states of $\mathcal{F}_\mathcal{A}$ is the same as that of $\mathcal{A}$ so that we obtain, as the product of $\mathcal{F}_\mathcal{A}$ and $\mathcal{B}_1$, a finite automaton $\mathcal{B}_2$ of size

$$n := |S| \cdot (|\Sigma| + 1)!$$

recognizing $L(\mathcal{F}_\mathcal{A}) \cap \mathit{LexNF}$. As $\mathcal{B}_1$ is loop-connected, so is $\mathcal{B}_2$. According to [16, 20], there is an I-diamond finite automaton $\mathcal{B}_3$ over Ω with

$$N := (n^2 \cdot 2^{|\Sigma|})^{(n-1)(|\Sigma|+1)+1}$$

many states that recognizes $[L(\mathcal{B}_2)]_{\sim_{\widetilde{\Omega}}}$. In the next step, we constructed, from $\mathcal{B}_3$, a Cvpa $\mathcal{C} = ((S'_p)_{p \in \mathit{Proc}}, \Gamma', (\delta'_a)_{a \in \Sigma}, \iota', F')$ such that $L(\mathcal{C}) = L(\mathcal{B}_3)$. From [14], we know that the sum $\sum_{p \in \mathit{Proc}} |S'_p|$ can be bounded by

$$2^{N^2 \cdot (|\mathit{Proc}|^2 + |\mathit{Proc}|) + 2|\mathit{Proc}|^4}$$

As $\mathcal{C}'$ and $\mathcal{C}$ have the same local states, we conclude that the number of local states of $\mathcal{C}'$ is exponential in $|\mathit{Proc}|$, doubly exponential in $|S|$, and triply exponential in $|\Sigma|$. □

Since $L(\mathcal{A}) = [L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$ implies $\mathrm{Min}_{<_{\mathrm{lex}}}([L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}) \subseteq L(\mathcal{A})$, we obtain, by Lemma 11, the following extension of Zielonka's Theorem.

Theorem 12. *Let $\mathcal{A}$ be an* Mvpa *over $\widetilde{\Sigma}$ (say with set of states S) such that $L(\mathcal{A})$ is $\sim_{\widetilde{\Sigma}}$-closed. There is a* Cvpa $\mathcal{C} = ((S_p)_{p \in \mathit{Proc}}, \Gamma, (\delta_a)_{a \in \Sigma}, \iota, F)$ *over $\widetilde{\Sigma}$ satisfying $L(\mathcal{C}) = L(\mathcal{A})$. For all $p \in \mathit{Proc}$, the size of S_p is exponential in $|\mathit{Proc}|$, doubly exponential in $|S|$, and triply exponential in $|\Sigma|$.*

This result demonstrates that Mvpa recognizing a $\sim_{\widetilde{\Sigma}}$-closed language are suitable specifications for Cvpa. Unfortunately, it is in general undecidable if an Mvpa has this property. However, a restriction to k-phase words will allow us to define a decidable sufficient criterion for the transformation of an Mvpa into a Cvpa. Next, we state a Zielonka-like theorem that is tailored to this restriction. There, we require that an Mvpa *represents* the k-phase words of a system, while the final implementation can produce non-k-phase executions. The following definition will clarify what we mean by a representation.

Definition 13. *For $k \in \mathbb{N}$, we call a language $L \subseteq \mathrm{W}_k(\widetilde{\Sigma})$ a* k-phase representation *if, for all $u, v \in \Sigma^*$ and $(a, b) \in I_{\widetilde{\Sigma}}$ with $\{uabv, ubav\} \subseteq \mathrm{W}_k(\widetilde{\Sigma})$, we have $uabv \in L$ iff $ubav \in L$.*

Next, we show that the closure of a k-phase representation that is given by an MVPA can be realized as a CVPA.

Theorem 14. *Let $k \in \mathbb{N}$ and let $\mathcal{A}$ be an* MVPA *over $\widetilde{\Sigma}$ (say with set of states S) such that $L_k(\mathcal{A})$ is a k-phase representation. There is a* CVPA *$\mathcal{C} = ((S_p)_{p \in Proc}, \Gamma, (\delta_a)_{a \in \Sigma}, \iota, F)$ over $\widetilde{\Sigma}$ such that $L(\mathcal{C}) = [L_k(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$. Moreover, for all $p \in Proc$, the size of S_p is doubly exponential in $|Proc|$, $|S|$, and k, and triply exponential in $|\Sigma|$.*

Proof. Again, we exploit Lemma 11. Unlike in Theorem 12, we cannot apply it directly, as there is no way to define the lexicographic ordering $<_{\mathrm{lex}}$ in such a way that $\mathrm{Min}_{<_{\mathrm{lex}}}([L_k(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}) \subseteq L_k(\mathcal{A})$ if $L_k(\mathcal{A})$ is a k-phase representation. Our trick is to extend $\widetilde{\Sigma}$ by a component that indicates the current phase of a letter. An appropriate definition of a normal form over this extended alphabet will then allow us to apply Lemma 11.

So let $k \in \mathbb{N}$ and let $\mathcal{A} = (S, \Gamma, \delta, \iota, F)$ be an MVPA over $\widetilde{\Sigma}$ such that $L_k(\mathcal{A})$ is a k-phase representation. Without loss of generality, we assume $L_k(\mathcal{A}) = L(\mathcal{A})$.

