Technical Report 2019-002
CMVP, a Process Algebra Based on Multi-Stack Visibly Pushdown Languages and Their Disjoint Operations*

Davidson Madudu, Stefan D. Bruda
Department of Computer Science, Bishop's University
Sherbrooke, Quebec, Canada
emails: madudu@cs.ubishops.ca, stefan@bruda.ca

Md. Tawhid Bin Waez
Ford Motor Company
Dearborn, MI, USA
email: mwaez@ford.com

26 January 2019

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. The more general 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 these restrictions, then MVPL loose almost all their closure properties. Therefore we introduce 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 allows us to introduce an MVPL-based process algebra called Communicating Multi-stack Visibly Pushdown Processes (CMVP). CMVP defines a superset of CSP by combining the interesting properties of finite-state algebras (such as CSP) with the context-free features of MVPL. CMVP includes support for parallel composition and also for recursion.

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

1 Introduction

Modelling complex, recursive and concurrent systems is a major challenge in software verification. Current standard verification techniques such as model checking [13] or mainstream process algebras [4] are unable to model non-regular properties and so become quickly useless with the increase in complexity of the system being specified. Context-free techniques such as basic process algebra or BPA [6] present a number of issues stemming from the lack of closure of context-free languages under several operations, which in turn limit their compositionality. This has all lead to research in formal methods to develop a concurrent process algebra that would allow the specification and verification of concurrent and recursive systems.

A first step toward this goal is the introduction of visibly pushdown languages (VPL) [3] 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

*Part of 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.

(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 [21]. As a consequence attempts at specifying the behaviour of concurrent systems [10] found that such a specification is awkward at best and severely crippled at worst.

Multi-stack visibly pushdown languages (MVPL) [18] are an extension of VPL. They have most of the nice theoretical properties VPL enjoy and 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 $2n+1$ 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.

As interesting as MVPL look, we show in this paper that they have the same disadvantages as VPL when it comes to the compositional specification of 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. In addition, we also find that MVPL do not support the modelling of dynamic creation of threads. Indeed, closure under all the properties involving MVPL exist under very strict restrictions on the partitions of the two MVPL: once these restrictions are not in place 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 [18], 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; the 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, 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.

With the disjoint operations in place we are able to define an MVPL-based process algebra which we call CMVP. CMVP is an extension of CSP [24]. We define the syntax and semantics as usual, and then we also introduce a framework for CMVP trace specification and verification.

The remainder of this paper is organized as follows: We present in the next section the necessary preliminaries, while Section 3 summarizes the previous work in the area. We then address the various kind of operations over MVPL as follows: we show that the operations over MVPL as defined originally

lack necessary closure properties (Section 4), then we show that relaxing the definition of the MVPL operations to closer reflect the practice of computing systems results in a set that is not closed under most interesting operations (Section 5), and then we finally introduce our "disjoint operations" which turn out to have the desired closure properties (Section 6) while also being faithful to the real world. Based on these operations we define the syntax and the structural operational semantics of CMVP (Section 7), followed by the trace semantics of CMVP (Section 8) and finally a framework for trace specification and verification in CMVP (Section 9). We conclude in Section 10.

Part of this work has been published in preliminary form in conference proceedings [11]. This publication covers in preliminary form a large part of Sections 4, 5, and 6.

2 Preliminaries

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

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 $L \setminus A = \{w \setminus A : w \in L\}$.

Sequences are denoted by listing the elements of the sequence surrounded by angled brackets. The empty sequence is therefore $\langle\rangle$. Given a set A, A^* denotes the set of all finite sequences of symbols from A. The concatenation of two sequences seq_1 and seq_2 is denoted by $seq_1.seq_2$. Note that concatenation is associative. The sequence seq^n is the concatenation of n copies of the finite sequence seq, with $seq^0 = \langle\rangle$. If seq is a non-empty sequence then it can be rewritten as $a.seq'$ where a is the head of seq and seq' is the tail of seq. That is, $head(a.seq) = a$ and $tail(a.seq) = seq$. If $seq = seq''.b$ for some symbol b we define $init(seq) = seq''$ and $foot(seq) = b$. The length of a sequence seq denoted by $|seq|$. By abuse of notation $a \in seq$ is true iff the symbol a appears in the sequence seq. We use $\sigma(seq)$ to denote the set of all symbols that are in the sequence seq. We extend a mapping f on the elements of seq to sequences such that $f(seq)$ is the sequence obtained by applying f to all the elements of seq.

2.1 Visibly Pushdown Languages

A vPDA [3] 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 $w = a_1a_2\ldots a_k$ is a sequence of configurations $(q_0,\gamma_0)(q_{01},\gamma_0)\cdots(q_{0m_0},\gamma_0)(q_1,\gamma_1)(q_{11},\gamma_1)\cdots(q_{1m_1},\gamma_1)(q_2,\gamma_2)\cdots(q_k,\gamma_k)(q_{k1},\gamma_k)\cdots(q_{km_k},\gamma_k)$ such that $\gamma_0 = \perp$, $q_0 \in Q_I$, $(q_{j-1i},\varepsilon) \xrightarrow{\varepsilon} (q_{ji},\varepsilon) \in \Delta$ for all $1 \le i \le k$, $1 \le j \le m_i$, and $(q_{i-1m_{i-1}}, \gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Delta$ for every $1 \le i \le k$ 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 $\widetilde{\Sigma_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\le i\le n}, \Sigma_l\rangle$ 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) [18] over the n-stack call-return alphabet $\widetilde{\Sigma_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\le i\le n}, \Sigma_l\rangle$ is then a natural extension of a vPDA. It is 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 $\gamma = \perp$ 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 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 $\gamma = \langle \gamma^1, \ldots, \gamma^n\rangle$ with $\gamma^l \in (\Gamma \setminus \{\perp\})^*\perp$ for all $1 \le l \le n$. For a word $w = a_1 a_2 \ldots a_m \in \Sigma^*$ 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 $\gamma_0^l = \perp$ for all $1 \le l \le n$, $q_0 \in Q_I$, $(q_{j-1i},\varepsilon) \xrightarrow{\varepsilon} (q_{ji},\varepsilon) \in \Delta$ for all $1 \le i \le k$, $1 \le j \le m_i$; whenever $a_i \in \Sigma_c^p \cup \Sigma_r^p$, $\gamma_{i-1}^l = \gamma_i^l$ for all $l \ne p$, $(q_{i-1m_{i-1}}, \gamma_{i-1}') \xrightarrow{a_i} (q_i, \gamma_i') \in \Delta$ for every $1 \le i \le k$ 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 [18] only when the alphabets of the two languages are identical. We will however relax this condition starting from Section 5.

In the following we always denote by $\mathfrak{a}$ the symbol pushed on a stack by a call symbol a, setting in effect $\Gamma = \{\mathfrak{a} : a \in \Sigma_c\} \cup \{\perp\}$. 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*.

2.3 Communicating Sequential Processes

Our process algebra will be developed as an extension of Communicating Sequential Processes (CSP), and so our development is also based on the development of CSP. CSP models a the behaviour of a system or process using eight major operators: event prefix, choice, recursion, parallel composition, hiding, renaming, sequential composition, and interrupt [5, 8, 7, 14, 15, 22, 23, 24]. A prefix or suffix of a process is also regarded as a process, therefore the domain language is closed under prefix and suffix. The syntax of CSP is defined as follows:

$$S \quad ::= \quad x : A \to S(x) \,|\, S \,\Box\, R \,|\, S \sqcap R \,|\, X \,|\, S_A\|_B R \,|\, S \setminus A \,|\, f(S) \,|\, f^{-1}(S) \,|\, S;R \,|\, S \triangle R \,|\, STOP \,|\, SKIP$$

where Σ is the set of (elementary) actions, S and R range over CSP processes, x ranges over Σ, A and B range over 2^Σ, and f ranges over the set $\{f : \Sigma \to \Sigma : \forall a \in \Sigma\text{: } f^{-1}(a) \text{ is finite} \wedge f(a) = \checkmark \text{ iff } a = \checkmark\}$ of

[1] This set is called the set of internal actions originally [18]. However, this notation conflicts with the notion of internal action in a labelled transition system [16], so we revert to the original terminology used for VPL [3].

Σ-transformations.

The most common underlying semantical model of CSP (like any other process algebra) is the labelled transition system. A labelled transition system (LTS) [9] is a tuple $(\Theta, \Sigma, \Delta, I)$, where Θ is a set of states , Σ is a finite set of actions (not containing the internal action τ), $I \in \Theta$ is the initial state, and Δ is the transition relation such that $\Delta \subseteq \Theta \times (\Sigma \cup \{\tau\}) \times \Theta$. If Δ is unambiguous and understood from the context, then we often use the following shorthands: $P \xrightarrow{a} Q$ whenever $(P,a,Q) \in \Delta$, $P \xrightarrow{a}$ whenever there exists a Q such that $P \xrightarrow{a} Q$, and $P \not\xrightarrow{a}$ whenever $P \xrightarrow{a}$ does not hold.

A run of a labelled transition system M is a sequence $q_0 \tau q_{01} \tau \cdots \tau q_{0m_0} a_1 q_1 \tau q_{11} \tau \cdots \tau q_{1m_1} a_2 q_2 \cdots a_k q_k \tau q_{k1} \tau \cdots \tau q_{km_k}$ such that $q_0 = I$, $q_{j-1i} \xrightarrow{\tau} q_{ji}$ for all $1 \le i \le k$, $1 \le j \le m_i$, and $q_{i-1m_{i-1}} \xrightarrow{a_i} q_i$ for all $1 \le i \le k$. The trace of this run is the sequence $a_1 a_2 \cdots a_k$. The run is maximal whenever there is no x such that $q_{km_k} \xrightarrow{x}$. The trace of a maximal run is called a complete trace. The language traces(M) contains exactly all the traces of all the possible runs of M. Similarly, ctraces(M) contains exactly all the complete traces of all the possible maximal runs of M.

One way of presenting the operational semantics of CSP using LTS is by using structural operational semantics rules as shown in Figure 1, where $A^{\checkmark} = A \cup \{\checkmark\}$.

In addition to operators, CSP features two basic processes namely, *STOP* and *SKIP*. *STOP* is a process which does not execute any event, while *SKIP* can only execute the termination event ✓. A process that proves useful in trace semantics is RUN_A, which can perform any action in A and so is defined as $RUN_A = (x : A \to RUN_A) \ \Box \ SKIP$. The process RUN_Σ is often written *RUN*.

2.4 Trace Semantics

The set of all the sequences of actions that a process can perform (or that might possibly be recorded) is the set of traces of that process [24]. The set of all possible traces of a process P is traces(P). Any termination event ✓ occurring in a trace must appear at the end. The set of all traces is defined as: $TRACE = \{tr | \sigma(tr) \subseteq \Sigma^{\checkmark} \wedge |tr| \in \mathbb{N} \wedge \checkmark \notin \sigma(init(tr))\}$. Since all traces are sequences, they inherit all of the operators over sequences. However, sequence concatenation maps traces $tr1$ and $tr2$ to a trace $tr1.tr2$ only if $\checkmark \notin \sigma(tr1)$. Thus tr^n will be a trace only if $\checkmark \notin \sigma(tr)$. If a function f maps Σ to Σ and $f(\checkmark)$ to ✓, then $f(tr)$ will always be a trace. The notation $P \stackrel{tr}{\Rightarrow} P'$ states that there exists a sequence of transitions whose initial process is P and whose final process is P' after executing tr. The notation $P \stackrel{tr}{\Rightarrow}$ is a shorthand for $\exists P' : P \stackrel{tr}{\Rightarrow} P'$.

Trace semantics models each process by associating it with its set of traces. Processes are *trace equivalent* iff they have exactly the same set of possible traces: $P =_T Q$ iff traces(P) = traces(Q). The theory of trace equivalence allows the definition of algebraic laws for individual operators, and also laws concerning the relationships between various operators.

In the trace model, a specification of a process [7, 24] is given in terms of the traces the process may engage in. A process satisfies its specification if all of its executions are acceptable. If $S(tr)$ is a predicate on trace tr, then process P satisfies $S(tr)$ if $S(tr)$ holds for every traces tr of P: $P \vdash S(tr) = \forall tr \in \text{traces}(P) : S(tr)$. To satisfy a trace specification it is necessary to ensure that no violating action occur at any point in an execution. This kind of specification is referred to as a *safety* specification. It stipulates that nothing 'bad' should ever happen, and is precisely the kind of property that is expressed as a specification on traces.

