Technical Report 2009-001
Unrestricted and Disjoint Operations over Multi-Stack Visibly Pushdown Languages*

Stefan D. Bruda and Md Tawhid Bin Waez
Department of Computer Science
Bishop's University
Sherbrooke, Quebec J1M 1Z7, Canada
email: {bruda|mtbwaez}@cs.ubishops.ca

18 May 2009

Abstract

Visibly Pushdown Languages (VPL) have been proposed as a formalism useful for specifying and verifying complex, recursive systems such as application software. However, VPL turn out to be unsuitable for the compositional specification of concurrent software, as they are not closed under shuffle. Multi-stack Visibly Pushdown Languages (MVPL) express naturally concurrent constructions. We find however that concurrency cannot be expressed compositionally (for indeed MVPL are not closed under shuffle either). Furthermore, MVPL operations must be expressed under rigid restrictions on the input alphabet, that hinder between others the specification of dynamic creation of threads of execution. If we remove the restrictions, then MVPL loose almost all their closure properties; we find however a natural renaming process that yields the notion of disjoint MVPL operations. These operations eliminate the restrictions and also creates closure under shuffle. This effort opens the area of MVPL-based compositional specification and verification of complex systems.

Keywords: multi-stack visibly pushdown language shuffle, visibly pushdown languages, multi-stack visibly pushdown languages, closure properties, compositional specification and verification.

1 Introduction

Pushdown automata naturally model the control flow of sequential computation in typical programming languages with nested, and potentially recursive invocations of program modules such as procedures and methods. Many non-regular properties are therefore required for software verification. We need to specify and verify properties such as "if p holds when a module is invoked, the module must return, and q must hold upon return" [1]. Many non-regular properties generate

*This research was supported by the Natural Sciences and Engineering Research Council of Canada. Part of this research was also supported by Bishop's University.

an infinite state space, which cannot be handled by finite-state process algebrae or by standard verification techniques such as model checking. Context-free process algebrae such as basic process algebra (BPA) [3] can specify such context-free properties. Still, most of the software use many parallel components (such as multiple threads). In addition, many conformance-testing techniques (such as may/must testing [5]) use test cases that run in parallel with the process under test. Concurrency is therefore required for software verification, but cannot be provided by context-free process algebrae since context-free languages are not closed under intersection [8].

A first step toward the specification and verification of recursive, concurrent systems is the introduction of the class of visibly pushdown languages (VPL) [2] which lies between balanced languages and deterministic context-free languages. VPL have all the appealing theoretical properties that the regular languages enjoy: deterministic accepters of VPL are as expressive as their nondeterministic counterparts; the class is closed under union, intersection, complementation, concatenation, Kleene star, prefix, and language homomorphisms; membership, emptiness, language inclusion, and language equivalence are all decidable. VPL are accepted by visibly pushdown automata (vPDA) whose stack behaviour is determined by the input symbols. A vPDA operates on a word over an alphabet that is partitioned into three disjoint sets of call, return, and local symbols. Any input symbol can change the control state but call and return symbols can also change the stack content. While reading a call symbol the automaton must push one symbol on the stack and while reading a return symbol it must pop one symbol (unless the stack is already empty).

The closure of VPL under intersection suggests that these languages can express concurrency in an adequate, compositional manner. It turns out however that VPL are not closed under shuffle [9]. As a consequence attempts at specifying the behaviour of concurrent systems [4] found that such a specification is awkward at best and severely crippled at worst.

A recently proposed extension of VPL is the class of multi-stack visibly pushdown languages (MVPL) [11]. MVPL have most of the nice theoretical properties VPL enjoy. They also model concurrency in a natural manner, by requiring a separate stack (and a separate partition of call, return, and local symbols) for every thread of execution of a system. MVPL are accepted by multi-stack visibly pushdown automata (MvPDA). Each MvPDA featuring n stacks operates on an input alphabet that is partitioned into $n \times 2 + 1$ partitions: one set of call symbols and one set of return symbols for each stack, plus one global set of local symbols. The behaviour of an MvPDA is similar to the behaviour of a vPDA, with the obvious addition that a certain call (or return) symbol operates only on its designated stack.

This all being said, we show in this paper that MVPL have the same disadvantages as VPL when it comes to specifying compositionally complex concurrent systems. We find for one thing that MVPL are not closed under shuffle either. Therefore, while MVPL are suitable for specifying concurrent systems, they cannot do it compositionally: the ensemble of two systems specified as MvPDA and put to work together in parallel cannot necessarily be specified as a MvPDA. The compositionality of parallel composition aside, we also find that MVPL do not support the modelling of dynamic creation of threads. Indeed, closure under all the properties involving two MVPL exist under very strict restrictions on the partitions of the two MVPL: once the partitions do not coincide, the closure properties disappear.

These restrictions are crippling for many important applications, such as compositional approaches conformance testing of concurrent, recursive systems. In order to emphasize the significance of all of this consider the `fork(2)` system call (the standard way of creating processes in

UNIX), which duplicates the caller; the two initially identical copies then run concurrently, diverging in their behaviour. It turns out that such a behaviour cannot be expressed using the original, restricted MVPL operations.[11], which are not even defined for the two languages modelling the parent and the child process (and even if the operations are defined, we still lack the critical closure under shuffle). Using unrestricted operations on the other hand gets us out of the MVPL domain (for once the restrictions are eliminated MVPL ceases to be closed under almost any interesting operation).

