UNRESTRICTED AND DISJOINT OPERATIONS OVER MULTI-STACK VISIBLY PUSHDOWN LANGUAGES

Stefan D. Bruda
Department of Computer Science, Bishop's University, Sherbrooke, Quebec, Canada

Tawhid Bin Waez
School of Computing, Queen's University, Kingston, Ontario, Canada

Keywords: Visibly pushdown languages, Multi-stack visibly pushdown languages, Closure properties, Shuffle, Formal methods, Compositional specification and verification.

Abstract: Visibly pushdown languages (VPL) were proposed as a formalism for specifying and verifying complex, recursive systems such as application software. However, VPL are unsuitable for compositional specification, as they are not closed under shuffle. Multi-stack visibly pushdown languages (MVPL) express naturally concurrent constructions. Concurrency cannot be expressed compositionally however, for MVPL are not closed under shuffle either. MVPL operations also restrict rigidly the input alphabets, which hinders the specification of dynamic threads; if we remove these restrictions MVPL loose almost all their closure properties. We find however a natural renaming process that yields disjoint MVPL operations. These operations eliminate the mentioned restrictions and creates closure under shuffle. This effort opens the area of MVPL-based compositional specification and verification.

1 INTRODUCTION

Typical programming languages (and so the control flow of application software) feature recursive invocations of modules (such as procedures or functions). This is modelled naturally by pushdown automata, so context-free (i.e., non-regular) properties are necessary in order to verify software (Alur et al., 2004). Such context-free properties generate an infinite state space, that cannot be handled by standard process algebrae or by verification techniques such as model checking; we have to turn to context-free process algebrae (Bergstra and Klop, 1988). Still, concurrency is needed for handling software: most software use parallel threads of execution, and some conformance-testing techniques (e.g., model-based testing) use tests that run in parallel with processes. Therefore context-free process algebrae cannot be used, for indeed context-free languages are not closed under intersection (Lewis and Papadimitriou, 1998).

A first step in the specification and verification of recursive, concurrent systems is the introduction of visibly pushdown languages (VPL) (Alur and Madhusudan, 2004), accepted by by visibly pushdown automata (vPDA), which have apparently all the appealing theoretical properties that the regular languages enjoy. However VPL are not closed under shuffle (Madhusudan, 2008), so attempts at specifying the behaviour of concurrent systems are awkward at best and crippled at worst (Bruda and Bin Waez, 2009).

A recently proposed extension of VPL are multi-stack visibly pushdown languages (MVPL) (La Torre et al., 2007). MVPL have most of the nice theoretical properties VPL enjoy, and model concurrency in a natural manner. MVPL are accepted by multi-stack visibly pushdown automata (MvPDA). Still, we show that MVPL have the same disadvantages when it comes to compositional specifications. We find for one thing that MVPL are not closed under shuffle either so while they are suitable for specifying concurrent systems, they cannot do it compositionally. Compositionality aside, we also find that MVPL do not support the modelling of dynamic creation of threads. Indeed, closure under all the operations involving two MVPL exist under very strict restrictions on the partitions of the two MVPL; once these restrictions are eliminated the closure properties disappear.

These restrictions are crippling for many impor-

D. Bruda S. and Bin Waez T..
UNRESTRICTED AND DISJOINT OPERATIONS OVER MULTI-STACK VISIBLY PUSHDOWN LANGUAGES.
DOI: 10.5220/0003494101560161
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 156-161
ISBN: 978-989-8425-77-5

tant applications, such as compositional approaches to 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 as the computation progresses. Such a situation cannot be expressed using the original, restricted MVPL operations, which are not even defined for the two languages modelling the parent and the child process (and even then we still lack the critical closure under shuffle). Using unrestricted operations on the other hand gets us out of the MVPL domain. Indeed, once the restrictions are eliminated MVPL ceases to be closed under almost any interesting operation, as mentioned earlier.

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 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!). 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.

We are aware of one other attempt at the same problem, which considers vPDA with two stacks (2-VPA and 2-OVPA) (Carotenuto et al., 2007). While 2-OVPA offer the same appealing closure properties, our constructions are more intuitive and more natural. In particular each process keeps its stack in our approach, while stacks are amalgamated in 2-OVPA. We also model inter-process communication via synchronized (local) symbols, while 2-OVPA use their stacks for this purpose; our approach matches better practical inter-process communication and avoids problems caused by hiding (as mentioned in Section 6).