Based on $\widetilde{\Sigma}$, we define a new concurrent pushdown alphabet $\widetilde{\Omega}$ by $\Omega_p^c = \Sigma_p^c \times \{1, \ldots, k\}$, $\Omega_p^r = \Sigma_p^r \times \{1, \ldots, k\}$, and $\Omega_p^{int} = \Sigma_p^{int} \times \{1, \ldots, k\}$ for all $p \in Proc$. From $\mathcal{A}$, one can construct an MVPA $\mathcal{B}$ over $\widetilde{\Omega}$ accepting the words $(a_1, ph_1) \ldots (a_n, ph_n)$ such that both $a_1 \ldots a_n \in L(\mathcal{A})$ and, for all $i \in \{1, \ldots, n\}$, $ph_i = \min\{j \in \{1, \ldots, k\} \mid a_1 \ldots a_i \text{ is a } j\text{-phase word}\}$. Intuitively, the additional components ph_i give rise to a unique *tight* factorization of $a_1 \ldots a_n$ into phases (cf. [17]). Now consider any lexicographic ordering $<'_{\mathrm{lex}} \subseteq \Omega \times \Omega$ such that $i < j$ implies $(a, i) <'_{\mathrm{lex}} (b, j)$ and, moreover, $a <_{\mathrm{lex}} b$ implies $(a, i) <'_{\mathrm{lex}} (b, i)$. We claim that $L(\mathcal{B})$ contains, for every word $x \in L(\mathcal{B})$, the normal form of x wrt. $<'_{\mathrm{lex}}$. Indeed $x \in L(\mathcal{B})$ can be written as a concatenation $x_1 \cdot \ldots \cdot x_k$ with $x_i \in (\Sigma \times \{i\})^*$ for all $i \in \{1, \ldots, k\}$. I.e., for two letters α and β occurring in x_i and, respectively, x_j with $i < j$, we have $\alpha <'_{\mathrm{lex}} \beta$. In particular, the normal form of x can be obtained by reordering letters within the factors x_i, i.e., $\mathrm{Min}_{<'_{\mathrm{lex}}}([x]_{\sim_{\widetilde{\Omega}}}) \subseteq \mathrm{Min}_{<'_{\mathrm{lex}}}([x_1]_{\sim_{\widetilde{\Omega}}}) \cdot \ldots \cdot \mathrm{Min}_{<'_{\mathrm{lex}}}([x_k]_{\sim_{\widetilde{\Omega}}})$. Note that the reordering does not increase the number of phases. As $L_k(\mathcal{A})$ is a k-phase representation, the reordering also preserves containment in $L(\mathcal{B})$ and we have $\mathrm{Min}_{<'_{\mathrm{lex}}}([x]_{\sim_{\widetilde{\Omega}}}) \subseteq L(\mathcal{B})$. By Lemma 11, there is a CVPA $\mathcal{C}$ over $\widetilde{\Omega}$ with $L(\mathcal{C}) = [L(\mathcal{B})]_{\sim_{\widetilde{\Omega}}}$. It is easy to see that the class of languages of CVPA over $\widetilde{\Omega}$ is closed under the projection from $\widetilde{\Omega}$ to $\widetilde{\Sigma}$ that is induced by the function $f : \Omega \to \Sigma$ given by $f((a, i)) = a$ (this was shown for MVPA in [17]). Thus, there is a CVPA $\mathcal{C}'$ over $\widetilde{\Sigma}$ such that $L(\mathcal{C}') = f([L(\mathcal{B})]_{\sim_{\widetilde{\Omega}}})$ (where f is canonically extended to words and, then, to languages). As $f([L(\mathcal{B})]_{\sim_{\widetilde{\Omega}}}) = [f(L(\mathcal{B}))]_{\sim_{\widetilde{\Sigma}}} = [L(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$, we are done.

To establish the upper bound of the number of local states, observe that the size of $\mathcal{B}$ can be bounded by $|S| \cdot |Proc| \cdot (k + 1)$. The rest of the construction follows that from the proof of Lemma 11. □

Remark 15. The transformations in the proofs of Lemma 11 and Theorems 12 and 14 are effective. In particular, one can explicitly give a decomposition of states and transitions of the CVPA, as required in Definition 5.

When we restrict to k-phase words, it is actually decidable whether the previous theorems are applicable to a given MVPA:

Theorem 16. *The following problems are decidable in elementary time:*
INPUT*: Concurrent pushdown alphabet* $\widetilde{\Sigma}$*;* $k \in \mathbb{N}$*;* MVPA $\mathcal{A}$ *over* $\widetilde{\Sigma}$*.*
QUESTION 1*: Is* $L_k(\mathcal{A}) \sim_{\widetilde{\Sigma}}$*-closed?*
QUESTION 2*: Is* $L_k(\mathcal{A})$ *a* k*-phase representation?*

Proof. Our proof is inspired by [22] where similar problems are addressed in the word setting. Nevertheless, the main difficulty in our proof arises from the presence of stacks.

We first show decidability of Question 1. Let $k \in \mathbb{N}$ and let furthermore $\mathcal{A}_1 = (S_1, \Gamma_1, \delta_1, \iota_1, F_1)$ be the MVPA over $\widetilde{\Sigma}$ in question. By Theorem 4, one can obtain from $\mathcal{A}_1$ a further MVPA $\mathcal{A}_2 = (S_2, \Gamma_2, \delta_2, \iota_2, F_2)$ over $\widetilde{\Sigma}$ such that $L(\mathcal{A}_2) = \overline{L_k(\mathcal{A}_1)}$. We will now construct an MVPA $\mathcal{A}$ over $\widetilde{\Sigma}$ recognizing words of the form $uabv$ with $u, v \in \Sigma^*$, $(a, b) \in I_{\widetilde{\Sigma}}$, and both $uabv \in L(\mathcal{A}_1)$ and $ubav \in L(\mathcal{A}_2)$. Thus, if $L(\mathcal{A})$ contains a k-phase word $uabv$, then $uabv$ is contained in $L_k(\mathcal{A}_1)$ and $ubav$ (which is a $(k+2)$-phase word) is equivalent to $uabv$, but not contained in $L_k(\mathcal{A}_1)$. Indeed, $L_k(\mathcal{A}_1) \neq [L_k(\mathcal{A}_1)]_{\sim_{\widetilde{\Sigma}}}$ iff $L_k(\mathcal{A}) \neq \emptyset$. The latter question is decidable (Theorem 3).