Trace semantics [7, 24] have a compositional nature which paves the way for a compositional proof system that can be applied in describing trace specifications. Hence, the specification of a process can be deduced from the specifications of its components, in a way which reflects the trace semantics of the operators. The proof system is defined as a set of proof (or inference) rules for all of the operators. Each rule provides a specification which holds for a composite process starting from antecedents which

$$\frac{}{P \xrightarrow{\tau} Q} \qquad \frac{}{(x : A \to P(x)) \xrightarrow{a} P(a)} \; [a \in A]$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{P \xrightarrow{\mu} P'}{N \xrightarrow{\mu} P'} \; [N = P]$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} Q} \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P_A \|_B Q \xrightarrow{a} P'_A \|_B Q'} \; [a \in A^{\checkmark} \cap B^{\checkmark}]$$

$$\frac{P \xrightarrow{a} P'}{\begin{array}{l} P \,\square\, Q \xrightarrow{a} P' \\ Q \,\square\, P \xrightarrow{a} P' \end{array}} \qquad \frac{P \xrightarrow{\mu} P'}{\begin{array}{l} P_A \|_B Q \xrightarrow{\mu} P'_A \|_B Q \\ Q_B \|_A P \xrightarrow{\mu} Q_B \|_A P' \end{array}} \; [\mu \in A \cup \{\tau\} \setminus B]$$

$$\frac{P \xrightarrow{\tau} P'}{\begin{array}{l} P \,\square\, Q \xrightarrow{\tau} P' \,\square\, Q \\ Q \,\square\, P \xrightarrow{\tau} Q \,\square\, P' \end{array}} \qquad \frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A} \; [\mu \notin A]$$

$$\frac{P \xrightarrow{\checkmark} P'}{P; Q \xrightarrow{\tau} Q} \qquad \frac{P \xrightarrow{\mu} P'}{P \triangle Q \xrightarrow{\mu} P' \triangle Q} \; [\mu \neq \checkmark]$$

$$\frac{P \xrightarrow{f(a)} P'}{f^{-1}(P) \xrightarrow{a} f^{-1}(P')} \qquad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \; [a \in A]$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \triangle Q \xrightarrow{\checkmark} P'} \qquad \frac{P \xrightarrow{\mu} P'}{f(P) \xrightarrow{\mu} f(P')} \; [\mu \in \{\tau \cup \checkmark\}]$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'} \qquad \frac{P \xrightarrow{\mu} P'}{P; Q \xrightarrow{\mu} P'; Q} \; [\mu \neq \checkmark]$$

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')} \qquad \frac{P \xrightarrow{\mu} P'}{f^{-1}(P) \xrightarrow{\mu} f^{-1}(P')} \; [\mu \in \{\tau \cup \checkmark\}]$$

$$\frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'}$$

Figure 1: Operational Semantics of CSP [15]

describe specifications that hold for the component processes.

3 Previous Work

The formal verification field has been enriched by the relatively recent introduction of the class of multi-stack visibly pushdown languages (MVPL). MVPLs are a natural extension of visibly pushdown languages (VPL) [18]. Both VPL and MVPL have useful applications in the modelling of multithreaded recursive systems, although it could be argued that MVPL models recursive systems in a more expres-

sive and natural manner than VPL [12, 18]. MVPL are defined using a multi-stack visibly pushdown automaton whose computation is split into k stages such that only one stack can be popped in each stage. Intermediate models also exist, such as the 2-stack visibly pushdown automata (2-VPDA) [12].

Earlier studies [12] showed 2-VPDA are closed under all Boolean operations and are determinizable in EXPTime. However it was later shown in [17] that 2-VPDA are underterminizable and it was instead explained [19] that only the class of languages accepted by multi-stack pushdown automata with a bounded phase are determinizable. In this context a multi-stack PDA (MPDA) is defined similarly to a MvPDA except that the input alphabets no longer determine the stack operations [19]. A bounded phase is a restriction on the computation of class of strings that are accepted by an automaton. For a bounded phase MPDA, a uniform bound k is fixed and only strings that can reduced into k substrings with each substring having at most one kind of return node are considered to be in the language accepted by the bounded phase MPDA [19].

Various researchers have also attempted to use VPLs to model and to check equivalence of recursive and concurrent systems. One study [25] examines the equivalence checking on visibly pushdown automata and showed that the complexity (upper and lower bound) for simulation, completed simulation, ready simulation, 2-nested simulation preorders/equivalences and bisimulation equivalence are EXP-time complete. Other research [10] attempted to use VPL to develop a process algebra that would be a superset of CSP, suitable for modelling recursive and concurrent systems. However, as a result of lack of closure under shuffle the resulting process algebra proved to be awkward and of limited use.

Finally, part of this work is motivated by research done in both CARET and NWTL temporal logics [1, 2]. Both temporal logics have been proposed for the formal verification of application software as they are able to effectively specify and verify non-regular properties like partial and total correctness and access control properties of application software [1, 2]. All of these efforts pave the way for a fully compositional MVPL-based specification for the verification of recursive and concurrent systems.

4 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 [21]; 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). □

5 Unrestricted Operations over MVPL

The usual operations over two MVPL (union, concatenation, etc.) are defined [18] 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 $(\Sigma'_l \cup \Sigma''_l)$, $(\bigcup_{1\leq i\leq n'} \Sigma'^i_c) \cup (\bigcup_{1\leq i\leq n''} \Sigma''^i_c)$, and $(\bigcup_{1\leq i\leq n'} \Sigma'^i_c) \cup (\bigcup_{1\leq i\leq n''} \Sigma''^i_c)$ 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 $x \leq n'+n''$).*

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 $S \to c_1 S r_1$, $S \to B$, $B \to c_2 B r_2$, $B \to r_2 C$, $C \to r_2 C$, $C \to \varepsilon$.

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 [18]. 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 [20] (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. □

6 Disjoint Operations over MVPL

The restrictions imposed originally over the MVPL operations [18] 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 $\widetilde{\Sigma_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\le i\le n}, \Sigma_l\rangle$. The p-stack renaming $\mathscr{R}_p(L)$ of L is an MVPL over the n-stack call-return alphabet $\widetilde{\Sigma'_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\le i\le n, i\ne p}, (\Sigma_c^{n+1}, \Sigma_r^{n+1}), \Sigma_l\rangle$ 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, $\mathscr{R}_p(L) = \{r(w) : w \in L\}$, 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 $r(a_1 a_2 \ldots a_l) = r(a_1) r(a_2) \ldots r(a_l)$. By abuse of notation $\mathscr{R}_{p_1,p_2,\ldots,p_k}(L) = \mathscr{R}_{p_1}(\mathscr{R}_{p_2}(\ldots \mathscr{R}_{p_k}(L)\ldots))$. 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 $\mathscr{R}_{p_1,p_2,\ldots,p_k}(L)$ 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 $\mathscr{R}_{1,\ldots,n'}$ such that $\mathscr{R}_{1,\ldots,n'}(L') \cup L''$, $\mathscr{R}_{1,\ldots,n'}(L') \circ L''$, and $\mathscr{R}_{1,\ldots,n'}(L') \,|||\, L''$ 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 $\mathscr{R}_{1,\ldots,n'}$.

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

$\mathscr{R}_{1,\ldots,n'}$ guarantees that there will be no stack manipulation that is common between M' and M''. That $\mathscr{R}_{1,\ldots,n'}(L') \cup L''$ 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 [20].

For $\mathscr{R}_{1,\ldots,n'}(L') \circ L''$ 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 $\mathscr{R}_{1,\ldots,n'}(L') \,|||\, L''$ 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 $\mathscr{R}_{1,\ldots,n'}(L') \cup L''$ (unrestricted union) [$\mathscr{R}_{1,\ldots,n'}(L') \circ L''$ (unrestricted concatenation), $\mathscr{R}_{1,\ldots,n'}(L') \,|||\, L''$ (unrestricted shuffle)], where $\mathscr{R}_{1,\ldots,n'}$ renames the alphabet of L' in such a way that no stack alphabet is common between $\mathscr{R}_{1,\ldots,n'}(L')$ 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.*

7 Communicating Multi-Stack Visibly pushdown Processes

A communicating multi-stack visibly pushdown (or CMVP) process is an agent which interacts with its environment (itself also regarded as a process) by performing certain events drawn from a multi-stack visibly pushdown n-stack of call, return alphabets and local symbols : $\Sigma_c = \bigcup_{i=1}^{n} \Sigma_c^i$, $\Sigma_r = \bigcup_{i=1}^{n} \Sigma_r^i$, $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_l$. The syntax of CMVP will be based on the following description:

$$S \quad ::= \quad x : A \to S(x) \,|\, S \,\Box\, R \,|\, S \sqcap R \,|\, X \,|\, S_A \|_B R \,|\, S \setminus A \,|\, \overline{S} \,|\, f(S) \,|\, f^{-1}(S) \,|\, S;R \,|\, S \triangle R \,|\, STOP \;|\, SKIP$$

where S, R and X range over CMVP processes, x over $\widetilde{\Sigma}$, A and B over $2^{\widetilde{\Sigma}}$, f over the set $\{f : \widetilde{\Sigma} \to \widetilde{\Sigma} : \forall a \in \widetilde{\Sigma}$: $f(a), f^{-1}(a) \in \Sigma_c\ [\Sigma_l, \Sigma_r]$ iff $a \in \Sigma_c\ [\Sigma_l, \Sigma_r] \wedge f^{-1}(a)$ is finite $\wedge\ f(a) = \checkmark$ iff $a = \checkmark \wedge f(a) = \perp$ iff $a = \perp\}$ of $\widetilde{\Sigma}$-transformations. All the common operators between CMVP and CSP have a similar constructions in LTS semantics with the exception of the parallel composition operator. Indeed, the CMVP parallel composition operator applies the stack renaming as described Section 6 to the processes it operates on, though this operation is implicit; details are provided in Section 7.1. The new operator "Abstract" $\bar{\cdot}$ is also introduced; it can be used to hide the sub-modules of a module (further discussed in Section 7.1). The notion of a module is defined naturally using call and return symbols: A new module is launched once a call c is performed; the execution of that module lasts until the return matching c. The module may perform local actions but also (possibly recursive) calls to other modules (sometimes called "sub-modules" of that module).

The process definitions for both *STOP* and *SKIP* are the same in CSP and CMVP. Just like CSP process *STOP* in CMVP is never prepared to perform any event hence it has no transitions, while *SKIP* is a state in a process that represents the successful termination of that process (the only event it can perform is the termination event $\checkmark$). Every other CMVP process P_Γ is defined as consisting of an MvPDA state P and a finite number of stacks represented as an n-tuple $\Gamma = (\gamma_1, \ldots, \gamma_n)$. $P_\varepsilon = P_{(\perp,\perp,\ldots,\perp)}$ is defined as a CMVP process P with all its stacks empty. Other than P_ε, an LTS state corresponding to a CMVP process will be represented syntactically as $P_{(i/\gamma)}$, where P is the current MvPDA state, i represents the stack currently being operated on, $1 \le i \le n$, and γ is a current stack prefix of the i-th MvPDA stack. The implicit assumption of this notation is that all the other stacks will be unchanged, but the prefix γ of the i-th stack will be altered as a result of the current operation. It should be noted that in our algebra each individual action can only affect a single stack. Therefore our CMVP syntax is refined as follows:

$$\begin{aligned} P_{i/\gamma} \quad ::= \quad & x : A \to P(x)_{i/\gamma} \,|\, P_{i/\gamma} \,\Box\, Q_{i/\delta} \,|\, P \sqcap Q \,|\, N_{i/\gamma} \,|\, P_{i/\gamma A} \|_B Q_{i/\delta} \,|\, P \setminus A \\ & |\, \overline{P_{i/\gamma}} \,|\, f(P_{i/\gamma}) \,|\, f^{-1}(P_{i/\gamma}) \,|\, P_{i/\gamma}; Q_{i/\delta} \,|\, P_{i/\gamma} \triangle Q_{i/\delta} \,|\, STOP \;|\, SKIP \end{aligned}$$

where P, Q, N range over MvPDA states, i represents the stack being affected by the current operation in the set of stacks Γ and γ and δ represent some finite prefix of the current stack content of the stack in operation i. Given that an MvPDA operation manipulates only the top symbol of a stack, it is further the case that $|\gamma| \le 1$ and $|\delta| \le 1$. When the length of a stack prefix is zero then we will not mention that prefix at all (for brevity and also to make sure that CSP is a subset of CMVP). We thus reach the final syntax of CMVP:

$$\begin{aligned} P' \quad ::= \quad & x : A \to P(x)' \,|\, P' \,\Box\, Q' \,|\, P \sqcap Q \,|\, N' \,|\, P'_A \|_B Q' \,|\, P \setminus A \\ & |\, \overline{P'} \,|\, f(P') \,|\, f^{-1}(P') \,|\, P'; Q' \,|\, P' \triangle Q' \,|\, STOP \;|\, SKIP \end{aligned}$$

with $P' ::= P \,|\, P_{(i/a)}$, $Q' ::= Q \,|\, Q_{(i/b)}$, and a and b ranging over Γ. When an operator executes a local action we do not mention any stack since we do not operate on any stack. However, if we have have an operation that executes a call or return action we denote the top of the stack by the putting the subscript i/a next to the process, where as stated earlier i would represent the current stack in operation among

the set of stacks Γ, and a represents the symbol at the top of the current stack content of the stack in operation i. After the execution of a call or return action there will always be a change (push or pop) in the stack content γ with the exception of when a return action is performed on an empty stack (which will have the symbol $\perp$ at the top of its stack to indicate it is empty).