2 PRELIMINARIES

The shuffle of L_1 and L_2 over Σ is $L_1 \,|||\, L_2 = \{w_1v_1w_2v_2\cdots w_mv_m : w_1w_2\cdots w_m \in L_1, v_1v_2\cdots v_m \in L_2 \text{ for all } w_i, v_i \in \Sigma^*\}$. Given $w \in \Sigma^*$ and $A \subseteq \Sigma$, the result of hiding A in w is $w \setminus A$, the word w with all the occurrences of symbols in A erased. The result of hiding A in the language L is $L \setminus A = \{w \setminus A : w \in L\}$.

A vPDA (Alur and Madhusudan, 2004) is a tuple $M = (Q, Q_I, \widetilde{\Sigma}, \Gamma, \Delta, Q_F)$. Q is a finite set of states, $Q_I \subseteq Q$ are initial states, $Q_F \subseteq Q$ are 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 calls, and Σ_r is the set of returns. Every $((P,\gamma), a, (Q,\eta)) \in \Delta$ (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 returns) 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.

A run of M on some word $w = a_1a_2\ldots a_k$ is a sequence of configurations $(q_0,\gamma_0)(q_{01},\gamma_0)\cdots(q_{0m_0},\gamma_0)(q_1,\gamma_1)(q_{11},\gamma_1)\cdots(q_{1m_1},\gamma_1)(q_2,\gamma_2)\cdots(q_k,\gamma_k)(q_{k1},\gamma_k)\cdots(q_{km_k},\gamma_k)$ with $\gamma_0 = \perp$, $q_0 \in Q_I$, $(q_{j-1i},\varepsilon) \xrightarrow{\varepsilon} (q_{ji},\varepsilon) \in \Delta$, and $(q_{i-1m_{i-1}},\gamma'_{i-1}) \xrightarrow{a_i} (q_i,\gamma'_i) \in \Delta$ for γ'_{i-1} and γ'_i prefixes of γ_{i-1} and γ_i, respectively. The run is accepting iff $q_{km_k} \in Q_F$; M accepts w iff there exists an accepting run of M on w. The VPL $\mathcal{L}(M)$ contains exactly all the words accepted by M.

An n-stack call-return alphabet is a tuple $\widetilde{\Sigma}_n = \langle(\Sigma_c^i, \Sigma_r^i)_{1\le i\le n}, \Sigma_l\rangle$ of pair-wise disjoint alphabets. Σ_c^i and Σ_r^i are the finite set of *calls of stack i* and the finite set of *returns of stack i*, respectively. Σ_l is the finite set of local symbols. 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) (La Torre et al., 2007) over $\widetilde{\Sigma}_n = \langle(\Sigma_c^i, \Sigma_r^i)_{1\le i\le n}, \Sigma_l\rangle$ is then a natural extension of a vPDA. It is a tuple $M = (Q, Q_I, \Gamma, \Delta, Q_F)$, with Q, Q_I, Q_F, and Γ as in the definition of a vPDA. 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 $\gamma = \perp$ then $\gamma = \eta$ else $\gamma = \mathfrak{a}$ and $\eta = \varepsilon$. Again 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 (q,γ), where $q \in Q$ and $\gamma = \langle\gamma^1,\ldots,\gamma^n\rangle$ with $\gamma^l \in (\Gamma \setminus \{\perp\})^*\perp$. A run of M

over $w = a_1 a_2 \dots a_m \in \Sigma^*$ 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 $\gamma'_0 = \perp$, $q_0 \in Q_I$, $(q_{j-1i},\varepsilon) \xrightarrow{\varepsilon} (q_{ji},\varepsilon) \in \Delta$; whenever $a_i \in \Sigma_c^p \cup \Sigma_r^p$, $\gamma'_{i-1} = \gamma'_i$ for all $l \neq p$, $(q_{i-1m_{i-1}},\gamma'_{i-1}) \xrightarrow{a_i} (q_i,\gamma'_i) \in \Delta$ for γ'_{i-1} and γ'_i prefixes of γ^p_{i-1} and γ^p_i, respectively; whenever $a_i \in \Sigma_l$, $(q_{i-1m_{i-1}},\varepsilon) \xrightarrow{a_i} (q_i,\varepsilon) \in \Delta$ and $\gamma_{i-1} = \gamma_i$. A run is accepting iff $q_{km_k} \in Q_F$; M accepts w iff there exists an accepting run of M on w. The MVPL $\mathcal{L}(M)$ accepted by M contains exactly all the words w accepted by M.