Fortunately, we are able to define a natural and intuitive renaming process (natural in the sense that it matches well what happens in a real system). Such a renaming eliminates the need of restrictions on the partitions of two languages being composed to each other: such restrictions were needed in order to keep MVPL closed under most operations, but our renaming process keeps the closure properties of MVPL even when the restrictions are not in place. Our renaming process creates disjoint operations (union, concatenation, shuffle), such that MVPL is closed over all of them (including shuffle!). In all, introducing a renaming process that observes what happens in practice together with the associated disjoint operations creates the framework needed for compositional approaches to the specification and verification of application software.

The remainder of this paper is organized as follows: We present in the next section the necessary preliminaries. We then show in Section 3 that neither VPL nor MVPL are closed under shuffle or under hiding call or return symbols. Section 4 shows what happens if we allow two MVPL to be combined (using concatenation, union, etc.) even if their alphabet partitions do not coincide (we show that we loose closure under most interesting operations). We introduce in Section 5 the notion of disjoint operations (union, concatenation, shuffle), which is based on a natural renaming process and we show that MVPL become once more closed under the disjoint operations. We conclude in Section 6, where we mention the consequences of this work in the area of algebraic specification and verification of complex software systems.

2 Preliminaries

The shuffle of two languages L_1 and L_2 over an alphabet Σ is defined as $L_1 \,|||\, L_2 = \{w_1v_1w_2v_2\cdots w_mv_m : w_1w_2\cdots w_m \in L_1, v_1v_2\cdots v_m \in L_2 \text{ for all } w_i, v_i \in \Sigma^*\}$.

Given a work w over an alphabet Σ and a set $A \subseteq \Sigma$, the result of hiding A in w is a word $w \setminus A$ which is the word w with all the occurrences of symbols in A erased. Given a language L over an alphabet Σ and a set $A \subseteq \Sigma$, the result of hiding A in L is the set $L \setminus A = \{w \setminus A : w \in L\}$.

2.1 Visibly Pushdown Languages

A vPDA [2] is a tuple $M = (Q, Q_I, \widetilde{\Sigma}, \Gamma, \Delta, Q_F)$. Q is a finite set of states, $Q_I \subseteq Q$ is a set of initial states, $Q_F \subseteq Q$ is the set of final states, Γ is the (finite) stack alphabet that contains a special bottom-of-stack symbol $\perp$, and $\Delta \subseteq (Q \times \Gamma^*) \times \widetilde{\Sigma} \times (Q \times \Gamma^*)$ is the transition relation. $\widetilde{\Sigma} = \Sigma_l \uplus \Sigma_c \uplus \Sigma_r$ is a finite set of visibly pushdown input symbols where Σ_l is the set of local symbols, Σ_c is the set of call symbols, and Σ_r is the set of return symbols. Every tuple $((P,\gamma), a, (Q,\eta)) \in \Delta$ (also written $(P,\gamma) \xrightarrow{a} (Q,\eta)$) must have the following form: if $a \in \Sigma_l \cup \{\varepsilon\}$ then $\gamma = \eta = \varepsilon$, else if $a \in \Sigma_c$ then $\gamma = \varepsilon$ and $\eta = \mathfrak{a}$ (where $\mathfrak{a}$ is the stack symbol pushed for a), else if $a \in \Sigma_r$ then if $\gamma = \perp$ then $\gamma = \eta$ (hence vPDA allow unmatched return symbols) else $\gamma = \mathfrak{a}$ and $\eta = \varepsilon$ (where $\mathfrak{a}$ is the stack

symbol popped for a). Note that ε-transitions (that is, transitions that do not consume any input) are not permitted to modify the stack.

The notion of run, acceptance, and language accepted by a vPDA are defined as usual: A run of M on some word $w = a_1 a_2 \dots a_k$ is a sequence of configurations $(q_0, \gamma_0)(q_{01}, \gamma_0) \cdots (q_{0m_0}, \gamma_0)(q_1, \gamma_1)(q_{11}, \gamma_1) \cdots (q_{1m_1}, \gamma_1)(q_2, \gamma_2) \cdots (q_k, \gamma_k)(q_{k1}, \gamma_k) \cdots (q_{km_k}, \gamma_k)$ such that $\gamma_0 = \bot$, $q_0 \in Q_I$, $(q_{j-1i}, \varepsilon) \xrightarrow{\varepsilon} (q_{ji}, \varepsilon) \in \Delta$ for all $1 \leq i \leq k$, $1 \leq j \leq m_i$, and $(q_{i-1m_{i-1}}, \gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Delta$ for every $1 \leq i \leq k$ and for some prefixes γ'_{i-1} and γ'_i of γ_{i-1} and γ_i, respectively. Whenever $q_{km_k} \in Q_F$ the run is accepting; M accepts w iff there exists an accepting run of M on w. The VPL $L(M)$ accepted by M contains exactly all the words w accepted by M.

2.2 Multi-Stack Visibly Pushdown Languages

An n-stack call-return alphabet is a tuple $\widetilde{\Sigma_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1 \leq i \leq n}, \Sigma_l \rangle$ of pair-wise disjoint alphabets. Σ_c^i and Σ_r^i are the finite set of *call symbols of stack i* and the finite set of *return symbols of stack i*, respectively. Σ_l is the finite set of local symbols[1]. We use the following notations: $\Sigma_c = \sum_{i=1}^n \Sigma_c^i$, $\Sigma_r = \sum_{i=1}^n \Sigma_r^i$, $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_l$.