7.1 The Operational Semantics of CMVP

The subscripts l, c and r are used to represent the sets of local, call and return events, respectively. Hence, any interface $A \in 2^{\widetilde{\Sigma}}$ for n stacks will be the union of $2n+1$ sets A_l, $(A_c)_i$, $(A_r)_i$, $1 \leq i \leq n$. It should be noted that unlike call and return symbols, local symbols are global to a CMVP process. Any call action b that is a member of an interface A is denoted as $b \in (A_c)_i$ meaning that b belongs to the call symbols in interface A and it operates on the i-th stack. Similarly a return action b in an interface A is denoted $b \in (A_r)_i$. A CMVP operation is allowed between CMVP processes only if their stack alphabets $\widetilde{\Sigma}'$ and $\widetilde{\Sigma}''$ do not overlap[2] (else the main restriction of our MVPL is violated). We will guarantee that the stacks of CMVP processes will not overlap by applying as needed stack renaming to the stacks of one process.

The *matched* calls and returns are established by the specification (which will determine which return can pop (or *match*) which call: The matching calls of a return event are defined at specification time by specifying which stack symbols can be popped by the given return. *Balanced* calls and returns on the other hand are determined at run time: A return *balances* a call if it is labelled as a matching return of that call in the specification and also happens to match that call at run time. For example, if it is specified that $\{a,b\} \subseteq \Sigma_c^i$ and $c \in \Sigma_r^i$ which will pop either a or b in stack i, then c is the matching return of both a and b; however, during one particular execution is possible that c will only balance a but never b. It must be noted that unbalanced returns are accepted in CMVP processes whenever they appear when the respective stack is empty. In the event that there is an unbalanced return, the (empty) stack content of the CMVP process will remain unchanged.

For ease of presentation we assume without loss of generality that there is a one-to-one mapping between the set of stack symbols and the set of calls, and so a call event for the ith stack will be written a^i and will push a on the i-th stack of that process. Similarly, a return event for stack i will be denoted by b^i. The one-to-one mapping of calls and stack symbols is without loss of generality because we can still specify what can and cannot be popped by a return symbol; the aforementioned one-to-one mapping will thus not limit the capability of defining matched calls and returns. Superscripts are not used on local events because local events are considered to be global and there is no stack manipulation during their execution.

For the reminder of this paper the stack in operation of CMVP processes will grow to the left, therefore the rightmost place on the stack in operation is reserved for the bottom-of-stack symbol $\perp$. Given a tuple $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_i, \ldots, \gamma_n)$ we denote the tuple $(\gamma_1, \gamma_2, \ldots, \gamma, \ldots, \gamma_n)$ by $\Gamma(i/\gamma)$. That is, $\Gamma(i/\gamma)$ is the tuple Γ with the i-th component replaced by γ.

Prefix Choice Prefix choice can be introduced syntactically in six different forms: $P = a \to P'$, $P = b^i \to P'_{(i/b)}$, $P_{(i/c)} = c^i \to P'_{(i/\perp)}$, $P_{(i/\perp)} = d^i \to P'_{(i/\perp)}$, $P_{(i/\perp)} = STOP$ and $P_{(i/\perp)} = SKIP$. From these syntactic rules it can be determined that a is local, b^i is a call, c^i is a balanced return, and d^i is an unbalanced return. $P_{(i/\perp)} = STOP$ [$P_{(i/\perp)} = SKIP$] requires that the process enter the *STOP* [*SKIP*] state when the MvPDA state is P and the top of the stack in operation i is empty ($\perp$).

Generally, a system can be specified with as well as without an explicit partitioning of its events. However, if an explicit partition is not given, then a process can be represented as a sequence of actions (similarly with CSP) only when all the actions are local actions. On the other hand, any finite process

[2]Meaning $\Sigma' \cap \Sigma'' = \emptyset$ for any $\Sigma' \in \{\Sigma_c^i, \Sigma_r^i, 1 \leq i \leq n\}$ and $\Sigma'' \in \{\Sigma_c^k, \Sigma_r^k, 1 \leq k \leq m\}$.

$$\frac{}{(x:A\to P(x))_\Gamma \xrightarrow{a} P(a)_\Gamma}\ [a\in(A_l)]$$

$$\frac{}{(x:A\to P(x))_{\Gamma(i/\gamma)} \xrightarrow{a^i} P(a)_{\Gamma(i/a\gamma)}}\ [a^i\in(A_c)_i]$$

$$\frac{}{(x:A\to P(x))_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P(a)_{\Gamma(i/\gamma)}}\ [a^i\in(A_r)_i]$$

$$\frac{}{(x:A\to P(x))_{\Gamma(i/\perp)} \xrightarrow{a^i} P(a)_{\Gamma(i/\perp)}}\ [a^i\in(A_r)_i]$$

Figure 2: Prefix choice

$$(a)\quad \frac{}{P_\Gamma \xrightarrow{\tau} Q_\Gamma} \qquad (b)\quad \frac{}{P_\Gamma \sqcap Q_\Gamma \xrightarrow{\tau} P_\Gamma} \quad \frac{}{P_\Gamma \sqcap Q_\Gamma \xrightarrow{\tau} Q_\Gamma}$$

Figure 3: Internal action (a), internal choice (b)

(including processes with calls and returns) can be represented as a sequence (desirable in a large system), provided that we specify a partition on its events. For instance, let P_ε be the following process without an explicit partition: $P = a \to P_{(1/a)}$, $P = b \to P1$, $P1 = e \to P2_{(1/e)}$, $P2 = d \to P3$, $P3_{(1/e)} = f \to P4_{(1/\perp)}$, $P4_{(1/a)} = c \to P4_{(1/\perp)}$, $P4_{(1/\perp)} = STOP$. The process can be written with an explicit partition $A_l = \{b,d\}$, $(A_c)_1 = \{a^1, e^1\}$, $(A_r)_1 = \{c^1, f^1\}$ as follows: $P_{(1/\perp)} = a^1 \to P_{(1/a)}$, $P = b \to e^1 \to d \to P3_{(1/ea)}$, $P3_{(1/ea)} = f^1 \to P4_{(1/a)}$, $P4_{(1/a)} = c^1 \to P4_{(1/\perp)}$, $P4_{(1/\perp)} = STOP$. Figure 2 shows the semantics of prefix choice.

Internal Event A CMVP process can perform actions not noticeable to the environment called internal and denoted by τ. Internal actions can change the current MvPDA state of a process but will not change the stack content of any stack. The behaviour of the τ transition is described in Figure $3(a)$.

Choice The semantics of internal and external choice are given in Figures $3(b)$ and 4, respectively. The choice operator does not modify the matched calls and returns. The set of stacks of the composite process

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{\begin{array}{l}P_\Gamma \,\square\, Q_\Gamma \xrightarrow{\tau} P'_\Gamma \,\square\, Q_\Gamma \\ Q_\Gamma \,\square\, P_\Gamma \xrightarrow{\tau} Q_\Gamma \,\square\, P'_\Gamma\end{array}} \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{\begin{array}{l}P_\Gamma \,\square\, Q_\Gamma \xrightarrow{a} P'_\Gamma \\ Q_\Gamma \,\square\, P_\Gamma \xrightarrow{a} P'_\Gamma\end{array}}\ [a\in(A_l)]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{\begin{array}{l}P_{\Gamma(i/\gamma)} \,\square\, Q_\Gamma \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)} \\ Q_\Gamma \,\square\, P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}\end{array}}\ [a^i\in(A_c)_i] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{\begin{array}{l}P_{\Gamma(i/a\gamma)} \,\square\, Q_\Gamma \xrightarrow{a^i} P'_{\Gamma(i/\gamma)} \\ Q_\Gamma \,\square\, P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}\end{array}}\ [a^i\in(A_r)_i]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{\begin{array}{l}P_{\Gamma(i/\perp)} \,\square\, Q_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)} \\ Q_{\Gamma(i/\perp)} \,\square\, P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}\end{array}}\ [a^i\in(A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{\begin{array}{l}P_\Gamma \,\square\, Q_\Gamma \xrightarrow{\checkmark} P'_\Gamma \\ Q_\Gamma \,\square\, P_\Gamma \xrightarrow{\checkmark} P'_\Gamma\end{array}}$$

Figure 4: External choice

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{N_\Gamma \xrightarrow{\tau} P'_\Gamma}\ [N=P] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{N_\Gamma \xrightarrow{a} P'_\Gamma}\ [a \in (A_l), N=P]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{N_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}\ [a^i \in (A_c)_i, N=P] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{N_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}\ [a^i \in (A_r)_i, N=P]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{N_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}\ [a^i \in (A_r)_i, N=P] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{N_\Gamma \xrightarrow{\checkmark} P'_\Gamma}\ [N=P]$$

Figure 5: Recursion

in a choice construct is also similar to the set of stacks of the component processes of the construct. A process that chooses (once!) between '$]$' and '$\rangle$' as balanced return for '$[$' can be defined as follows: $P = [^1\rightarrow\ P_{(1/[)}, P_{(1/[)} =]^1 \rightarrow\ P1_{(1/\perp)}, P_{(1/[)} =\rangle^1 \rightarrow\ P2_{(1/\perp)}, P1_{(1/\perp)} = STOP, P2_{(1/\perp)} = STOP$. Note that '$]$' and '$\rangle$' are both matching returns of '$[$' (since they both pop it from the stack), but only one is used as a balanced return, depending on the environment. In contrast, a process Q defined with the following rules: $Q = [^1\rightarrow\ Q_{(1/[)}, Q = \tau \rightarrow\ Q1, Q = \tau \rightarrow\ Q2, Q1_{(1/[)} =]^1 \rightarrow Q1_{(1/\perp)}, Q2_{(1/[)} =\rangle^1 \rightarrow\ Q2_{(1/\perp)}, Q1_{(1/\perp)} = STOP, Q2_{(1/\perp)} = STOP$, makes an internal choice to transition from a state Q to either states $Q1$ or $Q2$ independent of the environment, and so the decision of whether '$]$' or '$\rangle$' matches '$[$' does not depend on the environment anymore.

Recursion Recursion in CSP and CMVP are defined similarly. However, a CSP recursive process creates loops among LTS states while CMVP recursive processes creates loops among MvPDA states and can perform recursive function calls using call events. A recursive MvPDA state may perform a sequence of calls and returns in addition to local actions before transitioning back into itself, so that each recursive loop can change the stack content of the stack in operation. If the changes made to the stack content of the set of stacks is zero (i.e. if the executed events are all local events), then the recursive process can be represented by a finite state machine just like in CSP.