The set of states of $\mathcal{A}$ is $S = S_1 \times S_2 \times (\{0, 1\} \cup (I_{\widetilde{\Sigma}} \times \Gamma_1 \times \Gamma_2))$. Basically, the first component of a state from S is used to simulate $\mathcal{A}_1$, while the second component simulates $\mathcal{A}_2$. The third component starts in 0. While being in states of the form $(s_1, s_2, 0)$, both automata proceed synchronously: Reading an action a, the global automaton $\mathcal{A}$ applies a-transitions $(s_1, a, A_1, s_1') \in \delta_1$ and $(s_2, a, A_2, s_2') \in \delta_2$ to the first and the second component, respectively, resulting in a global step $((s_1, s_2, 0), a, (A_1, A_2), (s_1', s_2', 0))$. Note that the stack alphabet is extended to $\Gamma_1 \times \Gamma_2$ to take into account that the local transitions may come along with different stack symbols A_1 and A_2.

When reading an input word, the first component should eventually perform an action sequence ab with $(a, b) \in I_{\widetilde{\Sigma}}$, while the second component executes the permutation ba. Thus, we are faced with the problem that the automaton $\mathcal{A}$ has to read simultaneously an a (namely, in the first component) and a b (in the second component). Hereby, a might be a push operation, while b is a pop, and even if both a and b are contained in Σ^c, it can happen that their corresponding pop operations are far away from each other so that the corresponding pops cannot be performed synchronously. We propose the following solution: Suppose $\mathcal{A}$ is about to simulate transitions (s_1, a, A_1, s_1') followed by (s_1', b, B_1, s_1'') in the first component (i.e., in $\mathcal{A}_1$) and (s_2, b, B_2, s_2') followed by (s_2', a, A_2, s_2'') in the second component (i.e., in $\mathcal{A}_2$). The global automaton $\mathcal{A}$ will produce this transition sequence "crosswise". It will first read the a and apply the transition involving the stack symbol $A_1 \in \Gamma_1$ to the first component. At the same time, the second

component only changes its local state into s_2'. As the stack symbol B_2 cannot be applied directly, it is stored in the third component of the subsequent global state of $\mathcal{A}$, which is of the form $(s_1', s_2', ((a,b), B_2, A_2))$. Observe that the stack symbol A_2, which is associated to executing a in the second component, must be applied together with reading a so that (A_1, A_2) acts as the stack symbol. Since a corresponding local transition (s_2', a, A_2, s_2'') has to follow in the second component, the A_2 needs to be stored as well. The formal description of this step can be found below (1). Now, being in the global state $(s_1', s_2', ((a,b), B_2, A_2))$, $\mathcal{A}$ will, according to the local transition (s_1', b, B_1, s_1''), perform a b and apply (B_1, B_2) to the designated stack. Again, the second component will only change its local state into s_2''. However, the local transition has to conform to the symbol A_2 that had been stored (recall that the A_2 had already been applied during the execution of a in $\mathcal{A}_1$). This step corresponds to rule (2) below. We are now in a global state of the form $(s_1'', s_2'', 1)$. In states with 1 in the third position, $\mathcal{A}_1$ and $\mathcal{A}_2$ again act simultaneously.

Formally, $\mathcal{A} = (S, \Gamma, \delta, \iota, F)$ is given by $S = S_1 \times S_2 \times (\{0,1\} \cup (I_{\widetilde{\Sigma}} \times \Gamma_1 \times \Gamma_2))$, $\Gamma = \Gamma_1 \times \Gamma_2$, $\iota = (\iota_1, \iota_2, 0)$, and $F = F_1 \times F_2 \times \{1\}$. Let $(s_1, s_2, \sigma), (s_1', s_2', \sigma') \in S$, $a \in \Sigma$, and $(A_1, A_2) \in \Gamma$. Then, $((s_1, s_2, \sigma), a, (A_1, A_2), (s_1', s_2', \sigma')) \in \delta$ if there are $(B_1, B_2) \in \Gamma$ and $b \in \Sigma$ such that one of the following holds:

(1) $(\sigma = \sigma' = 0$ or $\sigma = \sigma' = 1)$, $(s_1, a, A_1, s_1') \in \delta_1$, and $(s_2, a, A_2, s_2') \in \delta_2$, or
(2) $\sigma = 0$, $\sigma' = ((a,b), B_2, A_2)$, $(s_1, a, A_1, s_1') \in \delta_1$, and $(s_2, b, B_2, s_2') \in \delta_2$, or
(3) $\sigma' = 1$, $\sigma = ((b,a), A_2, B_2)$, $(s_1, a, A_1, s_1') \in \delta_1$, and $(s_2, b, B_2, s_2') \in \delta_2$.

The only difference in the decision procedure for Question 2 is that $\mathcal{A}_2$ is such that $L(\mathcal{A}_2) = L_k(\mathcal{A}_2) = \overline{L_k(\mathcal{A}_1)} \cap \mathrm{W}_k(\widetilde{\Sigma})$.