A multi-stack visibly pushdown automaton (MvPDA) [11] over the n-stack call-return alphabet $\widetilde{\Sigma_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1 \leq i \leq n}, \Sigma_l \rangle$ is then a natural extension of a vPDA. It is tuple $M = (Q, Q_I, \Gamma, \Delta, Q_F)$, with Q, Q_I, Q_F, and Γ identical to the ones used in the definition of a vPDA (Section 2.1). Every tuple $((P, \gamma), a, (Q, \eta)) \in \Delta$ (also written $(P, \gamma) \xrightarrow{a} (Q, \eta)$) must have the following form: if $a \in \Sigma_l \cup \{\varepsilon\}$ then $\gamma = \eta = \varepsilon$, else if $a \in \Sigma_c^i$ then $\gamma = \varepsilon$ and $\eta = \mathfrak{a}$ (where $\mathfrak{a}$ is the stack symbol pushed for a on the i-th stack), else if $a \in \Sigma_r^i$ then if $\gamma = \bot$ then $\gamma = \eta$ (hence vPDA allow unmatched return symbols) else $\gamma = \mathfrak{a}$ and $\eta = \varepsilon$ (where $\mathfrak{a}$ is the stack symbol popped for a on the i-th stack). Note again that ε-transitions (that is, transitions that do not consume any input) are not permitted to modify the stack. The original MvPDA construction does not allow ε-transitions; it is quite immediate that the introduction of such transitions does not alter the language accepted by an MvPDA, so we introduce them for the sake of consistency with the definition of vPDA.

A configuration of M is a tuple (q, γ), where $q \in Q$ and $\gamma = \langle \gamma^1, \dots, \gamma^n \rangle$ with $\gamma^l \in (\Gamma \setminus \{\bot\})^* \bot$ for all $1 \leq l \leq n$. For a word $w = a_1 a_2 \dots a_m \in \Sigma^*$ a run of an MvPDA over w is a sequence of configurations $(q_0, \gamma_0)(q_{01}, \gamma_0) \cdots (q_{0m_0}, \gamma_0)(q_1, \gamma_1)(q_{11}, \gamma_1) \cdots (q_{1m_1}, \gamma_1)(q_2, \gamma_2) \cdots (q_k, \gamma_k)(q_{k1}, \gamma_k) \cdots (q_{km_k}, \gamma_k)$ such that $\gamma_0^l = \bot$ for all $1 \leq l \leq n$, $q_0 \in Q_I$, $(q_{j-1i}, \varepsilon) \xrightarrow{\varepsilon} (q_{ji}, \varepsilon) \in \Delta$ for all $1 \leq i \leq k$, $1 \leq j \leq m_i$; whenever $a_i \in \Sigma_c^p \cup \Sigma_r^p$, $\gamma_{i-1}^l = \gamma_i^l$ for all $l \neq p$, $(q_{i-1m_{i-1}}, \gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Delta$ for every $1 \leq i \leq k$ and for some prefixes γ'_{i-1} and γ'_i of γ_{i-1}^p and γ_i^p, respectively; whenever $a_i \in \Sigma_l$, $(q_{i-1m_{i-1}}, \varepsilon) \xrightarrow{a_i} (q_i, \varepsilon) \in \Delta$ and $\gamma_{i-1} = \gamma_i$. Whenever $q_{km_k} \in Q_F$ the run is accepting; M accepts w iff there exists an accepting run of M on w. The MVPL $L(M)$ accepted by M contains exactly all the words w accepted by M.

Operations over VPL and MVPL (complement, union, etc.) are defined as usual, as set operations. However, such operations between two languages are originally defined [11] only when the alphabets of the two languages are identical. We will however relax this condition starting from Section 4.

[1] This set is called the set of internal actions originally [11]. However, this notation conflicts with the notion of internal action in a labelled transition system [7], so we revert to the original terminology used for VPL [2].

In the following we always denote by $\mathfrak{a}$ the symbol pushed on a stack by a call symbol a, setting in effect $\Gamma = \{\mathfrak{a} : a \in \Sigma_c\} \cup \{\bot\}$. This is done for presentation convenience only, we make no implicit assumption of how $\mathfrak{a}$ and a relate to each other (except that the former is pushed to the stack by the latter). Whenever a call symbol a pushes $\mathfrak{a}$ on the stack which is in turn popped off the stack by a return symbol b, we say that a and b are *matched*.

3 VPL and MVPL Are Not Closed under Shuffle or Hiding

That VPL is not closed under shuffle presents a major stumbling block for a compositional approach to concurrent, recursive systems. VPL are also not closed under hiding call or return symbols. Such a lack of closure has been communicated to us privately [9]; for completeness we include a proof here. We find that that these properties (and associated problems) extend immediately to MVPL.