The semantics of recursion is shown in Figure 5. Consider as an example the following CMVP recursive processes, which produces balanced parentheses: $P = (^1\rightarrow P_{(1/()}, P_{(1/()} =)^1 \rightarrow P_{(1/\perp)}, P_{(1/\perp)} = STOP$. P_ε can produce an infinite number of LTS states and infinitely many possible traces although we only have one MvPDA state P. Here is how a CMVP process P_ε can produce the trace $(()())$:

$$\begin{aligned} P_\varepsilon &= (^1\rightarrow P_{((\perp)} = (^1\rightarrow (^1\rightarrow P_{(((\perp)} = (^1\rightarrow (^1\rightarrow)^1 \rightarrow P_{((\perp)} = (^1\rightarrow (^1\rightarrow)^1 \rightarrow (^1\rightarrow P_{(((\perp)} \\ &= (^1\rightarrow (^1\rightarrow)^1 \rightarrow (^1\rightarrow)^1 \rightarrow P_{((\perp)} = (^1\rightarrow (^1\rightarrow)^1 \rightarrow (^1\rightarrow)^1 \rightarrow)^1 \rightarrow P_{(\perp)} \\ &= (^1\rightarrow (^1\rightarrow)^1 \rightarrow (^1\rightarrow)^1 \rightarrow)^1 \rightarrow STOP \end{aligned}$$

Consider now the process $Q = (^1\rightarrow Q1_{(1/()}, Q1_{(1/()} =)^1 \rightarrow Q_{(1/\perp)}, Q_{(i/\perp)} = STOP$. It may have infinitely long traces and can be written as follows: $Q_\varepsilon = (^1\rightarrow)^1 \rightarrow Q, Q_{(1/\perp)} = STOP$. An argument can be made that the events "$(^1$" and "$)^1$" are behaving like locals in Q_ε, hence, Q_ε can be represented by a finite state machine. However, in P_ε above the event "(" *must* be a call and the event ")" *must* be a return.

Parallel Composition The CMVP parallel composition is ${}_A\|_B$, with A and B being the respective event interfaces of the processes on either side of the parallel composition operator. Figure 6 shows the semantics of parallel composition. Since having an overlap in call and return stack alphabets of MvPDA violates the restriction of our process algebra, CMVP processes in a parallel composition can

$$\frac{P_\Gamma \xrightarrow{a} P'_\Gamma \qquad Q_{\Gamma'} \xrightarrow{a} Q'_{\Gamma'}}{(P_A \|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{a} (P'_A \|_B Q')_{\Gamma\cdot\Gamma'}} \; \left[a \in A_l \cap B_l\right]$$

$$\frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{\begin{array}{l}(P_A \|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{a} (P'_A \|_B Q)_{\Gamma\cdot\Gamma'} \\ (Q_B \|_A P)_{\Gamma'\cdot\Gamma} \xrightarrow{a} (Q_B \|_A P')_{\Gamma'\cdot\Gamma}\end{array}} \; \left[a \in A_l \setminus B_l\right]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{\begin{array}{l}(P_A \|_B Q)_{(\Gamma\cdot\Gamma')(i/\gamma)} \xrightarrow{a^i} (P'_A \|_B Q)_{(\Gamma\cdot\Gamma')(i/a\gamma)} \\ (Q_B \|_A P)_{(\Gamma'\cdot\Gamma)(n+i/\gamma)} \xrightarrow{a^{n+i}} (Q_B \|_A P')_{(\Gamma'\cdot\Gamma)(n+i/a\gamma)}\end{array}} \; \begin{bmatrix} a^i \in (A_c)_i \setminus B \\ \implies a^{n+i} \in (A_c)_{n+i} \setminus B \end{bmatrix}$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{\begin{array}{l}(P_A \|_B Q)_{(\Gamma\cdot\Gamma')(i/a\gamma)} \xrightarrow{a^i} (P'_A \|_B Q)_{(\Gamma\cdot\Gamma')(i/\gamma)} \\ (Q_B \|_A P)_{(\Gamma'\cdot\Gamma)(n+i/a\gamma)} \xrightarrow{a^{n+i}} (Q_B \|_A P')_{(\Gamma'\cdot\Gamma)(n+i/\gamma)}\end{array}} \; \begin{bmatrix} a^i \in (A_r)_i \setminus B \\ \implies a^{n+i} \in (A_r)_{n+i} \setminus B \end{bmatrix}$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{\begin{array}{l}(P_A \|_B Q)_{(\Gamma\cdot\Gamma')(i/\perp)} \xrightarrow{a^i} (P'_A \|_B Q_{(\Gamma\cdot\Gamma')(i/\perp)} \\ (Q_B \|_A P)_{(\Gamma'\cdot\Gamma)(n+i/\perp)} \xrightarrow{a^{n+i}} (Q_B \|_A P')_{(\Gamma'\cdot\Gamma)(n+i/\perp)}\end{array}} \; \begin{bmatrix} a^i \in (A_r)_i \setminus B \\ \implies a^{n+i} \in (A_r)_{n+i} \setminus B \end{bmatrix}$$

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{\begin{array}{l}(P_A \|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{\tau} (P'_A \|_B Q)_{\Gamma\cdot\Gamma'} \\ (Q_B \|_A P)_{\Gamma'\cdot\Gamma} \xrightarrow{\tau} (Q_B \|_A P')_{\Gamma'\cdot\Gamma}\end{array}}$$

$$\frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma \qquad Q_{\Gamma'} \xrightarrow{\checkmark} Q'_{\Gamma'}}{(P_A \|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{\checkmark} (P'_A \|_B Q')_{\Gamma\cdot\Gamma'}}$$

Figure 6: Alphabetized parallel

only synchronize over local events. To guarantee that there will be no stack overlap the CMVP parallel composition is implemented using disjoint operations. A stack renaming must therefore be applied to one of the two processes using the concept of *stack renaming* as introduced in Section 6. Thus the notation $\Gamma \cdot \Gamma'$ represent the concatenation of the two tuples Γ and Γ'' with Γ'' the result of a suitable $|\Gamma|$-stack renaming of Γ'.

Hiding The semantics for hiding is given in Figure 7. Hiding does not change the stack content of the set of stacks of a process because only local symbols can be hidden. If a hidden local event is executed the process is seen to perform an internal action τ.

Abstract The abstract operator hides all the sub-modules of a module. The motivation for this operator is the abstract path in CARET and NWTL [1, 2]. By executing the abstract operator sub-modules can be hidden from the environment, which in turn allows for the abstraction of call and return symbols. The local trace of a module can be defined using the abstract operator, allowing for the specification of the internal properties of recursive modules. The semantics of abstract is given in Figure 8. When a call symbol is executed, abstract pushes the corresponding stack symbol to the stack labeled with one of the two special markers: $\tilde{\cdot}$ denotes the internal call of the main module and $\bar{\cdot}$ denotes the internal call of a sub-module. If the top of the stack in operation contains any special marker then every local event will

$$\frac{P_\Gamma \xrightarrow{t} P'_\Gamma}{P_\Gamma \setminus B \xrightarrow{t} P'_\Gamma \setminus B} \left[t \in \{\tau, \checkmark\}\right] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma \setminus B \xrightarrow{\tau} P'_\Gamma \setminus B} \left[a \in A_l \cap B\right]$$

$$\frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma \setminus B \xrightarrow{a} P'_\Gamma \setminus B} \left[a \in A_l \setminus B\right] \qquad \frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{P_{\Gamma(i/\gamma)} \setminus B \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)} \setminus B} \left[a^i \in (A_c)_i\right]$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{P_{\Gamma(i/a\gamma)} \setminus B \xrightarrow{a^i} P'_{\Gamma(i/\gamma)} \setminus B} \left[a^i \in (A_r)_i\right] \qquad \frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{P_{\Gamma(i/\perp)} \setminus B \xrightarrow{a^i} P'_{\Gamma(i/\perp)} \setminus B} \left[a^i \in (A_r)_i\right]$$

Figure 7: Hiding

$$\frac{P_{\Gamma(i/b\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/ab\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/b\gamma)}} \xrightarrow{a^i} \overline{P'_{\Gamma(i/\tilde{a}b\gamma)}} \\ \overline{P_{\Gamma(i/\tilde{b}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\bar{a}\tilde{b}\gamma)}} \\ \overline{P_{\Gamma(i/\bar{b}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\bar{a}\bar{b}\gamma)}}\end{array}} [a^i \in (A_c)_i] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{b} P'_{\Gamma(i/a\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/a\gamma)}} \xrightarrow{b} \overline{P'_{\Gamma(i/a\gamma)}} \\ \overline{P_{\Gamma(i/\tilde{a}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\tilde{a}\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\bar{a}\gamma)}}\end{array}} [b \in (A_l)]$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/a\gamma)}} \xrightarrow{a^i} \overline{P'_{\Gamma(i/\gamma)}} \\ \overline{P_{\Gamma(i/\tilde{a}\gamma)}} \xrightarrow{a^i} \overline{P'_{\Gamma(i/\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\gamma)}}\end{array}} \left[a^i \in (A_r)_i\right] \qquad \frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{\overline{P_{\Gamma(i/\perp)}} \xrightarrow{a^i} \overline{P'_{\Gamma(i/\perp)}}} \left[a^i \in (A_r)_i\right]$$

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{\overline{P_\Gamma} \xrightarrow{\tau} \overline{P'_\Gamma}} \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{\overline{P_\Gamma} \xrightarrow{\checkmark} P'_\Gamma}$$

$$\frac{P_{\Gamma(k/\gamma)} \xrightarrow{a^k} P'_{\Gamma(k/a\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/a\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/a\gamma)}} \\ \overline{P_{\Gamma(i/\bar{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/a\gamma)}}\end{array}} [a^k \in (A_c)_k] \qquad \frac{P_{\Gamma(k/a\gamma)} \xrightarrow{a^k} P'_{\Gamma(k/\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/\gamma)}} \\ \overline{P_{\Gamma(i/\bar{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/\gamma)}}\end{array}} [a^k \in (A_r)_k]$$

Figure 8: Abstract

be hidden; calls and returns are pushed to/popped off the stack in operation but are otherwise hidden as well (except for top-level calls and returns in the module). If a return symbol is executed, and the top of the stack in operation is not marked, then the process goes out of abstraction.

Let B (with call b^i and return f^i), and C (with call d^i and return e^i) be two modules. The top-level process P calls B and B calls C: $P = a \to Q$, $Q = b^i \to R_{(i/b)}$, $R = c \to S$, $S = d^i \to T_{(i/d)}$, $T = c \to U$, $U_{(i/d)} = e^i \to V_{(i/\perp)}$, $V_{(i/b)} = f^i \to W_{(i/\perp)}$, and $W = c \to STOP$. The sub-modules of B can be hidden by using abstract: $\overline{P_{(\gamma_1,\ldots,\gamma_n)}} = a \to \overline{Q_{(\gamma_1,\ldots,\gamma_n)}} \to b^i \to \overline{R_{(\gamma_1,\ldots,\gamma_{i-1},\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to c \to \overline{S_{(\gamma_1,\ldots,\gamma_{i-1},\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} = d^i \to \overline{T_{(\gamma_1,\ldots,\gamma_{i-1},\bar{d}\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to c \to \overline{U_{(\gamma_1,\ldots,\gamma_{i-1},\bar{d}\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to e^i \to \overline{V_{(\gamma_1,\ldots,\gamma_{i-1},\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to f^i \to W_{(\gamma_1,\ldots,\gamma_{i-1},\perp,\gamma_{i+1},\ldots,\gamma_n)} \to c \to STOP$. We actually hide sub-module C in this particular example. A run of P without the abstract operator will be as follows: $P_{(\gamma_1,\ldots,\gamma_n)} = a \to Q_{(\gamma_1,\ldots,\gamma_n)} \to b^i \to R_{(\gamma_1,\ldots,\gamma_{i-1},b\perp,\gamma_{i+1},\ldots,\gamma_n)} \to c \to S_{(\gamma_1,\ldots,\gamma_{i-1},b\perp,\gamma_{i+1},\ldots,\gamma_n)} \to d^i \to T_{(\gamma_1,\ldots,\gamma_{i-1},db\perp,\gamma_{i+1},\ldots,\gamma_n)} \to c \to U_{(\gamma_1,\ldots,\gamma_{i-1},db\perp,\gamma_{i+1},\ldots,\gamma_n)} \to e^i \to V_{(\gamma_1,\ldots,\gamma_{i-1},b\perp,\gamma_{i+1},\ldots\gamma_n)} \to f^i \to W_{(\gamma_1,\ldots,\gamma_{i-1},\perp,\gamma_{i+1},\ldots,\gamma_{i+1}n)} \to c \to STOP$.