An inspection of the constructions from [17] tells us that the size of $\mathcal{A}_2$ is in both cases triply exponential in $|Proc|$, $|S_1|$, and k. As emptiness of MVPA wrt. k-phase words is decidable in doubly exponential time, we obtain elementary decision procedures for Question 1 and Question 2. □

4 Specifying Programs in MSO Logic

In Section 3, we considered the language L of an MVPA to be a specification, and our aim was to find a CVPA $\mathcal{C}$ such that $L(\mathcal{C}) = [L]_{\sim_{\widetilde{\Sigma}}}$. Unfortunately, one cannot always find such a CVPA (consider, e.g., $L = (ab)^*$ with $(a,b) \in I_{\widetilde{\Sigma}}$). We now present a specification language that operates directly on equivalence classes of $\sim_{\widetilde{\Sigma}}$ so that, provided that we restrict to k-phase words, any specification can be realized as a CVPA. In doing so, we extend the classical connection between monadic second-order (MSO) logic and finite automata [6, 28]. This study of relations between logical formalisms that may serve as a specification language and automata has had many generalizations, including MVPA [17].

Actually, we present an MSO logic that is interpreted over partial orders, which arise naturally from words in the presence of a concurrent pushdown alphabet and the induced independence relation. Any such partial order represents one equivalence class of words so that a formula defines a set of equivalence classes or, in other words, a set of words that is $\sim_{\widetilde{\Sigma}}$-closed.

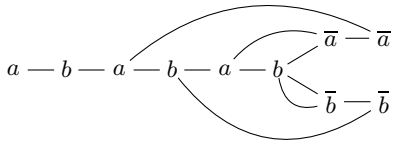

Fig. 2. A visibly pushdown trace

Let $w = a_1 \dots a_n \in \Sigma^*$. With w, we associate the labeled structure $T_{\widetilde{\Sigma}}(w) = (E, \preceq, \mu, \lambda)$, where $E = \{1, \dots, n\}$ is the set of *events*, $\lambda : E \to \Sigma$ assigns to any event $i \in E$ the action $\lambda(i) = a_i$ it executes, and $\mu \subseteq E \times E$ contains the matching pairs in w (i.e., $(i, j) \in \mu$ iff (i, j) is a matching pair). Finally, $\preceq \subseteq E \times E$ is a partial-order relation (i.e., it is reflexive, transitive, and antisymmetric), which is defined to be the transitive closure of $\{(i, j) \in E \times E \mid i \leq j \text{ and } (a_i, a_j) \notin I_{\widetilde{\Sigma}}\}$. We call the structure $T_{\widetilde{\Sigma}}(w)$ that arises from a word $w \in \Sigma^*$ a *(visibly pushdown) trace* over $\widetilde{\Sigma}$. The set of traces over $\widetilde{\Sigma}$ is denoted by $\mathrm{Tr}(\widetilde{\Sigma})$. It is standard to prove that $T_{\widetilde{\Sigma}}(w) = T_{\widetilde{\Sigma}}(w')$ iff $w \sim_{\widetilde{\Sigma}} w'$ where we consider equality of traces up to isomorphism. In other words, there is a one-to-one correspondence between traces and equivalence classes of $\sim_{\widetilde{\Sigma}}$. We remark that visibly pushdown traces are a merge of Mazurkiewicz traces [11] and *nested words* [3], which, in turn, generalize themselves the notion of a word.

Example 17. Figure 2 depicts $T = T_{\widetilde{\Sigma}}(a\,b\,a\,b\,a\,b\,\overline{a}\,\overline{a}\,\overline{b}\,\overline{b}) = T_{\widetilde{\Sigma}}(a\,b\,a\,b\,a\,b\,\overline{b}\,\overline{b}\,\overline{a}\,\overline{a})$ where $\widetilde{\Sigma}$ is taken from Example 1. Hereby, the straight edges form the cover relation $\preceq \setminus \preceq^2$ of the underlying partial-order relation $\preceq$, and the curved edges represent μ, i.e., the matching pairs. There are two unmatched events in T.

Fixing supplies of first-order variables $x, y, \dots$ and second-order variables $X, Y, \dots$, the syntax of our MSO logic complies with the signature of a trace. Formally, formulas from $\mathrm{MSO}(\widetilde{\Sigma})$ are given by the grammar

$$\varphi ::= x \preceq y \mid (x, y) \in \mu \mid \lambda(x) = a \mid x \in X \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x \varphi \mid \exists X \varphi$$

where x and y are first-order variables, X is a second-order variable, and $a \in \Sigma$. Moreover, we will make use of the usual abbreviations such as $\varphi_1 \wedge \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and $\forall x \varphi$. To determine the semantics, let $T = (E, \preceq, \mu, \lambda)$ be a trace over $\widetilde{\Sigma}$ and $\mathbb{I}$ be an interpretation function, which assigns to a first-order variable an element from E and to a second-order variable a subset of E. Let us define when $T, \mathbb{I} \models \varphi$ for $\varphi \in \mathrm{MSO}(\widetilde{\Sigma})$. Namely, $T, \mathbb{I} \models x \preceq y$ if $\mathbb{I}(x) \preceq \mathbb{I}(y)$, $T, \mathbb{I} \models (x, y) \in \mu$ if $(\mathbb{I}(x), \mathbb{I}(y)) \in \mu$, and $T, \mathbb{I} \models \lambda(x) = a$ if $\lambda(\mathbb{I}(x)) = a$. The rest of the semantics is classical for MSO logics. If φ is a sentence, i.e., a formula without free variables, we can write $T \models \varphi$ if $T, \mathbb{I} \models \varphi$ for some interpretation function $\mathbb{I}$. Now, given a sentence $\varphi \in \mathrm{MSO}(\widetilde{\Sigma})$, we set $\mathscr{L}(\varphi) = \{T \in \mathrm{Tr}(\widetilde{\Sigma}) \mid T \models \varphi\}$ to be the set of traces that satisfy φ.