Theorem 1 *Neither VPL nor MVPL are closed under shuffle or under hiding call or return symbols. Both VPL and MVPL are closed under hiding local symbols.*

Proof. Consider $L_1 = \{{c_1}^n {r_1}^n : n \geq 0\}$ and $L_2 = \{{c_2}^n {r_2}^n : n \geq 0\}$ and let $w = w_1 w_2$ with $w_1 = \{c_1^p c_2^q\}$ and $w_2 \in \{r_1, r_2\}^{p+q}$ for some $p, q \geq 0$. Note first that both r_1 and r_2 must match c_1 (and c_2) in the shuffle, as $c_1^p r_1^p c_2^q r_2^q \in L_1 \,|||\, L_2$ (and so r_1 must match c_1) and also $c_2^q c_1^p r_2^q r_1^p \in L_1 \,|||\, L_2$ (and so r_2 must match c_1; a similar construction shows that r_1 and r_2 must both match c_2). Also note that $w \in L_1 \,|||\, L_2$ iff $|w_2|_{r_1} = p$ and $|w_2|_{r_2} = q$.

Consider now a hypothetical vPDA that accepts $L_1 \,|||\, L_2$. Such a vPDA must discern between the forms of w in which $|w_2|_{r_1} = p$ and $|w_2|_{r_2} = q$ (that belong to the shuffle) and the other forms of w (that do not). The automaton must push on the stack exactly one symbol for each of the c_1 and c_2 symbols (for it needs to remember both p and q) and then recognize precisely p symbols r_1 and q symbols r_2. However, the automaton can remember neither p nor q in its finite state control (since both are arbitrarily large), and cannot differentiate between p and q on the stack (since both r_1 and r_2 match c_1 and c_2 equally well). Therefore no vPDA can distinguish between the valid and invalid forms of w, so no vPDA exists that can accept their shuffle.

The same kind of a language shows that MVPL are not closed under shuffle either. We note that the language $L_1 \,|||\, L_2$ can be easily accepted by an MvPDA with two stacks; however, MVPL come with well-defined partitions, so we can force the symbols c_1, c_2, r_1, and r_2 to belong to the same stack simply by choosing a suitable alphabet. When this happens, the same argument yields the impossibility of $L_1 \,|||\, L_2$ to be accepted by any MvPDA.

Consider now the language $\{(caaa)^n (rb)^n : n \geq 0\}$. The language is clearly VPL provided that c is a call symbol, r is a return symbol, and a and b are local symbols; however, hiding c yields the language $\{(aaa)^n (rb)^n : n \geq 0\}$, which cannot be VPL under any partition. Indeed, we must push on the stack one symbol for each of the a symbols, for otherwise we have no means of remembering n; once we decide that some a symbols must push something on the stack, then all the a symbols must push (given the nature of VPL). However, the stack height becomes then $3n$, which cannot be compared by any vPDA with n or $2n$ (the number of return symbols, depending on whether we consider r a return symbol, b a return symbol, or both r and b return symbols). Hiding r instead of c, or hiding both c and r yield languages with similar structure, (that cannot be VPL). The same construction shows the lack of closure under hiding call or return symbols for

MVPL; we can just pick one stack and build a language like the one above which uses only that one stack.

Closure under hiding local symbols is immediate for both VPL and MVPL (since ε-transitions that do not modify the stack are permitted). □

4 Unrestricted Operations over MVPL

The usual operations over two MVPL (union, concatenation, etc.) are defined [11] only when the two languages are over exactly the same n-stack call-return alphabet. For considerations related to dynamic creation of threads of execution in concurrent systems it would be preferable to replace such a strong restriction with the restriction that two languages can be composed iff the sets of call, local, and return symbols of the two languages do not overlap (meaning that a call symbol in one language is not a return or a local symbol in the other, and so on). By contrast with the original definition, we call an operation that imposes this kind of weaker restriction *unrestricted*.

Definition 1 RESTRICTED AND UNRESTRICTED MVPL OPERATIONS: *Let L' and L'' be any two MVPL over two alphabets $\widetilde{\Sigma'_{n'}}$ and $\widetilde{\Sigma''_{n''}}$, respectively. We define two variants of any operation @ between L' and L'', @ $\in \{\cup, \cap, \circ, |||\}$. Both variants are defined as usual set operations, but the applicability, as well as the alphabet of the resulting composite language are different.*

The restricted *@ is defined only when $\widetilde{\Sigma'_{n'}} = \widetilde{\Sigma''_{n''}}$. The result in a language over $\widetilde{\Sigma'_{n'}}$ (or $\widetilde{\Sigma''_{n''}}$). By contrast, the* unrestricted *@ is defined between any two L' and L'' with the only restriction that the sets $(\Sigma'_l \cup \Sigma''_l)$, $(\bigcup_{1\le i\le n'} \Sigma'^i_c) \cup (\bigcup_{1\le i\le n''} \Sigma''^i_c)$, and $(\bigcup_{1\le i\le n'} \Sigma'^i_c) \cup (\bigcup_{1\le i\le n''} \Sigma''^i_c)$ are pairwise disjoint (meaning that the sets of all local symbols, all call symbols, and all return symbols are pairwise disjoint). The alphabet $\widetilde{\Sigma_x}$ of L'@L'' for some $x > 0$ is constructed as follows:*