Operations over VPL and MVPL (complement, union, etc.) are set operations. Such operations are originally defined (La Torre et al., 2007) only when the alphabets of the participating languages are identical. We will relax this starting from Section 4.

We always denote by $\mathfrak{a}$ the symbol pushed on a stack by a call symbol a, setting in effect $\Gamma = \{\mathfrak{a} : a \in \Sigma_c\} \cup \{\perp\}$. This is done for presentation convenience only and without loss of generality. Whenever a call a pushes $\mathfrak{a}$ on the stack which is in turn popped off the stack by a return b, we say that a and b are *matched*.

3 SHUFFLE AND HIDING

The lack of closure of VPL under shuffle and under hiding calls or returns has been communicated to us privately (Madhusudan, 2008); for completeness we include a proof here. We also find that these properties (and associated problems) extend to MVPL.

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

Proof. Let $L_1 = \{{c_1}^n {r_1}^n : n \geq 0\}$ 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 $w_2 \in \{r_1, r_2\}^{p+q}$, $p,q \geq 0$. 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 $|w_2|_{r_2} = q$.

Let M be a vPDA that accepts $L_1 \,|||\, L_2$. M 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). M must push on the stack exactly one symbol for each c_1 and c_2 (for it needs to remember both p and q) and then recognize precisely p symbols r_1 and q symbols r_2. M 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). M cannot exist.

$L_1 \,|||\, L_2$ can be easily accepted by an MvPDA with two stacks; however, MVPL come with well-defined partitions, so we force c_1, c_2, r_1, and r_2 to belong to the same stack 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.

The language $\{(caaa)^n(rb)^n : n \geq 0\}$ is clearly VPL provided that c is a call, r is a return, and a and b are local symbols; however, hiding c yields the language $\{(aaa)^n(rb)^n : n \geq 0\}$, which cannot be VPL. Indeed, we must push for a, for otherwise we have no means of remembering n; once we decide that some a must push then *all* the a must push. The stack height becomes then $3n$, which cannot be compared by any vPDA with n or $2n$ (the number of returns, depending on whether we consider r a return, b a return, or both). 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 calls and returns for MVPL: we just pick one stack and build a language as above.

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

The usual operations over two MVPL (union, concatenation, etc.) are defined (La Torre et al., 2007) 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 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. *Let L' and L'' be any two MVPL over two alphabets $\widetilde{\Sigma}'_{n'}$ and $\widetilde{\Sigma}''_{n''}$, respectively. Any operation $@ \in \{\cup, \cap, \cdot, |||\}$ between L' and L'' has two variants. 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 $\widetilde{\Sigma}'_{n'}$. The* unrestricted *$@$ is defined between any two L' and L'' with the only restriction that the sets $(\Sigma'_l \cup \Sigma''_l)$, $(\bigcup_{1\leq i\leq n'} \Sigma'^i_c) \cup (\bigcup_{1\leq i\leq n''} \Sigma''^i_c)$, and $(\bigcup_{1\leq i\leq n'} \Sigma'^i_c) \cup (\bigcup_{1\leq i\leq n''} \Sigma''^i_c)$ (the sets of all local symbols, all calls, and all returns) 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 $\Sigma_l = \Sigma'_l \cup \Sigma''_l$.*
- *Take all the call-return pairs from both alphabets $\widetilde{\Sigma}'_{n'}$ and $\widetilde{\Sigma}''_{n''}$ and put them in $\widetilde{\Sigma}_x$. We obtain a (not necessarily valid) $n'+n''$-stack alphabet.*
- *Collapse the resulting alphabet as follows: For any (Σ_c^p, Σ_r^p) and (Σ_c^q, Σ_r^q) from $\widetilde{\Sigma}_x$ such that $\Sigma_c^p \cap \Sigma_c^q \neq \emptyset$ or $\Sigma_r^p \cap \Sigma_r^q \neq \emptyset$, eliminate (Σ_c^p, Σ_r^p) and (Σ_c^p, Σ_r^p) from $\widetilde{\Sigma}_x$ and introduce in $\widetilde{\Sigma}_x$ instead $(\Sigma_c^p \cup \Sigma_c^q, \Sigma_r^p \cup \Sigma_r^q)$. Keep collapsing $\widetilde{\Sigma}_x$ for as long as possible.*