As the language of a CVPA $\mathcal{C}$ is closed under $\sim_{\widetilde{\Sigma}}$, it is legitimate to assign to $\mathcal{C}$ a set of traces, too, letting $\mathscr{L}(\mathcal{C}) = \{T_{\widetilde{\Sigma}}(w) \mid w \in L(\mathcal{C})\}$.

Example 18. Suppose T to be the trace given in Figure 2 and consider the sentences $\varphi_1 = \forall x\,((\lambda(x) = a \vee \lambda(x) = b) \to \exists y\,(x,y) \in \mu)$ expressing that there is no pending call, and $\varphi_2 = \forall x\,((\lambda(x) = \overline{a} \vee \lambda(x) = \overline{b}) \to \exists y\,(y,x) \in \mu)$, which expresses that there is no pending return. We have $T \notin \mathscr{L}(\varphi_1)$ but $T \in \mathscr{L}(\varphi_2)$. Note also that $T \in \mathscr{L}(\mathcal{C})$ for the CVPA $\mathcal{C}$ from Example 7 (Figure 1).

Before we look at a logical characterization of general CVPA, let us recall a result that has already been found in the context of asynchronous automata, i.e., of CVPA over simple concurrent pushdown alphabets.

Theorem 19 (Thomas [27]). *Suppose $\Sigma = \Sigma^{int}$ and let $\mathscr{L} \subseteq \mathrm{Tr}(\widetilde{\Sigma})$. Then, $\mathscr{L} = \mathscr{L}(\mathcal{C})$ for some* CVPA *$\mathcal{C}$ over $\widetilde{\Sigma}$ iff $\mathscr{L} = \mathscr{L}(\varphi)$ for some $\varphi \in \mathrm{MSO}(\widetilde{\Sigma})$.*

Now let us turn towards CVPA over general concurrent pushdown alphabets. It has been shown in [4] that MSO logic is in general strictly more expressive than CVPA. We will therefore extend the notion of k-phase words to traces. For $k \in \mathbb{N}$, a trace $T \in \mathrm{Tr}(\widetilde{\Sigma})$ is called a *k-phase trace* if there is $w \in \mathrm{W}_k(\widetilde{\Sigma})$ such that $T_{\widetilde{\Sigma}}(w) = T$. The set of k-phase traces over $\widetilde{\Sigma}$ is denoted by $\mathrm{Tr}_k(\widetilde{\Sigma})$. For example, the trace T from Figure 2 is a 2-phase trace, even though we have $T = T(w)$ for $w = a\,b\,a\,b\,a\,b\,\overline{a}\,\overline{b}\,\overline{a}\,\overline{b} \notin \mathrm{W}_2(\widetilde{\Sigma})$. The domain of k-phase traces is particularly interesting, because it is decidable whether $\mathscr{L}(\mathcal{C}) \cap \mathrm{Tr}_k(\widetilde{\Sigma}) \neq \emptyset$ holds for a CVPA $\mathcal{C}$. To see this, observe that the latter holds iff $L(\mathcal{C}) \cap \mathrm{W}_k(\widetilde{\Sigma}) \neq \emptyset$, which is decidable according to Theorem 3.

For a logical characterization of CVPA, we will need the following lemma.

Lemma 20. *Let $k \in \mathbb{N}$ and let $\mathcal{C}$ be a* CVPA *over $\widetilde{\Sigma}$ such that $\mathscr{L}(\mathcal{C}) \subseteq \mathrm{Tr}_k(\widetilde{\Sigma})$. There is a* CVPA *$\mathcal{C}'$ over $\widetilde{\Sigma}$ such that $\mathscr{L}(\mathcal{C}') = \overline{\mathscr{L}(\mathcal{C})} \cap \mathrm{Tr}_k(\widetilde{\Sigma})$, where $\overline{\mathscr{L}(\mathcal{C})} = \mathrm{Tr}(\widetilde{\Sigma}) \setminus \mathscr{L}(\mathcal{C})$.*

Proof. Let $k \in \mathbb{N}$ and let $\mathcal{C}$ be a CVPA over $\widetilde{\Sigma}$ satisfying $\mathscr{L}(\mathcal{C}) \subseteq \mathrm{Tr}_k(\widetilde{\Sigma})$. Due to Theorem 4, there is an MVPA $\mathcal{A}$ over $\widetilde{\Sigma}$ such that $L_k(\mathcal{A}) = \overline{L_k(\mathcal{C})} \cap \mathrm{W}_k(\widetilde{\Sigma})$. Observe that $L_k(\mathcal{A})$ is a k-phase representation. Thus, by Theorem 14, there is a CVPA $\mathcal{C}'$ over $\widetilde{\Sigma}$ such that $L(\mathcal{C}') = [L_k(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$. One easily verifies that we actually have $\mathscr{L}(\mathcal{C}') = \overline{\mathscr{L}(\mathcal{C})} \cap \mathrm{Tr}_k(\widetilde{\Sigma})$. □

As a corollary, we obtain that, for every $k \in \mathbb{N}$, there is a CVPA $\mathcal{C}$ with $\mathscr{L}(\mathcal{C}) = \mathrm{Tr}_k(\widetilde{\Sigma})$. This is an important fact in the proof of Theorem 22. Indeed, the following two theorems constitute a logical characterization of CVPA (restricted to k-phase words).