- *The set of local symbols in $\widetilde{\Sigma_x}$ is $\Sigma_l = \Sigma'_l \cup \Sigma''_l$.*
- *We take first all the call-return pairs from both alphabets $\widetilde{\Sigma'_{n'}}$ and $\widetilde{\Sigma''_{n''}}$ and we put them in $\widetilde{\Sigma_x}$. We obtain an $n' + n''$-stack call-return alphabet, i.e., $x = n' + n''$ (which is not necessarily valid).*
- *We collapse the resulting alphabet as follows: For any (Σ^p_c, Σ^p_r) and (Σ^q_c, Σ^q_r) from $\widetilde{\Sigma_x}$ such that $\Sigma^p_c \cap \Sigma^q_c \neq \emptyset$ or $\Sigma^p_r \cap \Sigma^q_r \neq \emptyset$, we eliminate (Σ^p_c, Σ^p_r) and (Σ^p_c, Σ^p_r) from $\widetilde{\Sigma_x}$ and we introduce in $\widetilde{\Sigma_x}$ instead $(\Sigma^p_c \cup \Sigma^q_c, \Sigma^p_r \cup \Sigma^q_r)$. We keep collapsing $\widetilde{\Sigma_x}$ for as long as possible (this making it a valid x-stack call-return alphabet for some $x \le n' + n''$).*

It is easy to see that this is indeed a valid operation, specifically, that $\widetilde{\Sigma_x}$ is an x-stack call-return alphabet. The alphabet is constructed naturally, in the sense that whether a call symbol or a return symbol appear on two different stacks in the composite language, then the two stacks collapse into one. We also note that a restricted MVPL operation is a particular case of the unrestricted variant of that operation; indeed, if the two alphabets of the two languages are the same, the collapsing process from Definition 1 yields the same alphabet as the original one (save for a possible renumbering of the pairs of call and return symbols; however, the numbering of these pairs is done solely for presentation convenience and does not affect language definitions).

Unfortunately, allowing unrestricted operations does not preserve the nice closure properties of MVPL. The following simple languages will help us show this.

Lemma 2 *$L_{p<q} = \{{c_1}^n {c_2}^p {r_2}^q {r_1}^n : p < q\}$, $L_{p>q} = \{{c_1}^n {c_2}^p {r_2}^q {r_1}^n : p > q\}$, $L_{p\neq q} = \{{c_1}^n {c_2}^p {r_2}^q {r_1}^n : p \neq q\}$ are context-free but are not VPL.*

Proof. We present a proof for $L_{p<q}$. The proofs for the other two languages are trivial variations.

$L_{p<q}$ is clearly context-free, being generated by the grammar $S \rightarrow c_1 S r_1$, $S \rightarrow B$, $B \rightarrow c_2 B r_2$, $B \rightarrow r_2 C$, $C \rightarrow r_2 C$, $C \rightarrow \varepsilon$.

Assume now that a vPDA M that accepts $L_{p<q}$ exists. Given that the number of c_1 symbols is in direct relation with the number of r_1 symbols in the input, and that n can exceed the number of states in M, one of the symbols c_1 and r_1 must be a call symbol and the other must be a return symbol (as the stack is the only thing that can keep a count of the number of occurrences of c_1 and then r_1). Since c_1 precedes r_1, the call symbol must be c_1 (and the return symbol r_1). For the same reasons c_2 must be a call symbol and r_2 must be a return symbol.

After the inspection of all the c_1 and c_2 in the input, the stack will contain $n + p + 1$ stack symbols (including $\perp$). After q occurrence of r_2 there will be $n - (q - p) + 1$ stack symbols on the stack. There is no way however for M to remember the number $q - p$ (indeed, this number can grow arbitrarily above the number of states of M), so there is no way M can determine the value of n out of the $n - (q - p) + 1$ stack symbols. M cannot therefore accept exactly n occurrence of r_1. $L_{p<q}$ is therefore not a VPL. □

Theorem 3 *MVPL are closed under prefix, suffix, Kleene closure, and complement but are not closed under unrestricted union, concatenation, intersection, and shuffle.*

Proof. We note first that the property of being (or not) unrestricted does not apply to prefix, suffix, Kleene closure, and complement (since only one language is involved in these operations). Closure under complement remains valid then as per previous results [11]. Closure under prefix and postfix can be established by standard techniques, namely by making all the states in the initial automaton initial and final, respectively (this will suffice since an MvPDA accepts by final state). Finally closure under Kleene closure is easily established by techniques similar to the ones used for finite automata [8] (again, MvPDA accept by final state; we also note that the Kleene closure of balanced words is balanced, and any lack of balance is kept intact by a Kleene closure operation).

$L_{p<q}$ is trivially accepted by an MvPDA with two stacks (one for c_1 and r_1 and another for c_2 and r_2). Similarly, the language $L_1 = \{{c_1}^n {r_2}^n : n \geq 0\}$ is trivially accepted by an MvPDA with one stack. Assume that $L = L_{p<q} \cup L_1$ is accepted by an MvPDA M. L_1 forces c_1 to be a call symbol on the same stack as the return symbol r_2. $L_{p<q}$ on the other hand forces c_1 to be a call symbol on the same stack as the return symbol r_1 and c_2 to be a call symbol on the same stack as the return symbol r_2. This forces M to behave as a one stack MvPDA, or a vPDA. We know however (by Lemma 2) that no vPDA can accept $L_{p<q}$ so there is no MvPDA that can accept L (since no vPDA can accept exactly n symbols following $c_1^n c_2^p r_2^q$ with $p \neq q$).

Let M' be now an MvPDA that accepts $L' = L_1 L_{p<q}$. For the same reason as in the above case M must be a one stack MvPDA, or again a vPDA. L' also inherits the problem of $L_{p<q}$ (as detailed above), so M' cannot exist.