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{f(P)_{\Gamma(i/\gamma)} \xrightarrow{f(a)^i} f(P')_{\Gamma(i/f(a)\gamma)}} [a^i \in (A_c)_i] \qquad \frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{f(P)_\Gamma \xrightarrow{\tau} f(P')_\Gamma}$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{f(P)_{\Gamma(i/f(a)\gamma)} \xrightarrow{f(a)^i} f(P')_{\Gamma(i/\gamma)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{f(P)_\Gamma \xrightarrow{f(a)} f(P')_\Gamma} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{f(P)_{\Gamma(i/\perp)} \xrightarrow{f(a)^i} f(P')_{\Gamma(i/\perp)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{f(P)_\Gamma \xrightarrow{\checkmark} f(P')_\Gamma}$$

Figure 9: Forward renaming

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{f(a)^i} P'_{\Gamma(i/f(a)\gamma)}}{f^{-1}(P)_{\Gamma(i/\gamma)} \xrightarrow{a^i} f^{-1}(P')_{\Gamma(i/a\gamma)}} [a^i \in (A_c)_i] \qquad \frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{f^{-1}(P)_\Gamma \xrightarrow{\tau} f^{-1}(P')_\Gamma}$$

$$\frac{P_{\Gamma(i/f(a)\gamma)} \xrightarrow{f(a)^i} P'_{\Gamma(i/\gamma)}}{f^{-1}(P)_{\Gamma(i/a\gamma)} \xrightarrow{a^i} f^{-1}(P')_{\Gamma(i/\gamma)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{f(a)} P'_\Gamma}{f^{-1}(P)_\Gamma \xrightarrow{a} f^{-1}(P')_\Gamma} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{f(a)^i} P'_{\Gamma(i/\perp)}}{f^{-1}(P)_{\Gamma(i/\perp)} \xrightarrow{a^i} f^{-1}(P')_{\Gamma(i/\perp)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{f^{-1}(P)_\Gamma \xrightarrow{\checkmark} f^{-1}(P')_\Gamma}$$

Figure 10: Backward renaming

Renaming The semantics of forward and backward renaming are given in Figures 9 and 10, respectively. Renaming can change the matched calls and returns of a CMVP process, however it cannot modify the MVPL partition. For instance, if we have a call event a^i with a matching return event b^i, if a^i is renamed $f(a)^i$, then b^i will be the matching return event for $f(a)^i$. There might be no "reverse" renaming that retrieves the original process or set of matched call-returns: Suppose that a renaming f is applied to the return event $\rangle$ in process P_ε in our previous example illustrating choice such that $f(\rangle) =]$. We then get a process whose traces define the language $[^n]^n$; no renaming can give back the original.

Sequential Composition and Interrupt Using sequential composition or interrupt the control of execution can be passed from one CMVP process to a second, either because the first process reaches a

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{P_{\Gamma(i/\gamma)}; Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}; Q_{\Gamma'}} [a^i \in (A_c)_i] \qquad \frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{P_\Gamma; Q_{\Gamma'} \xrightarrow{\tau} P'_\Gamma; Q_{\Gamma'}}$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{P_{\Gamma(i/a\gamma)}; Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}; Q_{\Gamma'}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma; Q_{\Gamma'} \xrightarrow{a} P'_\Gamma; Q_{\Gamma'}} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{P_{\Gamma(i/\perp)}; Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}; Q_{\Gamma'}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{P_\Gamma; Q_{\Gamma'} \xrightarrow{\tau} Q_{\Gamma'}}$$

Figure 11: Sequential composition

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{\tau} P'_\Gamma \triangle Q_{\Gamma'}} \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{a} P'_\Gamma \triangle Q_{\Gamma'}} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{P_{\Gamma(i/\gamma)} \triangle Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)} \triangle Q_{\Gamma'}} [a \in (A_c)_i] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{P_{\Gamma(i/a\gamma)} \triangle Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)} \triangle Q_{\Gamma'}} [a \in (A_r)_i]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{P_{\Gamma(i/\perp)} \triangle Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\perp)} \triangle Q_{\Gamma'}} [a \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{\checkmark} P'_\Gamma}$$

$$\frac{Q_{\Gamma'} \xrightarrow{\tau} Q'_{\Gamma'}}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{\tau} Q'_{\Gamma'}} \qquad \frac{Q_{\Gamma'(j/\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/a\gamma)}}{P_\Gamma \triangle Q_{\Gamma'(j/\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/a\gamma)}} [a \in (A_c)_j]$$

$$\frac{Q_{\Gamma'} \xrightarrow{a} Q'_{\Gamma'}}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{a} Q'_{\Gamma'}} [a \in (A_l)] \qquad \frac{Q_{\Gamma'(j/a\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/\gamma)}}{P_\Gamma \triangle Q_{\Gamma'(j/a\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/\gamma)}} [a \in (A_r)_j]$$

$$\frac{Q_{\Gamma'(j/\perp)} \xrightarrow{a^j} Q'_{\Gamma'(j/\perp)}}{P_\Gamma \triangle Q_{\Gamma'(j/\perp)} \xrightarrow{a^j} Q'_{\Gamma'(j/\perp)}} [a \in (A_r)_j]$$

Figure 12: Interrupt

particular point in its execution (termination), or because the second process demands it (in the case of interrupt). Figures 11 and 12 show the semantics of sequential composition and interrupt, respectively.

Unlike parallel composition, neither sequential composition nor interrupt use stack renaming explicitly. The set of stacks of processes in a sequential composition or interrupt will never overlap, since the first process can no longer perform actions once the control is passed onto the second process. So once the control is passed on, the set of stacks of the first process is no longer relevant (and can be discarded).

7.2 CMVP Is a Process Algebra

Theorem 8. *CMVP is an algebra; that is, CMVP is closed under all its operators. The underlying semantics of any CMVP process is a MvPDA (or an equivalent LTS).*

Proof. We proceed by structural induction. *STOP*, *SKIP* are defined as CMVP processes similarly to their definition in CSP. Also, CMVP's closure under prefix choice, external choice, internal choice, recursion, renaming, hiding and abstract are all immediate. Prefix choice follows the definition of a transition in the associated MvPDA. External choice is simply a prefix choice with more than one alternative (i.e. this allows more than one transition out of one state). The internal choice construct is connection between two LTS to a common start state via τ transitions (this does not change the stack contents of the set of stacks). Recursion is simply a loop from some MvPDA state P back into the same state P. The set of stacks of the MvPDA are manipulated according to the MvPDA semantics introduced by the other transitions and this does not introduce infinite MvPDA states. Renaming cannot changed the MVPL partition. Closure under hiding is straightforward since only local symbols can be hidden; τ transitions replace hidden local symbols while a change in state occurs in the LTS. When an abstract operation is applied on an MvPDA, the MvPDA transitions are hidden (that is, replaced with τ) after the first stack symbol is pushed into the stack in operation. The MvPDA comes out of the abstracted state after the

first stack symbol which was pushed to the stack in operation is popped. Hence, the abstract construct replaces whole portions of the LTS with τ transitions.

Let now P and Q be two CMVP processes whose semantics are given by the MvPDA L' and L'', respectively. Let L' and L'' have initial states I' and I'' and final states H' and H'', respectively.

Consider now the sequential composition of P and Q. An MvPDA L corresponding to this sequential composition can be constructed as follows: In a sequential composition L' will run to completion, followed by a run of L'' (which is only possible when L' has reached termination, case in which there is no further computation relying on the set of stacks of L'). We then take the disjoint union of the stacks of L' and L'' and call the results the set of stacks of L. The transitions of L will then be the union of the transition relations of L' and L'' (the latter suitably modified by the disjoint union operation), plus ε-transitions from all the states in H' to I''. The initial state of L is I' and the final states are exactly all the states from H''. That the resulting MvPDA is the semantic model of the sequential composition is immediate: The automaton L' runs until it reaches its final state. Once this happens, the control is given to the state I'', in effect launching L''. The disjoint union ensures that the stacks of L' are never used from this point on, and that the stacks of L'' are empty when the control is given to this automaton (since no transitions from L' operates on them), as desired.

Closure under interrupt is shown by a similar construction, with the exception that the control can be passed from P to Q or equivalently from L' to L'' at any time, hence in addition to the union of the transitions of L' and L'' (as for the sequential composition) we add one ε-transition from every state of L' to I'' (instead of ε-transition from the states in H' to I'' only). As above, once the control passes to L'' the stacks of L' are not longer needed, and at that moment the stacks of L'' are all empty, as desired.

The construction of the parallel composition L of L' and L'' is given as follows: the Cartesian product of the MvPDA states in L' and L'' is the set of MvPDA states in L, with the initial state of L' and the final states of L'' being the initial and final states of L, respectively. The set of stacks of L will be the disjoint union of the set of stacks of L' and L''. Recall that CMVP parallel composition makes use of stack renaming which implements such a disjoint union. Hence we are guaranteed that there will be no stack overlap between L' and L'' and so the restriction on CMVP processes is not violated. The transition relation of L is defined as follows:

- For synchronized local symbols l we have the transition $(P',\varepsilon)\xrightarrow{l}(Q',\varepsilon)$ in L' and the transition $(P'',\varepsilon)\xrightarrow{l}(Q'',\varepsilon)$ in L'' by the inductive hypothesis. In this case we add to L the transition $((P',P''),\varepsilon)\xrightarrow{l}((Q',Q''),\varepsilon)$. For unsynchronized local symbols that have the transition $(P',\varepsilon)\xrightarrow{l}(Q',\varepsilon)$ in L' [the transition $(P'',\varepsilon)\xrightarrow{l}(Q'',\varepsilon)$ in L''] we add to L the transitions $((P',X),\varepsilon)\xrightarrow{l}((Q',X),\varepsilon)$ for all the states X of L'' [$((X,P''),\varepsilon)\xrightarrow{l}((X,Q''),\varepsilon)$ for all the states X of L']. The correctness of this construction is immediate from the fact that there is no change in the content of any stack for local symbols, while the transitions above ensures that the two components of the state of L change according to the semantics of L' and L'', respectively.

- All the calls c^i of L' and c^{n+i} of L'' (after the stack renaming) and return symbols r^i in L' and r^{n+i} in L'' (again after the stack renaming) are unsynchronized. These are integrated in L as follows:

  For every set of rules $(P',\varepsilon)\xrightarrow{c^i}(Q',a)$, $(P'',\varepsilon)\xrightarrow{c^{n+i}}(Q'',b)$, $(R',a)\xrightarrow{r^i}(S',\varepsilon)$, and $(R'',b)\xrightarrow{r^{n+i}}(S'',\varepsilon)$ we add the following rules, respectively: $((P',X),\varepsilon)\xrightarrow{c^i}((Q',X),a)$, $((Y,P''),\varepsilon)\xrightarrow{c^{n+i}}((Y,Q''),b)$, $((R',X),a)\xrightarrow{r^i}((S',X),\varepsilon)$, and $((Y,R''),b)\xrightarrow{r^{n+i}}((Y,S''),\varepsilon)$ for all states X of L'' and Y of L'. Once more the correctness of the construction follows from the fact that the sets of stacks of L' and L'' do not overlap now that the stacks of L'' have been renamed.

- No other transitions are included in the transition relation of L.

□

8 CMVP Trace Semantics

STOP and *SKIP* are the same in CSP and CMVP. *RUN* however cannot be defined as a CMVP process that can be combined with other processes (and thus be useful in establishing laws and other properties), since having an overlap in call and return stack alphabets of MvPDA violates the restriction of our process algebra. So instead, we define a CMVP process (RUN_{Σ_l}) equivalent to the CSP process *RUN* in CMVP.