Theorem 21. *For every* CVPA *$\mathcal{C}$ over $\widetilde{\Sigma}$, there is a sentence $\varphi \in \mathrm{MSO}(\widetilde{\Sigma})$ such that $\mathscr{L}(\varphi) = \mathscr{L}(\mathcal{C})$.*

Proof. The main idea is standard. By means of second-order variables, one guesses an assignment of states to events, whereupon a first-order formula tests if we actually deal with a run (see [27]). In our extended setting, however, further second-variables represent an assignment of stack symbols to events, which has to conform to the transitions, too. This can also be checked in the first-order part of the formula. The whole construction is given in the appendix. □

Theorem 22. *Let $k \in \mathbb{N}$. For every sentence $\varphi \in \mathrm{MSO}(\widetilde{\Sigma})$, there is a* Cvpa $\mathcal{C}$ *over $\widetilde{\Sigma}$ such that $\mathscr{L}(\mathcal{C}) = \mathscr{L}(\varphi) \cap \mathrm{Tr}_k(\widetilde{\Sigma})$.*

Proof. As usual, we proceed by induction on the structure of an MSO formula. We will perform this induction in some more detail, as treating negation is less obvious than in classical settings such as words and trees. To begin with, however, we follow the classical approach and consider a logic that has only second-order variables in its repertoire but is equivalent to $\mathrm{MSO}(\widetilde{\Sigma})$. Formulas from the logic $\mathrm{MSO}_0(\widetilde{\Sigma})$ are derived by the grammar $\varphi ::= X \preceq Y \mid (X,Y) \in \mu \mid \lambda(X) \subseteq \{a\} \mid X \subseteq Y \mid \mathit{Sing}(X) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists X \varphi$ where X and Y are second-order variables and $a \in \Sigma$. The formula $\mathit{Sing}(X)$ is valid in a trace if X is interpreted as a singleton. Formula $X \preceq Y$ holds if X and Y are singletons $\{i\}$ and $\{j\}$, respectively, such that $i \preceq j$. The semantics of $(X,Y) \in \mu$ is defined similarly, and all the other operators correspond to those from $\mathrm{MSO}(\widetilde{\Sigma})$. It is easy to show that $\mathrm{MSO}(\widetilde{\Sigma})$ and $\mathrm{MSO}_0(\widetilde{\Sigma})$ are expressively equivalent so that we can proceed by an inductive translation of a formula $\varphi(Y_1, \ldots, Y_n) = (\exists/\neg\exists)X_m \ldots (\exists/\neg\exists)X_1 \psi(Y_1, \ldots, Y_n, X_m, \ldots, X_1) \in \mathrm{MSO}_0(\widetilde{\Sigma})$ with quantifier-free ψ into a Cvpa. Without loss of generality, we assume $n \geq 1$. Then, the formula φ naturally defines a set of traces over the concurrent pushdown alphabet ${}_n\widetilde{\Omega}$ where ${}_n\Omega_p^c = \Sigma_p^c \times \{0,1\}^n$, ${}_n\Omega_p^r = \Sigma_p^r \times \{0,1\}^n$, and ${}_n\Omega_p^{int} = \Sigma_p^{int} \times \{0,1\}^n$ for each $p \in \mathit{Proc}$. Namely, we consider a trace over ${}_n\widetilde{\Omega}$ to be a trace over $\widetilde{\Sigma}$ plus an interpretation of the variables $Y_1, \ldots, Y_n$ where 1 in the i-th component indicates containment in Y_i. The crucial point is now that φ is translated into a Cvpa $\mathcal{C}$ over ${}_n\widetilde{\Omega}$ such that $\mathscr{L}(\mathcal{C}) = \mathscr{L}(\varphi) \cap \mathrm{Tr}_k({}_n\widetilde{\Omega})$. This restriction to k-phase traces is an invariant that ensures that complementation of the current automaton is always possible. Transforming atomic formulas is easy, though technically tedious. At this point, it is also important to note that Cvpa are closed under intersection and that, by Lemma 20, there is a Cvpa recognizing $\mathrm{Tr}_k({}_n\widetilde{\Omega})$. To get a Cvpa for $\neg\varphi$, let $k \in \mathbb{N}$ and suppose that we already have a Cvpa $\mathcal{C}$ over ${}_n\widetilde{\Omega}$ such that $\mathscr{L}(\mathcal{C}) = \mathscr{L}(\varphi) \cap \mathrm{Tr}_k({}_n\widetilde{\Omega})$. By Lemma 20, there is a Cvpa $\mathcal{C}'$ such that $\mathscr{L}(\mathcal{C}') = \overline{\mathscr{L}(\mathcal{C})} \cap \mathrm{Tr}_k({}_n\widetilde{\Omega})$. The latter equals $\mathscr{L}(\neg\varphi) \cap \mathrm{Tr}_k({}_n\widetilde{\Omega})$ so that we are done. Like atomic formulas, disjunction and existential quantification follow standard methods, which use the fact that Cvpa are closed under union and projection (cf. [28]). □

Remark 23. The transformations from the proofs of Theorems 21 and 22 are effective. Note that the size of a Cvpa that arises from a formula cannot be bounded by an elementary function. This is, however, already the case in the setting of finite automata over words, even if we restrict to first-order formulas (with the transitive closure predicate $x \leq y$) [28].

5 Future Directions

Though the results in this paper are of rather theoretical nature, due to the high complexity of our constructions, we believe that Cvpa and the related notion

of a visibly pushdown trace may open a new branch in concurrency theory. We mention here some future directions:

We excluded an important question from our study, which has a well-known solution in the theory of Mazurkiewicz traces. In our setting, the problem reads as follows: For $k \in \mathbb{N}$ and an MVPA $\mathcal{A}$, when can we decide whether $[L_k(\mathcal{A})]_{\sim_{\widetilde{\Sigma}}}$ is the language of some MVPA and, hence, of some CVPA? If $\Sigma = \Sigma^{int}$, we know that this is the case iff $I_{\widetilde{\Sigma}} \cup \mathrm{id}_{\Sigma}$ is transitive [25]. In the general setting, the question remains open.