Let, M'' be the MvPDA that accepts the shuffle between $L_{p<q}$ and L_1. M'' is again forced to be a one-stack MvPDA (or a vPDA). We then have the same problem as the one we found earlier in Theorem 1 (no vPDA can accept exactly p symbols r_1 and q symbols r_2 following $p + q$ call symbols that are indistinguishable from each other on the stack), and so M'' cannot exist. We note

in passing that in the case of shuffle not even the original definition (or restricted shuffle) provides any closure property.

MVPL being closed under unrestricted intersection imply that they are also closed under unrestricted union (since they are already closed under complementation). Since closure under unrestricted union does not hold, MVPL are not closed under unrestricted intersection either. □

5 Disjoint Operations over MVPL

The restrictions imposed originally over the MVPL operations [11] have the advantage that they yield good closure properties (except under shuffle), but they are somehow artificial. Indeed, in a real concurrent system it is very common that the same function is run by two threads of execution that run in parallel, and also that the said same function starts identically but then behaves differently in the two threads. In all, it is rather common that the partitions of the alphabet are not identical between the two threads. This being said, we note that in practice the two threads that run in parallel operate on their own stack; there is never any overlap between the stack of one partition and the stack of the other. We will attempt to capture such a behaviour of real systems via a third kind of operations, namely *disjoint* operations.

Indeed, we can eliminate the requirement that two MVPL can be combined together only if their alphabets are identical. Instead, we rename (some of) the stacks of one of the languages under consideration so that the interferences used in the proof of Theorem 3 no longer happen.

Definition 2 STACK RENAMING: *Let L be an MVPL over the n-stack call-return alphabet $\widetilde{\Sigma_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\leq i\leq n}, \Sigma_l\rangle$. The p-stack renaming $\mathcal{R}_p(L)$ of L is an MVPL over the n-stack call-return alphabet $\widetilde{\Sigma'_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\leq i\leq n, i\neq p}, (\Sigma_c^{n+1}, \Sigma_r^{n+1}), \Sigma_l\rangle$ such that there exists a bijection $f : \Sigma_c^p \cup \Sigma_r^p \longrightarrow \Sigma_c^{n+1} \cup \Sigma_r^{n+1}$ with $f(x) \in \Sigma_c^{n+1}$ iff $x \in \Sigma_c^p$ and $f(x) \in \Sigma_r^{n+1}$ iff $x \in \Sigma_r^p$. Specifically, $\mathcal{R}_p(L) = \{r(w) : w \in L\}$, where $r : \Sigma \longrightarrow \Sigma'$ is the function $r(x) = x$ for any $x \in \Sigma \setminus (\Sigma_c^p \cup \Sigma_r^p)$ and $r(x) = f(x)$ otherwise, extended as usual to strings by $r(a_1a_2\ldots a_l) = r(a_1)r(a_2)\ldots r(a_l)$. By abuse of notation $\mathcal{R}_{p_1,p_2,\ldots,p_k}(L) = \mathcal{R}_{p_1}(\mathcal{R}_{p_2}(\ldots\mathcal{R}_{p_k}(L)\ldots))$. Further abusing the terminology we will also use the term stack renaming (or just renaming when there is no ambiguity) for this (composite) renaming.*

Given that a stack renaming gets rid of a stack only to replace it with another stack that operates identically to the original, the following is immediate:

Theorem 4 *Some stack renaming $\mathcal{R}_{p_1,p_2,\ldots,p_k}(L)$ of a language L is MVPL iff L is an MVPL.*

In other words, symbols associated with one stack in a given language can be renamed to the symbols associated with another stack with the following restrictions: if we rename one symbol then we also rename all the other symbols associated with the same stack, no symbol associated with the new stack will be in the language before renaming, and no symbol associated with the old stack will be in the language after renaming. The new stack is always new, meaning that the before-renaming MVPL is not using any symbol from that stack.

Using renaming judiciously, we get back the lost closure properties (and on top of it we also get closure under shuffle), no matter whether the alphabets of the two languages are identically partitioned or not.

Theorem 5 *Given two MVPL languages L' over $\widetilde{\Sigma'_{n'}}$ and L'' over $\widetilde{\Sigma''_{n''}}$ that can be combined using unrestricted MVPL operations, there exists a renaming $\mathcal{R}_{1,\ldots,n'}$ such that $\mathcal{R}_{1,\ldots,n'}(L') \cup L''$, $\mathcal{R}_{1,\ldots,n'}(L') \circ L''$, and $\mathcal{R}_{1,\ldots,n'}(L') \,|||\, L''$ are MVPL over an alphabet $\widetilde{\Sigma_{n'+n''}}$.*

Proof. Clearly, a renaming that moves all the stack partitions of L' so that they become completely different from the stacks of L'' exists. We take such a renaming as our $\mathcal{R}_{1,\ldots,n'}$.

Let $M' = (Q', Q'_I, \Gamma', \Delta', Q'_F)$ be the MvPDA that accepts $\mathcal{R}_{1,\ldots,n'}(L')$ and $M'' = (Q'', Q''_I, \Gamma'', \Delta'', Q''_F)$ be the MvPDA that accepts L''.