It is immediate 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 whenever a call or return appear on two different stacks the two stacks collapse into one.

Note that a restricted 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.

Unfortunately, allowing unrestricted operations does not preserve the nice closure properties of MVPL.

Theorem 2. *MVPL are not closed under unrestricted union, concatenation, intersection, and shuffle.*

Proof. Consider the context-free language $L_{p<q} = \{{c_1}^n {c_2}^p {r_2}^q {r_1}^n : n \geq 0, 0 \leq p < q\}$ and assume that a vPDA M that accepts $L_{p<q}$ exists. The number of c_1 symbols is in direct relation with the number of r_1 symbols in the input and n can exceed the number of states in M, so one of c_1 and r_1 must be a call and the other must be a return. Since c_1 precedes r_1, the call (return) must be c_1 (r_1). For the same reasons c_2 must be a call and r_2 a return. After the inspection of all the c_1 and c_2, the stack will contain $n+p+1$ symbols. After q occurrence of r_2 there will be $n-(q-p)+1$ symbols on the stack. There is no way for M to remember the number $q-p$, so there is no way M can determine the value of n out of $n-(q-p)+1$ stack symbols. $L_{p<q}$ is thus not a VPL. $L_{p<q}$ is however trivially accepted by an MvPDA with two stacks. Similarly, the language $L_1 = \{{c_1}^n {r_2}^n : n \geq 0\}$ is trivially accepted by an MvPDA with one stack. Assume now that $L = L_{p<q} \cup L_1$ is accepted by an MvPDA M'. L_1 forces c_1 to be a call on the same stack as the return r_2. $L_{p<q}$ on the other hand forces c_1 to be a call on the same stack as the return r_1 and c_2 to be a call on the same stack as the return r_2. Thus M' becomes a one stack MvPDA, or a vPDA. However no vPDA can accept $L_{p<q}$ (as shown above).

Let M'' be an MvPDA that accepts $L'' = L_1 L_{p<q}$. M'' must too be a one stack MvPDA, or again a vPDA. L'' also inherits the problem of $L_{p<q}$: after the L_1 component a hypothetical MvPDA must behave like the vPDA M above, an impossibility.

Let now M''' be the MvPDA that accepts $L_{p<q} \,|||\, L_1$. M''' is again forced to be a one-stack MvPDA (or vPDA). We then have the same problem as the one we found earlier in Theorem 1.

MVPL are not closed under unrestricted intersection since this implies that they are also closed under unrestricted union (indeed, they are closed under complementation), which is not the case. □

5 DISJOINT OPERATIONS

The restrictions imposed originally over the MVPL operations yield good closure properties (except under shuffle), but they are somehow artificial. In a real concurrent system it is very common that the same function starts identically but then behaves differently in two threads (i.e., the partitions of the alphabet are not identical between the two threads). We also note that in practice the two threads operate on their own stack. We will attempt to capture such a behaviour of real systems via a third kind of *disjoint* operations.