Given an MVPA $\mathcal{A}$, one may ask if $\mathcal{A}$ is indeed already a CVPA such that its local state spaces and transition relations can be computed effectively. Those questions are addressed and answered positively in [9, 19] for asynchronous automata.

In CVPA, processes communicate via shared memory. It will be interesting to study extensions of communicating finite-state machines (CFMs), where processes can send and receive messages through first-in first-out channels, by visibly pushdown stacks. While CVPA recognize sets of visibly pushdown traces, a *visibly pushdown CFM* would give rise to the notion of a *visibly pushdown message sequence chart*. Interestingly, there are theorems for CFMs that constitute counterparts of Zielonka's Theorem [13, 15]. We are therefore confident that the study of visibly pushdown CFMs will be fruitful.

For both Mazurkiewicz traces [10] and nested words [1], temporal logics have been studied. We raise the question if these logics can be combined towards expressive specification formalisms with decidable satisfiability and model-checking problems.

In a distributed setting, deadlock-free systems are particularly important. The paper [9] addresses the problem of synthesizing deadlock-free asynchronous automata from regular specifications. It remains to define a notion of deadlock-freeness for our setting and to study if the ideas from [9] can be adopted.

References

1. R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *LICS'07*, pages 151–160. IEEE Computer Society Press, 2007.
2. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC'04*, pages 202–211. ACM Press, 2004.
3. R. Alur and P. Madhusudan. Adding nesting structure to words. In *DLT'06*, volume 4036 of *LNCS*, pages 1–13. Springer, 2006.
4. B. Bollig. On the expressive power of 2-stack visibly pushdown automata. Research Report LSV-07-27, LSV, ENS Cachan, France, 2007.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *International Journal on Foundations of Computer Science*, 14(4):551–582, 2003.
6. J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.

7. I. Castellani, M. Mukund, and P.S. Thiagarajan. Synthesizing distributed transition systems from global specifications. In *FSTTCS'99*, volume 1739 of *LNCS*, pages 219–231. Springer, 1999.
8. S. Chaki, E.M. Clarke, N. Kidd, T.W. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS'06*, volume 3920, pages 334–349, 2006.
9. A. Ştefănescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *CONCUR'03*, volume 2761 of *LNCS*, pages 27–41. Springer, 2003.
10. V. Diekert and P. Gastin. LTL is expressively complete for Mazurkiewicz traces. In *ICALP'00*, volume 1853 of *LNCS*, pages 211–222. Springer, 2000.
11. V. Diekert and Y. Métivier. Partial commutation and traces. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, pages 457–534. Springer, 1997.
12. E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
13. B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Information and Computation*, 204(6):920–956, 2006.
14. B. Genest and A. Muscholl. Constructing exponential-size deterministic Zielonka automata. In *ICALP'06, Part II*, volume 4052 of *LNCS*, pages 565–576. Springer, 2006.
15. J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Information and Computation*, 202(1):1–38, 2005.
16. D. Kuske. Weighted asynchronous cellular automata. *Theoretical Computer Science*, 374(1-3):127–148, 2007.
17. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS'07*, pages 161–170. IEEE Computer Society Press, 2007.
18. S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS'08*, volume 4963 of *LNCS*, pages 299–314. Springer, 2008.
19. R. Morin. Decompositions of asynchronous systems. In *CONCUR'98*, volume 1466 of *LNCS*, pages 549–564. Springer, 1998.
20. A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In *MFCS'99*, volume 1672 of *LNCS*, pages 81–91. Springer, 1999.
21. E. Ochmański. Recognizable Trace Languages. In *The Book of Traces*, chapter 6, pages 167–204. World Scientific, 1995.
22. D. Peled, Th. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. *Theoretical Computer Science*, 195(2):183–203, 1998.
23. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'05*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
24. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416 – 430, 2000.
25. J. Sakarovitch. The "last" decision problem for rational trace languages. In *LATIN'92*, volume 583 of *LNCS*, pages 460–473. Springer, 1992.
26. K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

27. W. Thomas. On logical definability of trace languages. In *Proceedings of Algebraic and Syntactic Methods in Computer Science (ASMICS)*, Report TUM-I9002, Technical University of Munich, pages 172–182, 1990.
28. W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer, 1997.
29. W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. — Informatique Théorique et Applications*, 21:99–135, 1987.

Appendix (proof details for Theorem 21)

Theorem 21 *For every* CVPA $\mathcal{C}$ *over* $\widetilde{\Sigma}$*, there is a sentence* $\varphi \in \mathrm{MSO}(\widetilde{\Sigma})$ *such that* $\mathscr{L}(\varphi) = \mathscr{L}(\mathcal{C})$.