$\mathcal{R}_{1,\ldots,n'}$ guarantees that there will be no stack manipulation that is common between M' and M''. That $\mathcal{R}_{1,\ldots,n'}(L') \cup L''$ is an MVPL is then immediate: indeed, the MvPDA that accepts the union consists in the union of M' and M'' (meaning that we take the disjoint union of states and transitions, the union of initial, and the union of the final states of M' and M''). There is no stack interference between the transitions coming from M' and the transitions coming from M'' (since the two operate on completely different sets of stacks) and so the correctness of the construction follows from the construction that establishes the closure under union of regular languages [8].

For $\mathcal{R}_{1,\ldots,n'}(L') \circ L''$ we take the initial states of M' as being initial states, we take the final states if M'' as being the final states, we join by ε-transitions all the final states of M' with all the initial states of M'' and then we take the union of the transitions of M' and M''. Again there is no stack interference, and so the correctness of the construction follows from the corresponding construction for finite automata.

On to $\mathcal{R}_{1,\ldots,n'}(L') \,|||\, L''$ now: The MvPDA $M = (Q, Q_I, \Gamma, \Delta, Q_F)$ that accepts this language must simulate M' and M'' in the following manner: M must keep track of the state of both M' and M'', so we put $Q = Q' \times Q''$ (and therefore $Q_I = Q'_I \times Q''_I$). Whenever M' makes a move M'' must stay put, and the other way around. So for any $p', q' \in Q'$, $q'' \in Q''$, and $(p', \gamma) \xrightarrow{a'} (q', \eta) \in \Delta'$ we add the following transition to Δ: $((p', q''), \gamma) \xrightarrow{a'} ((q', q''), \eta)$. Conversely, for any $q' \in Q'$, $p'', q'' \in Q''$, and $(p'', \gamma) \xrightarrow{a''} (q'', \eta) \in \Delta''$ we add the following transition to Δ: $((q', p''), \gamma) \xrightarrow{a''} ((q', q''), \eta)$. Clearly, this transition relation is able to perform any number of steps of M' (or M'') while M'' (or M') stays in the same state, so the shuffle proceeds happily to the end of the input. At this point both the MvPDA must accept their portion of the input, so we have $Q_F = Q'_F \times Q''_F$. Once more there is no stack interference since the M' and M'' components of M operate on completely different sets of stacks. □

The renaming process outlined above motivates (together with the practical considerations mentioned at the beginning of this section) the following definition of operations over MVPL.

Definition 3 DISJOINT OPERATIONS OVER MVPL: *The* disjoint union *[*concatenation, shuffle*] of two MVPL L' and L'' that can be combined using unrestricted MVPL operations is the language $\mathcal{R}_{1,\ldots,n'}(L') \cup L''$ (unrestricted union) [$\mathcal{R}_{1,\ldots,n'}(L') \circ L''$ (unrestricted concatenation), $\mathcal{R}_{1,\ldots,n'}(L') \,|||\, L''$ (unrestricted shuffle)], where $\mathcal{R}_{1,\ldots,n'}$ renames the alphabet of L' in such a way that no stack alphabet is common between $\mathcal{R}_{1,\ldots,n'}(L')$ and L''.*

Given Theorem 5, the following is immediate.

Corollary 6 *MVPL are closed under disjoint union, concatenation, and shuffle.*

The disjoint union, concatenation, and shuffle are similar to their restricted or unrestricted counterparts. Indeed, the renamings that are used to define these operators only play around with stack names (or more accurately shift around the stacks to eliminate possible conflicts), so the following is immediate.

Theorem 7 *Disjoint union, concatenation, and shuffle are commutative and associative up to a stack renaming.*

6 Conclusions

VPL capture the properties of systems with recursive modules. Unfortunately the lack of closure under shuffle effectively prevents VPL-based compositional approaches to the specification of concurrent systems. MVPL on the other hand appear to model concurrent systems in a natural way. Unfortunately, it turns out that MVPL lack as well closure under shuffle, so they end up having the same limitations as VPL (Theorem 1).

Consider further the standard way of creating multiple processes in UNIX mentioned at the beginning of this paper, namely the `fork(2)` system call. Recall that such a call starts by duplicating the existing process; the two initially identical copies then run concurrently, often diverging in their behaviour as time goes by. Such a divergence cannot be specified using restricted operations. Indeed, consider the shuffle between the languages L_1 (the traces of a parent process) and L_2 (the traces of a child process, starting the same as the parent but then diverging in its behaviour) from the proof of Theorem 1. The interleaved run of these two processes, that is, the shuffle between these two languages is not an MVPL—note indeed that c_1 and c_2 are likely in this case to belong to the same stack (since the child process is created by its parent, which has one stack to begin with) whereas r_1 and r_2 will ideally belong to different stacks (for indeed the two, diverging behaviours happen on two different stacks and we assume that one needs to separate these behaviours). Given these considerations, the unrestricted shuffle between these two languages does not lead to an MVPL, whereas the restricted variant is not even defined. That is, the way the MVPL operations are defined originally (only on languages that agree completely on their alphabets) encumber the modelling of such a behaviour. Under relaxed (and more realistic) conditions however MVPL cease to be closed to almost every interesting operation, not just shuffle (Theorem 3); whenever a certain partition causes a collapse of stacks in the composition of two languages, the collapse can be manipulated so that the composite language ceases to be an MVPL.