Definition 2. *Let L be an MVPL over $\widetilde{\Sigma}_n = \langle (\Sigma_c^i, \Sigma_r^i)_{1 \leq i \leq n}, \Sigma_l \rangle$. The p-stack renaming $\mathbb{R}_p(L)$ of L is an MVPL over $\widetilde{\Sigma}'_n = \langle (\Sigma_c^i, \Sigma_r^i)_{1 \leq i \leq n, i \neq p}, (\Sigma_c^{n+1}, \Sigma_r^{n+1}), \Sigma_l \rangle$ such that there exists a bijection $f : \Sigma_c^p \cup \Sigma_r^p \to \Sigma_c^{n+1} \cup \Sigma_r^{n+1}$ with $f(x) \in \Sigma_c^{n+1}$ iff $x \in \Sigma_c^p$ and $f(x) \in \Sigma_r^{n+1}$ iff $x \in \Sigma_r^p$. $\mathbb{R}_p(L) = \{r(w) : w \in L\}$, where $r : \Sigma \to \Sigma'$ 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 $r(a_1 a_2 \ldots a_l) = r(a_1) r(a_2) \ldots r(a_l)$. By abuse of notation $\mathbb{R}_{p_1, p_2, \ldots, p_k}(L) = \mathbb{R}_{p_1}(\mathbb{R}_{p_2}(\ldots \mathbb{R}_{p_k}(L) \ldots))$.*

Symbols in a stack can be renamed to symbols in another stack. However, if we rename one symbol then we 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 before-renaming MVPL should not use any symbol from the new stack. The following is immediate:

Theorem 3. *A stack renaming $\mathbb{R}_{p_1, p_2, \ldots, p_k}(L)$ of a language L is MVPL iff L is an MVPL.* □

Using renaming judiciously, we get back the lost closure properties (and on top of it we also get closure under shuffle).

Theorem 4. *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 $\mathbb{R}_{1,\ldots,n'}$ such that $\mathbb{R}_{1,\ldots,n'}(L') \cup L''$, $\mathbb{R}_{1,\ldots,n'}(L') \cdot L''$, and $\mathbb{R}_{1,\ldots,n'}(L') \,|||\, L''$ are MVPL over an alphabet $\widetilde{\Sigma}_{n'+n''}$.*

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

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

$\mathbb{R}_{1,\ldots,n'}$ guarantees that there will be no common stack manipulation between M' and M''. That $\mathbb{R}_{1,\ldots,n'}(L') \cup L''$ and $\mathbb{R}_{1,\ldots,n'}(L') \cdot L''$ are MVPL is then immediate: The MvPDA that accepts the union consists in the union of M' and M''. For concatenation we take the initial states of M' as being initial states, 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''. The correctness of the constructions follow from the constructions that establish closure under union and concatenation for regular languages (Lewis and Papadimitriou, 1998).

The MvPDA $M = (Q, Q_I, \Gamma, \Delta, Q_F)$ that accepts $\mathbb{R}_{1,\ldots,n'}(L') \,|||\, L''$ simulate M' and M'' as follows: M must keep track of the state of both M' and M'', so $Q = Q' \times Q''$ (and $Q_I = Q'_I \times Q''_I$). 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 $((p', q''), \gamma) \xrightarrow{a'} ((q', q''), \eta)$ to Δ. Conversely, for any $q' \in Q'$, $p'', q'' \in Q''$, and $(p'', \gamma) \xrightarrow{a''} (q'', \eta) \in \Delta''$ we add $((q', p''), \gamma) \xrightarrow{a''} ((q', q''), \eta)$ to Δ. 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. 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. □

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. *The* disjoint union *[*concatenation, shuffle*] of two MVPL L' and L'' that can be combined using unrestricted operations is $\mathbb{R}_{1,\ldots,n'}(L') \cup L''$ (unrestricted union) [$\mathbb{R}_{1,\ldots,n'}(L') \cdot L''$ (unrestricted concatenation), $\mathbb{R}_{1,\ldots,n'}(L') \,|||\, L''$ (unrestricted shuffle)], where $\mathbb{R}_{1,\ldots,n'}$ renames the alphabet of L' such that no stack alphabet is common between $\mathbb{R}_{1,\ldots,n'}(L')$ and L''.*

Given Theorem 4, the following is immediate.

Corollary 5. *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 6. *Disjoint union, concatenation, and shuffle are commutative and associative up to a stack renaming.* □

6 A PRACTICAL NOTE

Consider again the `fork(2)` system call, with its two initially identical copies running concurrently and then diverging in their behaviour as time goes by. This cannot be specified using restricted operations. Indeed, consider the shuffle between L_1 (the traces of a parent process) and L_2 (the traces of a child process) 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 only one stack to begin with) whereas r_1 and r_2 belong to different stacks (for the two, diverging behaviours happen on two different stacks; we can reasonably assume that one needs to separate these behaviours, for instance because the child process performs a call to `execve(2)` and becomes a different process). Such a behaviour is naturally modelled by disjoint operations.