RUN_{Σ_l} is the process that can always perform any local event and cannot execute any call or return action and so its stacks is always empty. We have:

$$\text{traces}(RUN_{\Sigma_l}) = \{tr | tr \in TRACE \wedge \sigma(tr) \subseteq \Sigma_l\}$$

Prefix Choice There are only two possibilities when observing the process $((x : A \to P(x))_{\Gamma(i/\gamma)}$: either no event has executed, or an event $a \in A$ has executed, making its subsequent behaviour that of the corresponding process $P(a)_{\Gamma(i/\gamma')}$. If $a \in A_l$ then $\gamma' = \gamma$:

$$\text{traces}((x : A \to P(x))_\Gamma) = \{\langle\rangle\} \cup \{\langle a\rangle.tr | a \in A_l \wedge tr \in \text{traces}(P(a))_\Gamma\}$$

If $a \in (A_c)_i$ then $a = a^i$ and $\gamma' = a\gamma$:

$$\text{traces}((x : A \to P(x))_{\Gamma(i/\gamma)}) = \{\langle\rangle\} \cup \{\langle a\rangle.tr | a \in A_c \wedge tr \in \text{traces}(P(a)_{\Gamma(i/a\gamma)})\}$$

If $a \in (A_r)_i$ and $\gamma = a\delta$ then $a = a^i$ and $\gamma' = \delta$:

$$\text{traces}((x : A \to P(x))_{\Gamma(i/a\delta)}) = \{\langle\rangle\} \cup \{\langle a\rangle.tr | a \in A_r \wedge tr \in \text{traces}(P(a)_{\Gamma(i/\delta)})\}$$

If $a \in (A_r)_i$ and $\gamma = \perp$ then then $a = a^i$ and $\gamma' = \gamma$:

$$\text{traces}((x : A \to P(x))_{\Gamma(i/\perp)}) = \{\langle\rangle\} \cup \{\langle a\rangle.tr | a \in A_r \wedge tr \in \text{traces}(P(a)_{\Gamma(i/\perp)})\}$$

We note that $(x : A \to P(x))_{\Gamma(i/\gamma)} \xrightarrow{a \in A} P(a)_{\Gamma(i/\gamma')}$, meaning that after the event $x \in A$ is executed the stack in operation will be γ', where γ' will depend on the type of x (call, return, local) in the usual manner described earlier. Hence the above four rules can be written as the single rule as follows:

$$\begin{aligned}\text{traces}((x : A \to P(x))_{\Gamma(i/\gamma)}) \quad = \quad \{\langle\rangle\} \quad \cup \quad \{\langle a\rangle.tr | a \in A \wedge \\ (x : A \to P(x))_{\Gamma(i/\gamma)} \xrightarrow{a \in A} P(a)_{\Gamma(i/\gamma')} \\ \wedge tr \in \text{traces}(P(a)_{\Gamma(i/\gamma')})\}\end{aligned}$$

External Choice A choice will split a process into alternative processes, allowing the parent to choose once between them. The alternative processes operate on the same set of stacks as the parent process.

$$\text{traces}(P_\Gamma \,\Box\, Q_{\Gamma'}) = \text{traces}(P_\Gamma) \cup \text{traces}(Q_{\Gamma'})$$

Figure 13 enumerates the laws of external choice. The first three laws are derived from the properties of disjoint union. Law $\Box - unit$ states that any process P_Γ has precedence over *STOP* in a choice. In algebraic terms, *STOP* is a unit of external choice.

Internal Choice Similarly with external choice the traces of internal choice are as follows:

$$\text{traces}(P_\Gamma \sqcap Q_{\Gamma'}) = \text{traces}(P_\Gamma) \cup \text{traces}(Q_{\Gamma'})$$

Unlike in the external choice, the environment plays no part in choosing one process over another in internal choice, and therefore the two operators may have different executions in resolving a choice. However, the only concern of a trace observer is identifying the sequence of possible actions of the choice outcomes, which are the same for the two operators. The single law for internal choice is thus given in Figure 14.

$$P_\Gamma \,\square\, P_\Gamma = P_\Gamma \qquad \square - idem$$

$$P_\Gamma \,\square\, (Q_{\Gamma'} \,\square\, R_{\Gamma''}) = (P_\Gamma \,\square\, Q_{\Gamma'}) \,\square\, R_{\Gamma''} \qquad \square - assoc$$

$$P_\Gamma \,\square\, Q_{\Gamma'} = Q_{\Gamma'} \,\square\, P_\Gamma \qquad \square - sym$$

$$P_\Gamma \,\square\, STOP = P_\Gamma \qquad \square - unit$$

Figure 13: Laws for external choice

$$P_\Gamma \,\square\, Q_{\Gamma'} = P_\Gamma \sqcap Q_{\Gamma'} \qquad choice - equiv$$

Figure 14: Law for internal choice

Parallel Composition Any execution of $(P_A \|_B Q)_{\Gamma.\Gamma'}$ projected on interface A $[B]$ must be an event that P_Γ $[Q_{\Gamma'}]$ can execute. Therefore the traces of $(P_A \|_B Q)_{\Gamma.\Gamma'}$ are those sequences of events that P_Γ and $Q_{\Gamma'}$ can execute which are in interface A and interface B, and also termination. Hence, the set of events in the trace $(\sigma(tr))$, must be contained in $(A \cup B)^\checkmark$.

$$\begin{aligned}\text{traces}(P_A \|_B Q)_{\Gamma.\Gamma'} = \{ tr \in TRACE \quad | \quad & tr \upharpoonright A^\checkmark \in \text{traces}(P_\Gamma) \wedge tr \upharpoonright B^\checkmark \in \text{traces}(Q_{\Gamma'}) \\ & \wedge \sigma(tr) \subseteq (A \cup B)^\checkmark \}\end{aligned}$$

Figure 15 enumerates the laws of alphabetized parallel. In Law $\| - step$ and Law $\| - term\ 2$ γ and γ' represent the stack content of stack i (and $n+i$) before and after an execution (or transition): if a local or an empty return event is executed then $\gamma' = \gamma$, else if a call or an non-empty return event is executed them $\gamma' \neq \gamma$. Law $\| - assoc$ is a form of association derived from the disjoint union operator.

$$(P_A \|_{B \cup C} (Q_B \|_C R))_{\Gamma.\Gamma'.\Gamma''} = ((P_A \|_B Q)_{A \cup B} \|_C R)_{\Gamma.\Gamma'.\Gamma''} \qquad \| - assoc$$

$$(P_A \|_B Q)_{\Gamma.\Gamma'} = (Q_B \|_A P)_{\Gamma'.\Gamma} \qquad \| - sym$$

$$\begin{aligned}& C \subseteq A \wedge D \subseteq B \Rightarrow ((x : C \to P(x)_A \|_B (y : D \to Q(y))_{\Gamma.\Gamma'} \\ & = z : ((C \setminus B) \cup (D \setminus A) \cup (C \cap D)) \to R(z)_{(\Gamma(1,\ldots,n)'.(\Gamma'(n+1,\ldots,x)')} \\ & \text{where } R(z)_{(\Gamma(1,\ldots,n)') \cup (\Gamma'(n+1,\ldots,x)')} \\ & \qquad = (P(c)_A \|_B (y : D \to Q(y))_{\Gamma.\Gamma'(i/\gamma')}) \qquad \text{if} \quad c \in C \setminus B \\ & \qquad = ((x : C \to P(x))_A \|_B Q(c))_{\Gamma.\Gamma'(n+i/\gamma')} \qquad \text{if} \quad c \in D \setminus A \\ & \qquad = (P(c)_A \|_B Q(c))_{\Gamma.\Gamma'} \qquad \text{if} \quad c \in C_l \cap D_l \qquad \| - step\end{aligned}$$

$$SKIP_A \|_B SKIP = SKIP \qquad \| - term\ 1$$

$$\begin{aligned}& ((x : C \to P(x))_A \|_B SKIP)_{\Gamma(i/\gamma)} = ((x : C \to P(x))_A \|_B SKIP)_{\Gamma(i/\gamma')} \\ & = x : C \cap (A \setminus B) \to (P(x)_A \|_B SKIP)_{\Gamma(i/\gamma')} \qquad \| - term\ 2\end{aligned}$$

$$(P_A \|_A (RUN_{\Sigma_l}))_\Gamma = P_\Gamma \qquad \| - unit$$

Figure 15: Laws for alphabetized parallel

$$(P_\Gamma \setminus A) \setminus B = P_\Gamma \setminus (A \cup B) \qquad hide - combine$$

$$STOP \setminus A = STOP \qquad hide - STOP$$

$$(x : C \to P(x))_{\Gamma(i/\gamma)} \setminus A = (x : C \to P(x)_{\Gamma(i/\gamma')}) \setminus A$$
$$= (x : C \to (P(x)_{\Gamma(i/\gamma')} \setminus A)) \quad \text{if} \quad A \cap C = \emptyset \qquad hide - step\ 1$$

$$(x : C \to P(x))_{\Gamma(i/\gamma)} \setminus A = (x : C \to P(x)_{\Gamma(i/\gamma')}) \setminus A$$
$$= \sqcap_{x \in C}(P(x)_{\Gamma(i/\gamma')} \setminus A) \quad \text{if} \quad C \subseteq A \qquad hide - step\ 2$$

$$SKIP \setminus A = SKIP \qquad hide - term$$

Figure 16: Laws for hiding

The outcome of a parallel composition is independent of the order in which components of the parallel composition construct are composed according to Law $\| - assoc$. Law $\| - unit$ describes a unit for parallel composition namely, RUN_{Σ_l} (which is always ready to execute any local event in the common interface, hence imposes no restriction on P_Γ). Law $\| - step$ breaks down parallel composition into prefix choice (i.e. all the possible actions that a parallel composition can execute). It also shows that CMVP processes can only have synchronization over local events, and explains how the set of stacks in the parallel composition construct are manipulated. Laws $\| - term\ 1$ and $\| - term\ 2$ explain the termination of a parallel composition construct. Termination occurs only when both processes are ready to terminate.

Synchronization only occurs over local events, so parallel composition does not have a zero in CMVP (as opposed to *STOP* being a zero for parallel composition in CSP). Indeed, a CMVP process P can still execute interleaving call and return actions independent on *STOP*, and so the parallel composition between P and *STOP* is not equivalent to *STOP* anymore. Consider for instance $P = a^1 \to P_{(1/a)}$, $P_{(1/a)} = b^1 \to P_{(1/\perp)}$, and $P_{(1/\perp)} = STOP$. We have $a^1 b^1 \in \text{traces}(P_\varepsilon \| STOP)$ and so $\text{traces}(P_\varepsilon \| STOP) \neq \emptyset$. In fact in this example $(P_\varepsilon \| STOP) = P_\varepsilon$!

Hiding Hiding A replaces exactly all the actions from A with internal actions, which are ignored in trace semantics. Therefore:

$$\text{traces}(P_\Gamma \setminus A) = \{tr \setminus A | tr \in \text{traces}(P_\Gamma)\}$$

Figure 16 enumerates the laws of hiding. Law $hide - combine$ states that hiding interfaces succession and hiding them at the same time have the same outcome. Laws $hide - step\ 1$ and $hide - step\ 2$ describe the use of hiding over a prefix choice.