Based on the intuition of the the two processes created by `fork(2)` behaving identically to start with and then runing independently from each other (with different stacks), we then introduce a natural stack renaming process that not only observes what happens in real life, but also gives back all the closure properties of MVPL, with closure under shuffle added on top for good measure (Theorem 5). Indeed, based on this renaming process one can easily define disjoint variants of all the interesting operators such that MVPL are closed under them (Corollary 6).

Our results open the possibility of using MVPL in the process of specifying and verifying recursive, concurrent systems. Indeed, the closure properties established here show that such systems can be specified compositionally, so that algebraic approaches to specification and verification can be created with relative ease. Our future work intends to pursue such algebraic approaches. Some preliminary considerations on the matter (related to the concepts introduced in this paper) follow. These considerations refer to a future MVPL-based algebraic specification mechanism for

concurrent, recursive systems, following the same path as our previous, VPL-based attempt [4].

We note first that we now have three kinds of operators: the original [11], restricted ones, the unrestricted variants, and the disjoint ones. As we argued above, it makes practical sense to use disjoint shuffle over the other variants. Using disjoint shuffle makes theoretical sense as well, since this is the only variant under which MVPL are closed. Disjoint shuffle also illustrates quite eloquently the power of MVPL over VPL, for indeed such an operation cannot even be defined over VPL. At the other side of the spectrum, it makes no theoretical sense (and for that matter it is likely that it makes no practical sense either) to use unrestricted operations, for indeed MVPL is not closed under any of them. Things are less clear when it comes with the restricted versus disjoint variants of union and concatenation. We argue that it makes sense to use disjoint concatenation. Indeed, concatenation models two independent processes that follow one after the other, so it makes sense to give separate sets of stacks to the two processes. We would argue that it makes sense to use restricted union, for indeed union models a process with two, diverging behaviours (that nonetheless use the same set of stacks). For the sake of consistency though, we note that using disjoint union does not hurt, since the behaviour of the two processes diverge irreversibly anyway, so giving them separate sets of stacks does not have any consequence, however unrealistic this is; therefore we believe that disjoint union is the appropriate operation.

How about intersection? This operation becomes meaningless under the disjoint construction. Intersection can be used to model two portion of processes that synchronize with each other over all their actions. In the disjoint setting however processes can synchronize over local symbols only, which is easily specified at the MvPDA level in the same manner as for the vPDA-based algebraic specifications [4], using an intersection-like construct. Other than this, there is no need for intersection, so it can be safely ignored.

Note finally that the only interesting closure property that does not hold for MVPL is closure under hiding. Hiding is used in process algebrae based of regular languages such as CSP [6, 10] for encapsulation purposes, so that two processes synchronize on a well-defined common interface instead of synchronizing on any action that might happen to be common to both. We note the lack of closure under hiding for MVPL, but we also note that this is applicable only to call and return symbols. Indeed, we can freely hide local symbols, but then the local symbols are (and practically speaking should be!) the only candidates for synchronization (as the call and return symbols of two processes running in parallel will never be shared, given the necessary renaming as per Theorem 5 and Corollary 6). In other words, just hiding local symbols serves nicely all the encapsulation purposes that hiding serves in CSP (and in general in process algebrae based on regular languages). In passing we note that call and return symbols can actually be hidden (together with the local symbols that fall in between) under some circumstances, namely by using some "abstract" operation [1, 4], which is useful for specifying local properties of a recursive module.

References

[1] R. ALUR, K. ETESSAMI, AND P. MADHUSUDAN, *A temporal logic of nested calls and returns*, in Proceedings of the 10th International Conference on Tools and Algorithms for the construction and Analysis of Systems (TACAS 04), vol. 2988 of Lecture Notes in Computer Science, Springer, 2004, pp. 467–481.

[2] R. ALUR AND P. MADHUSUDAN, *Visibly pushdown languages*, in Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 04), ACM Press, 2004, pp. 202–211.

[3] J. A. BERGSTRA AND J. W. KLOP, *Process theory based on bisimulation semantics*, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, J. W. de Bakker, W. de Roever, and G. Rozenberg, eds., vol. 354 of Lecture Notes in Computer Science, Springer, 1988, pp. 50–122.

[4] S. D. BRUDA AND M. T. BIN WAEZ, *Communicating visibly pushdown processes*, in The 17th International Conference on Control Systems and Computer Science (to appear), Bucharest, Romania, May 2009.

[5] R. DE NICOLA AND M. HENNESSY, *Testing equivalences for processes*, Theoretical Computer Science, 34 (1983), pp. 83–133.

[6] A. R. HOARE, *Communicating Sequential Processes*, Prentice-Hall, 1988.

[7] J.-P. KATOEN, *Labelled transition systems*, in Model-Based Testing of Reactive Systems: Advanced Lectures, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, eds., vol. 3472 of Lecture Notes in Computer Science, Springer, 2005, pp. 615–616.

[8] H. R. LEWIS AND C. H. PAPADIMITRIOU, *Elements of the Theory of Computation*, Prentice-Hall, 2nd ed., 1998.

[9] P. MADHUSUDAN, *Private communication*.

[10] S. SCHNEIDER, *Concurrent and Real-time Systems: The CSP Approach*, John Wiley & Sons, 2000.

[11] S. L. TORRE, P. MADHUSUDAN, AND G. PARLATO, *A robust class of context-sensitive languages*, in LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, 2007, IEEE Computer Society, pp. 161–170.