Our results thus open the possibility of using MVPL for specifying and verifying recursive, concurrent systems. The closure properties established here show that such systems can be specified compositionally, so that algebraic approaches are possible. We include here some preliminary considerations about a future MVPL-based algebraic specification mechanism.

We now have three kinds of operators: restricted (original), unrestricted, and disjoint. As we argued, it makes practical and also theoretical sense to use disjoint shuffle (the only variant under which MVPL are closed) over the other variants. 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 end of the spectrum, it makes no theoretical sense (and probably no practical sense) to use unrestricted operations, for MVPL is not closed under any of them. We argue that

one should use disjoint concatenation: Concatenation models two independent processes that follow one after the other, so it makes sense to give them separate sets of stacks. 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 we note that using disjoint union does not hurt (as the behaviour of the processes diverges irreversibly anyway).

How about intersection? The disjoint variant is meaningless. Intersection can be used to model two portion of processes that synchronize with each other over all their actions. In the disjoint setting 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 (Bruda and Bin Waez, 2009), using an intersection-like construct. Other than this, there is no need for intersection, so it can be safely ignored.

The only interesting closure property that does not hold for MVPL no matter what is closure under hiding. Hiding is used in process algebrae such as CSP for encapsulation, so that two processes synchronize on a well-defined common interface instead of any action that might happen to be common to both. The lack of closure under hiding for MVPL applies only to calls and returns; we can freely hide local symbols. But then the local symbols are (and practically speaking should be!) the only candidates for synchronization, so hiding local symbols serves nicely all the encapsulation purposes. In passing we note that calls and returns can actually be hidden (together with the symbols that fall in between) by using "abstract" operations (Alur et al., 2004), useful for specifying local properties of a recursive module.

In all, disjoint operations are a solid base for a parallel composition operator for MvPDA-based processes. This in turn allows for compositional, algebraic approaches to the conformance testing of systems with nested and potentially recursive invocations of program modules, such as application software.

7 SUMMARY

VPL capture the properties of systems with recursive modules, but the lack of closure under shuffle effectively prevents VPL-based compositional approaches to specifying concurrent systems. MVPL appear to model concurrent systems naturally but end up having the same limitations as VPL (Theorem 1).

The divergence of the parent and child processes created by `fork(2)` cannot be specified using restricted operations, as mentioned. Under relaxed (and more realistic) conditions however MVPL cease to be closed to almost any interesting operation, not just shuffle (Theorem 2). Based on the intuition of the the two processes created by `fork(2)` we introduced 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 4). 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 5).

These disjoint operations turn out to form a solid base for compositional, algebraic approaches to the conformance testing of complex programs such as application software. In particular, a process algebra should be immediate.

ACKNOWLEDGEMENTS

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

REFERENCES

Alur, R., Etessami, K., and Madhusudan, P. (2004). 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)*, pages 467–481. Springer.

Alur, R. and Madhusudan, P. (2004). Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 04)*, pages 202–211. ACM Press.

Bergstra, J. A. and Klop, J. W. (1988). Process theory based on bisimulation semantics. In de Bakker, J. W., de Roever, W., and Rozenberg, G., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 50–122. Springer.

Bruda, S. D. and Bin Waez, M. T. (2009). Communicating Visibly pushdown Processes. In *The 17th International Conference on Control Systems and Computer Science*, volume 1, pages 507–514.

Carotenuto, D., Murano, A., and Peron, A. (2007). 2-visibly pushdown automata. In *Proceedings of the 11th International Conference on Developments in Language Theory (DLT 2007)*, pages 132–144. Springer.

La Torre, S., Madhusudan, P., and Parlato, G. (2007). A robust class of context-sensitive languages. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 07)*, pages 161–170, Washington, DC, USA. IEEE Computer Society.

Lewis, H. R. and Papadimitriou, C. H. (1998). *Elements of the Theory of Computation*. Prentice-Hall.

Madhusudan, P. (2008). Private communication.