Renaming The traces of $f(P)_\Gamma$ are traces of P_Γ with every event mapped through f:

$$\text{traces}(f(P_\Gamma)) = \{f(tr) | tr \in \text{traces}(P_\Gamma)\}$$

Conversely, any trace tr of a backward renamed process $f^{-1}(P_\Gamma)$ will result in a trace of P_Γ when mapped through the function f:

$$\text{traces}(f^{-1}(P_\Gamma)) = \{tr | f(tr) \in \text{traces}(P_\Gamma)\}$$

Figure 17 enumerates the laws of renaming. Law $f(.) - step\ 1$ states that if the mapping f is one-to-one then a choice of events from interface C translates to a choice of events from $f(C) = \{f(c) | c \in C\}$.

$$f(x:C \to P(x))_{\Gamma(i/\gamma)} = f(x:C \to P(x)_{\Gamma(i/\gamma')}) = \\ y:f(C) \to f(P(f^{-1}(y)))_{\Gamma(i/\gamma')} \qquad \text{if } f \text{ is 1-1} \qquad f(.)-step\ 1$$

$$f(x:C \to P(x))_{\Gamma(i/\gamma)} = f(x:C \to P(x)_{\Gamma(i/\gamma')}) \\ = y:f(C) \to \sqcap_{x|f(x)=y} f(P(x))_{\Gamma(i/\gamma')} \qquad f(.)-step\ 2$$

$$f(SKIP) = SKIP \qquad f(.)-term$$

$$f^{-1}(x:C \to P(x))_{\Gamma(i/\gamma)} = f^{-1}(x:C \to P(x)_{\Gamma(i/\gamma')}) \\ = y:f^{-1}(C) \to f^{-1}(P(f(y)))_{\Gamma(i/\gamma')} \qquad f^{-1}(.)-step$$

$$f^{-1}(SKIP) = SKIP \qquad f^{-1}(.)-term$$

Figure 17: Laws for Renaming

$$P_\Gamma;(Q_{\Gamma'};R_{\Gamma''}) = (P_\Gamma;Q_{\Gamma'});R_{\Gamma''} \qquad ;-assoc$$

$$SKIP;P_\Gamma = P_\Gamma \qquad ;-unit-l$$

$$P_\Gamma;SKIP = P_\Gamma \qquad ;-unit-r$$

$$(x:C \to P(x))_{\Gamma(i/\gamma)};Q_{\Gamma'} = (x:C \to P(x)_{\Gamma(i/\gamma')});Q_{\Gamma'} \\ = x:C \to (P(x)_{\Gamma(i/\gamma')};Q_{\Gamma'}) \qquad ;-step$$

$$STOP;P_\Gamma = STOP \qquad ;-zero-l$$

Figure 18: Laws for sequential composition

Any choice event y will be mapped to exactly one event $x(=f^{-1}(y))$ from the original choice of events from interface C, so the subsequent actions are that of $P(x)_\Gamma$ transformed through f. Law $f(.)-step\ 2$ states that if a process is initially ready to execute any event from an interface C, then the initial choice for its renamed process is the set of events in interface $f(C)$. Unlike law $f(.)-step\ 1$, the mapping of function f is not necessarily one-to-one, so choosing an event y in the original interface may result in the execution of any of the processes which follow $f(y)$ in the renamed process.

Sequential Composition The traces of a sequential composition construct $P_\Gamma;Q_{\Gamma'}$ can be broken down into two parts: the traces of P_Γ, and the traces of $Q_{\Gamma'}$:

$$\text{traces}(P_\Gamma;Q_{\Gamma'}) = \{ \ tr|tr \in \text{traces}(P_\Gamma) \wedge \checkmark \notin \sigma(tr)\} \cup \{tr_1.tr_2|tr_1\langle\checkmark\rangle \\ \in \text{traces}(P_\Gamma) \wedge tr_2 \in \text{traces}(Q_{\Gamma'})\}$$

Sequential composition satisfies all the laws stated in Figure 18. Law $;-assoc$ which is derived from the disjoint union operator states that a sequential composition construct is associative. The $;-unit$ laws provide a unit for sequential composition (namely, *SKIP*). Law $;-step$ states that a prefix choice in a sequential composition is the same as a prefix choice of the sequentially composed processes. Law $;-zero-l$ establishes *STOP* as a left zero (*STOP* does not offer any action but does not terminate either, so no further action is possible).

$$P_\Gamma \triangle (Q_{\Gamma'} \triangle R_{\Gamma''}) = (P_\Gamma \triangle Q_{\Gamma'}) \triangle R_{\Gamma''} \qquad \triangle - assoc$$

$$STOP \triangle P_\Gamma = P_\Gamma \qquad \triangle - unit - l$$

$$P_\Gamma \triangle STOP = P_\Gamma \qquad \triangle - unit - r$$

$$(x : C \to P(x))_{\Gamma(i/\gamma)} \triangle Q_{\Gamma'} = (x : C \to P(x)_{\Gamma(i/\gamma')}) \triangle Q_{\Gamma'}$$
$$= Q_{\Gamma'} \,\Box\, (x : C \to (P(x)_{\Gamma(i/\gamma')} \triangle Q_{\Gamma'})) \qquad \triangle - step$$

$$SKIP \triangle P_\Gamma = SKIP \,\Box\, P_\Gamma \qquad \triangle - term$$

Figure 19: Laws for interrupt

Interrupt The traces of $P_\Gamma \triangle Q_{\Gamma'}$ can be broken down into two parts: the traces of P_Γ or else the not necessarily terminating traces of P_Γ concatenated with the traces of $Q_{\Gamma'}$:

$$\text{traces}(P_\Gamma \triangle Q_{\Gamma'}) = \text{traces}(P_\Gamma) \cup \{tr_1.tr_2 | tr_1 \in \text{traces}(P_\Gamma) \wedge \checkmark \notin \sigma(tr_1) \wedge tr_2 \in \text{traces}(Q_{\Gamma'})\}$$

There are a number of laws appropriate for the interrupt construct as given in Figure 19 and governing its interaction with choice construct termination. Law $\triangle - assoc$ which is derived from the disjoint union operator states that the interrupt construct is associative. Law $\triangle - step$ gives a description of how a prefix choice interrupted by a process $Q_{\Gamma'}$ behaves: it either acts as $Q_{\Gamma'}$ instantly, or it executes an event in the prefix choice. The $\triangle - unit$ laws provide a unit (*STOP*) for the interrupt construct.

Recursion The relation that defines a CMVP recursive process remains $P = F(P)$ (just like the CSP recursion). however, a CMVP recursive process defines a loop from one *MvPDA state* back to the same MvPDA state, with the stack content possibly varying from one occurrence of p to the other. We therefore consider the recursive definition $P = F(P)$ within its proper place as a CMVP process i.e., $(P = F(P))_\Gamma$, or equivalently $P_\Gamma = F(P)_\Gamma$. That is, the MvPDA state P with the set of stacks Γ behaves the same as the MvPDA state $F(P)$ with the same set of stacks and so $\text{traces}(P_\Gamma) = \text{traces}(F(P)_\Gamma)$. That is, $\text{traces}(P_\Gamma)$ becomes a *fixed point* of the function on trace sets represented by the CMVP expression F.

CMVP is monotonic in respect to $\subseteq$, meaning that $\text{traces}(P_\Gamma) \subseteq \text{traces}(Q_\Gamma)$ implies $\text{traces}(F(P)_\Gamma) \subseteq \text{traces}(F(Q)_\Gamma)$ for any function F constructed out of CMVP operators and terms. On the other hand every CMVP process has the empty trace as one of its possible traces, hence $\langle\rangle \in \text{traces}(P_\Gamma)$. Then $\text{traces}(STOP_\Gamma) \subseteq \text{traces}(P_\Gamma)$ which implies $\text{traces}(F(STOP)_\Gamma) \subseteq \text{traces}(F(P)_\Gamma) = \text{traces}(P_\Gamma)$. An inductive argument will then show that $\text{traces}(F^n(STOP)_\Gamma) \subseteq \text{traces}(F(P)_\Gamma) = \text{traces}(P_\Gamma)$ for all $n \geq 0$. In other words, the traces obtained by unwinding the definition $(P = F(P))_\Gamma$ an arbitrary number of times are still traces of P_Γ. All of the $F^n(STOP)_\Gamma$ processes amounts to the finite unwinding of the recursive definition, so between them they account for all the possible traces of $(P = F(P))_\Gamma$:

$$\text{traces}((P = F(P))_\Gamma) = \bigcup_{n \in \mathbb{N}} \text{traces}(F^n(STOP)_\Gamma)$$

.

Abstract Abstract extracts the internal trace of the first module from a CMVP process and then follows the rest of the original traces. Only sub-modules that return to their parent can be processed this way. We therefore have:

$$\text{traces}(\overline{P_\Gamma}) = \{\mathfrak{A}(tr) | tr \in \text{traces}(P_\Gamma)\}$$

where $\mathfrak{A}(tr)$ is a function which extracts the trace tr' where tr' is the trace of $\overline{P_\Gamma}$, and tr is the trace of P_Γ. A definition for $\mathfrak{A}$ is given in Section 9.1.

9 Trace Specification and Verification in CMVP

We assume that the visible partitions of events are always known by a CMVP trace observer, so that a CMVP trace observer can recognize when a system (MvPDA) executes a call or return event. This assumption enables the definition of four crucial functions, which enable the specification of certain properties that are useful in the verification process: The abstract function is used to specify local properties of a module in a system. Stack limits, access control, and concurrent stack properties are all specified using the stack extract function. The module extract function enables the specification of properties that are specific to one module of a system, while the completeness function is used to specify partial and total correctness.

9.1 CMVP Trace Functions

The *abstract function* $\mathfrak{A}(tr)$ obtains the trace tr' of $\overline{P_\Gamma}$. Recall that for every trace tr' of $\overline{P_\Gamma}$ there is a trace tr of P_Γ. We then have $\mathfrak{A}(tr) = \{l_0.c_1{}^i.r_1{}^i.l_1.c_2{}^i.r_2{}^i.l_2 \ldots c_k{}^i.r_k{}^i.l_k.w | \forall i : 1 \leq i \leq n \wedge tr = l_0.t_1.t_2 \ldots t_k.w \wedge l_0 \in \Sigma_l^* \wedge \forall x : 1 \leq x \leq k \wedge t_x \in \{c_x{}^i.s_x.r_x{}^i.l_x, \langle\rangle\} \wedge (s_x = \langle\rangle \vee \forall s' < s_x : (s' = \langle\rangle \vee |s'|_{\widetilde{\Sigma}_c^i} \geq |s'|_{(A_r)_i}) \wedge |s_x|_{\widetilde{\Sigma}_c^i} = |s_x|_{\widetilde{\Sigma}_r^i}) \wedge c_x{}^i \in \widetilde{\Sigma}_c^i \wedge r_x{}^i \in \widetilde{\Sigma}_r^i \wedge l_x \in \Sigma_l^* \wedge (w = \langle\rangle \vee head(w) \in \widetilde{\Sigma}_r^{\checkmark} \vee \forall w' < w : (w' = \langle\rangle \vee |w'|_{\widetilde{\Sigma}_c^i} \geq |w'|_{\widetilde{\Sigma}_r^i}))\}$.

In a CMVP trace the number of call events on a stack i in the set of stacks Γ is equal to the number of stack symbols pushed onto the stack i, while the number of balanced return events in i is equal to the number of stack symbols popped off the stack i. Based on this property we define the *stack extract* function $\mathfrak{S}_i(tr)$ which will extract the content of stack i from the CMVP trace tr: Given a process $P_{(\gamma_1,\ldots,\gamma_i,\ldots,\gamma_n)}$ and with $\mathfrak{R}(tr)$ denoting the reversal of the trace tr, $\mathfrak{S}_i(tr) = \{c_{i+j}{}^i.c_{i+j-1}{}^i \ldots c_{i+2}{}^i.c_{i+1}{}^i.\bot | tr' = tr \setminus \widetilde{\Sigma}_l^{\checkmark} \wedge sq = \mathfrak{R}(tr').\bot \wedge sq = s_{i+j+1}.c_{i+j}{}^i.s_{i+j}.c_{i+j-1}{}^i \ldots s_{i+2}.c_{i+1}{}^i.s_{i+1}.r_i{}^i.s_i.r_{i-1}{}^i \ldots s_3.r_2{}^i.s_2.r_1{}^i.s_1.\bot \wedge \forall x : 1 \leq x \leq i \wedge r_x{}^i \in \{\Sigma_r^i \cup \langle\rangle\} \wedge \forall y : 1 \leq y \leq j \wedge c_{i+y}{}^i \in \{\Sigma_c^i \cup \langle\rangle\} \wedge \forall z : 1 \leq z \leq i+j+1 \wedge s_z \in S^* \wedge S = \{s | \forall s' < s : (s' = \langle\rangle \vee |s'|_{\Sigma_r^i} \geq |s'|_{\Sigma_c^i}) \wedge |s|_{\Sigma_r^i} = |s|_{\Sigma_c^i}\}\}$.

The *module extract* function $\mathfrak{M}(tr, a^i)$ extracts from the trace tr the trace tr' of the module which starts with the call event a^i: $\mathfrak{M}(tr) = \{tr' | tr = tr'.tr'' \wedge \forall t < tr' : (t = \langle\rangle \vee |t|_{\Sigma_c^1} \geq |t|_{\Sigma_r^1}) \wedge (tr'' = \langle\rangle \vee tr'' = \langle\checkmark\rangle \vee (head(tr'') \in \widetilde{\Sigma}_r \wedge |tr'|_{\widetilde{\Sigma}_c} = |tr'|_{\widetilde{\Sigma}_r}))\}$ and $\mathfrak{M}(tr, a^i) = \{tr' | tr = tr''.a^i.tr''' \wedge |tr''|_{a^i} = 0 \wedge tr' = \mathfrak{M}(tr''')\}$.