Proof. Let $\mathcal{C} = ((S_p)_{p\in Proc}, \Gamma, (\delta_a)_{a\in\Sigma}, \iota, F)$ be a CVPA over $\widetilde{\Sigma}$. Moreover, let $\mathscr{S} = \bigcup_{p\in Proc} S_p$. We suppose $\mathscr{S}$ and Γ to be disjoint. The sentence we are looking for is of the form

$$\begin{aligned}\varphi = \exists (X_s)_{s\in\mathscr{S}}\ \exists (X_A)_{A\in\Gamma}\ \big[\quad & \mathit{consistent}((X_s)_{s\in\mathscr{S}}, (X_A)_{A\in\Gamma}) \\ & \wedge\ \mathit{trans}((X_s)_{s\in\mathscr{S}}, (X_A)_{A\in\Gamma}) \\ & \wedge\ \mathit{final}((X_s)_{s\in\mathscr{S}})\ \big]\end{aligned}$$

Suppose $T = (E, \preceq, \mu, \lambda)$ is a trace. Observe that a run of CVPA on T can be represented as a pair of mappings *states* and *symbol* where *states* assigns to every event $i \in E$ an element from $\prod_{p\in proc(\lambda(i))} S_p$ (intuitively, the tuple is the collection of local states that are reached after executing i, whereas all processes not in $proc(\lambda(i))$ remain in their source local states), and *symbol* assigns to i a stack symbol from Γ. Actually, for every event $i \in E$, the formula *consistent* checks whether $(X_s)_{s\in\mathscr{S}}$ and $(X_A)_{A\in\Gamma}$ induce valid values for $states(i)$ and $symbol(i)$:

$$\begin{aligned}&\mathit{consistent}((X_s)_{s\in\mathscr{S}}, (X_A)_{A\in\Gamma}) = \\ &\forall x\Big[\ \bigwedge_{\substack{p\in Proc\\ a\in\Sigma_p}} \Big(\lambda(x) = a\ \rightarrow \bigvee_{s\in S_p} x \in X_s\Big) \\ &\quad\wedge \bigwedge_{\substack{p\in Proc\\ s\in S_p}} \Big(x \in X_s\ \rightarrow \bigvee_{a\in\Sigma_p} \lambda(x) = a\Big) \\ &\quad\wedge \bigwedge_{\substack{p\in Proc\\ s,s'\in S_p\\ s\neq s'}} \neg\Big(x \in X_s \wedge x \in X_{s'}\Big) \\ &\quad\wedge\ \mathit{partition}((X_A)_{A\in\Gamma}) \\ &\quad\wedge\ \lambda(x) \in \Sigma^c\ \rightarrow\ \neg(x \in X_\perp) \\ &\quad\wedge\ \mathit{unmatched\_pop}(x)\ \rightarrow\ x \in X_\perp \\ &\quad\wedge\ \forall y\Big((x,y) \in \mu\ \rightarrow \bigvee_{A\in\Gamma\setminus\{\perp\}} (x \in X_A \wedge y \in X_A)\Big)\ \Big]\end{aligned}$$

where the first-order formula $\mathit{partition}((X_A)_{A\in\Gamma})$ shall guarantee that $(X_A)_{A\in\Gamma}$ is a partition of the set of events (where X_A might be the empty set) and $\mathit{unmatched\_pop}(x) = \lambda(x) \in \Sigma^r \wedge \neg\exists y\ (y,x) \in \mu$. Moreover,

$\mathit{trans}((X_s)_{s\in\mathscr{S}}, (X_A)_{A\in\Gamma}) =$

$$\forall x \bigvee_{\substack{a\in\Sigma \\ (\overline{s},A,\overline{s}')\in\delta_a}} \Big[\lambda(x) = a \wedge x \in X_A \wedge \bigwedge_{p\in proc(a)} x \in X_{\overline{s}'_p} \wedge \bigvee_{\substack{P\subseteq proc(a) \\ \text{s.t. } \overline{s}_p=\iota_p\ \forall p\in P}} \Big(\bigwedge_{p\in P} \mathit{first}_p(x) \wedge \bigwedge_{p\in proc(a)\setminus P} \exists y\big(\mathit{prev}_p(y,x) \wedge y \in X_{\overline{s}_p}\big)\Big)\Big]$$

where $\overline{s}_p$, $\overline{s}'_p$, and ι_p refer to the p-components of $\overline{s}, \overline{s}', \iota \in S$, respectively. This formula ensures that, for every x, there is a transition that corresponds to the state and stack-symbol assignment. The set P will hereby contain all those processes p that have not moved yet and therefore are in the initial state ι_p. The subformula $\mathit{first}_p(x)$ holds true if x is the very first event that is executed by process p, i.e., $\mathit{first}_p(x) = \lambda(x) \in \Sigma_p \wedge \forall y\,(y \prec x \rightarrow \lambda(x) \notin \Sigma_p)$. Similarly, $\mathit{prev}_p(y,x) = \lambda(y) \in \Sigma_p \wedge \lambda(x) \in \Sigma_p \wedge y \prec x \wedge \forall z\,((\lambda(z) \in \Sigma_p \wedge y \prec z \preceq x) \rightarrow z = x)$ expresses that y is the last p-event that is executed before the p-event x. Then, y is actually the event where the source p-state of the transition taken at x is located. Finally,

$\mathit{final}((X_s)_{s\in S}) =$

$$\bigvee_{\substack{\overline{s}\in F \\ P\subseteq Proc \\ \text{s.t. } \overline{s}_p=\iota_p\ \forall p\in P}} \Big(\bigwedge_{\substack{p\in P \\ a\in\Sigma_p}} \big(\neg\exists x\,(\lambda(x) = a)\big) \wedge \bigwedge_{p\in Proc\setminus P} \exists x\big(\mathit{last}_p(x) \wedge x \in X_{\overline{s}_p}\big)\Big)$$

ensures that the global state that is reached after the execution of a trace is a final one. Hereby, we use $\mathit{last}_p(x) = \lambda(x) \in \Sigma_p \wedge \forall y\,(x \prec y \rightarrow \lambda(y) \notin \Sigma_p)$ to identify the event that has to provide the final p-state.

Indeed, we have $\mathscr{L}(\varphi) = \mathscr{L}(\mathcal{C})$. □