Completeness $\mathfrak{C}(tr, a^i)$ verifies that a trace tr contains the complete trace of a sub-module. If a sub-module is invoked by the call event a^i, then tr must include the call event a^i and its balanced return: $\mathfrak{C}(tr, a^i) = \{tr' | (tr = tr''.tr'.tr''' \wedge head(tr') = a^i \wedge foot(tr') \in \widetilde{\Sigma}_r^i \wedge t < tr' : (t = \langle\rangle \vee |t|_{\widetilde{\Sigma}_c^i} \geq |t|_{\widetilde{\Sigma}_r^i}) \wedge |tr'|_{\widetilde{\Sigma}_c^i} = |tr'|_{\widetilde{\Sigma}_r^i} \wedge |tr''|_{a^i} = 0) \vee tr' = \langle\rangle\}$. If the return balancing the call a^i is not included in the trace then the function will return the empty trace.

9.2 CMVP Trace Specification

CMVP is a generalization of CSP and so it behaves exactly like CSP when all the executing events are locals. Hence, CMVP can specify any property of a system that CSP can. CMVP can also specifies various crucial properties for software verification that regular or a context-free process algebra are unable to specify, as follows:

CMVP can specify the *access control* properties of a module. It can specify that a module can be invoked if a certain property holds. For instance, a specification might require that a procedure A (started

by a call event a^i) can be invoked only if another procedure B (started by a call event b^i) is on the stack i in the set of stacks Γ. In CMVP, this property is expressed as $S(tr) = |\mathfrak{S}_i(tr)|_{b^i} \neq 0 \Rightarrow |\mathfrak{S}_i(tr)|_{a^i} \neq 0$.

CMVP can specify *stack limits* that is, that a property holds whenever the size of the i-th stack of a CMVP process is bound by a given constant. For instance, a specification might require that there will be no occurrence of an event a in the trace if the size of the i-th stack is less than 7. In CMVP, this predicate can be expressed as $S(tr) = |\mathfrak{S}_i(tr)| < 7 \Rightarrow |tr|_a = 0$.

In a parallel composition construct a *concurrent stack property* is a requirement that a property holds in some stack i of a process $P_{(1,\ldots,n)}$ and another property holds in some stack $n+i$ of a concurrent process $Q_{(n+1,\ldots,n+n)}$. CMVP is able to specify such a predicate as follows: If tr is the trace of $P_{\Gamma A} \|_B Q_{\Gamma'}$, then $tr \upharpoonright A^{\checkmark}$ is the trace of P_Γ and $tr \upharpoonright B^{\checkmark}$ is the trace of $Q_{\Gamma'}$. Therefore $(\mathfrak{S}_i(tr) \upharpoonright A^{\checkmark})$ is the stack i of P_Γ and $(\mathfrak{S}_{(n+i)}(tr) \upharpoonright B^{\checkmark})$ is the stack $n+i$ of Q_Γ. For instance, we can specify that if the size of stack i of process $P_{(\gamma_1,\ldots,\gamma_n)}$ is less than 7, then there will be no invocation for module A (called by a call event a^{n+i}) in $Q_{\Gamma'}$ as follows: $S(tr) = |(\mathfrak{S}_i(tr) \upharpoonright A^{\checkmark})| < 7 \Rightarrow |(\mathfrak{S}_{(n+i)}(tr) \upharpoonright B^{\checkmark})|_{a^{n+i}} = 0$, where tr is the trace of $P_{\Gamma A} \|_B Q_{\Gamma'}$.

CMVP allows the internal trace of a (possibly recursive) module to be extracted from a CMVP trace. Any trace properties can then be specified on the extracted trace, thus formulating *internal properties of a module*. Let A be a recursive module (which is called by call event a^i) in process $P_{(\gamma_1,\ldots,\gamma_n)}$. The trace of module A is extracted using $\mathfrak{M}(tr, a^i)$, where tr is the trace of $P_{(\gamma_1,\ldots,\gamma_n)}$. Using the abstract function, the internal trace can be further extracted: $\mathfrak{A}(\mathfrak{M}(tr, a^i))$ is the internal trace of module A. For instance, a specification might require that the number of b events is always larger or equal than the number of c events in the local execution of A; this property can be expressed as $S(tr) = |\mathfrak{A}(\mathfrak{M}(tr, a^i))|_b \geq |\mathfrak{A}(\mathfrak{M}(tr, a^i))|_c$.

CMVP can accommodate the specification of the *pre- and post-conditions* of a module. The partial correctness of a procedure A specifies that if the pre-condition p holds when the procedure A is called, then if the procedure terminates then the post-condition q holds upon the return from A. Let module A be called by a^i. During the invocation of A, if some event b always precedes another event c, if A returns, then the number of b events will be smaller than the number of c events in module A. This predicate is specified as: $S(tr) = (tr = tr_1.a^i.tr_2) \wedge (tr_1 \upharpoonright b = \langle\rangle \Rightarrow tr_1 \upharpoonright c = \langle\rangle) \Rightarrow \mathfrak{C}(a^i.tr2, a^i) = \langle\rangle \vee (\mathfrak{C}(a^i.tr2, a^i) \neq \langle\rangle \wedge |\mathfrak{M}(tr2)|_b < |\mathfrak{M}(tr2)|_c)$.

Similarly, the total correctness of a procedure A specifies that if the pre-condition p holds when the procedure is called, then the procedure terminates and the post-condition q holds upon the return from A. For instance, adding to the above specification the requirement that A returns results in the following new specification: $S(tr) = (tr = tr_1.a^i.tr_2) \wedge (tr_1 \upharpoonright b = \langle\rangle \Rightarrow tr_1 \upharpoonright c = \langle\rangle) \Rightarrow \mathfrak{C}(a^i.tr2, a^i) = \langle\rangle \wedge (\mathfrak{C}(a^i.tr2, a^i) \neq \langle\rangle \wedge |\mathfrak{M}(tr2)|_b < |\mathfrak{M}(tr2)|_c)$.

9.3 CMVP Trace Verification

CMVP verification shares many features with CSP verification, while the major difference is obviously the presence of a set of stacks in a CMVP process. Since CMVP processes are MvPDA, CMVP verification rules are applied over a visible alphabet, whereas the CSP verification rules are applied over a local alphabet. In this section we give proof rules for all the CMVP operators.

The CMVP *prefix choice* construct contains a number of component processes and is always prepared to execute any one of the menu of events offered. The antecedent to the rule assumes a family of specifications $S^a(tr)$, one for each of the components $P(a)_{\Gamma(i/\gamma')}$ where $(x : A \to P(x))_{\Gamma(i/\gamma)} = x : A \to P(x)_{\Gamma(i/\gamma')}$. The proof rule for prefix choice is

$$\frac{\forall a \in A : P(a)_{\Gamma(i/\gamma')} \vdash S^a(tr)}{(x : A \to P(x))_{\Gamma(i/\gamma)} \vdash tr = \langle\rangle \vee \exists a \in A : head(tr) = a \wedge S^a(tail(tr))}$$

The *choice* constructs $P_\Gamma \Box Q_{\Gamma'}$ or $P_\Gamma \sqcap Q_{\Gamma'}$ acts either as P_Γ or as $Q_{\Gamma'}$. If $P_\Gamma \vdash S(tr)$ and $Q_{\Gamma'} \vdash T(tr)$, then the choice construct $P_\Gamma \Box Q_{\Gamma'}$ or $P_\Gamma \sqcap Q_{\Gamma'}$ satisfies the disjunction of these two specifications:

$$\frac{\begin{array}{c} P_\Gamma \vdash S(tr) \\ Q_{\Gamma'} \vdash T(tr) \end{array}}{\begin{array}{l} P_\Gamma \Box Q_{\Gamma'} \vdash S(tr) \vee T(tr) \\ P_\Gamma \sqcap Q_{\Gamma'} \vdash S(tr) \vee T(tr). \end{array}}$$

Let P_ε and Q_ε be two CMVP processes. Let A be a module which can execute only event d, called by c^i and returned by e^i. Let B be a module which after executing an event b executes its sub-module A. C is another module which first executes its sub-module A then executes an event h. Process P_ε calls module B and ends its execution when B returns, while Q_ε calls module C and ends its execution when C returns: $P = a^i \to P1_{(i/a)}$, $P1 = b \to P2$, $P2 = c^i \to P3_{(i/ca)}$, $P3 = d \to P4$, $P4_{(i/ca)} = e^i \to P5_{(i/a)}$, $P5_{(i/a)} = f^i \to P6_{(i/\perp)}$, $P6 = STOP$, and $Q = g^j \to Q1_{(j/g)}$, $Q1 = c^j \to Q2_{(j/c)}$, $Q2 = d \to Q3$, $Q3_{(j/cg)} = e^j \to Q4_{(j/g)}$, $Q4 = h \to Q5$, $Q5_{(j/g)} = i^j \to Q6_{(j/\perp)}$, $Q6 = STOP$.

Therefore $\text{traces}(P_\varepsilon) = \{\langle\rangle, \langle a^i\rangle, \langle a^i, b\rangle, \langle a^i, b, c^i\rangle, \langle a^i, b, c^i, d\rangle, \langle a^i, b, c^i, d, e^i\rangle, \langle a^i, b, c^i, d, e^i, f^i\rangle\}$, $\text{traces}(Q_\varepsilon) = \{\langle\rangle, \langle g^j\rangle, \langle g^j, c^j\rangle, \langle g^j, c^j, d\rangle, \langle g^j, c^j, d, e^j\rangle, \langle g^j, c^j, d, e^j, h\rangle, \langle g^j, c^j, d, e^j, h, i^j\rangle\}$, and $\text{traces}(P_\varepsilon \Box Q_\varepsilon) = \text{traces}(P_\varepsilon \sqcap Q_\varepsilon) = \{\langle\rangle, \langle a^i\rangle, \langle g^j\rangle, \langle a^i, b\rangle, \langle g^j, c^j\rangle, \langle a^i, b, c^i\rangle, \langle g^j, c^j, d\rangle, \langle a^i, b, c^i, d\rangle, \langle g^j, c^j, d, e^j\rangle, \langle a^i, b, c^i, d, e^i\rangle, \langle g^j, c^j, d, e^j, h\rangle, \langle a^i, b, c^i, d, e^i, f^i\rangle, \langle g^j, c^j, d, e^j, h, i^j\rangle\}$. P_ε satisfies the following non-regular property: module A can be called only if a is on its i-th stack, while Q_ε satisfies another non-regular property: module A can be called only if g is on its j-th stack:

$$P_\varepsilon \vdash S(tr) = (|\mathfrak{S}_i(tr)|_{a^i} = 0 \Rightarrow |\mathfrak{S}_i(tr)|_{c^i} = 0)$$

$$Q_\varepsilon \vdash T(tr) = (|\mathfrak{S}_j(tr)|_{g^j} = 0 \Rightarrow |\mathfrak{S}_j(tr)|_{c^j} = 0)$$

Then $P_\varepsilon \Box Q_\varepsilon$ or $P_\varepsilon \sqcap Q_\varepsilon$ satisfies the specification

$$(|\mathfrak{S}_i(tr)|_{a^i} = 0 \Rightarrow |\mathfrak{S}_i(tr)|_{c^i} = 0) \vee (|\mathfrak{S}_j(tr)|_{g^j} = 0 \Rightarrow |\mathfrak{S}_j(tr)|_{c^j} = 0)$$

A trace tr of a *parallel composition* construct $(P_A \|_B Q)_{\Gamma \cdot \Gamma'}$ will contain events from P_Γ and also $Q_{\Gamma'}$, restricted to the alphabets $A^\checkmark$ and $B^\checkmark$, respectively. Hence $P_\Gamma \vdash S(tr)$ implies $P_A \|_B Q)_{\Gamma \cdot \Gamma'} \vdash S(tr) \restriction A^\checkmark$ and $Q_{\Gamma'} \vdash T(tr)$ implies $P_A \|_B Q)_{\Gamma \cdot \Gamma'} \vdash T(tr) \restriction B^\checkmark$. Additionally, only events in $A^\checkmark$ or $B^\checkmark$ can be executed in the parallel composition, and so $\sigma(tr) \subseteq (A \cup B)^\checkmark$. We thus reach the following proof rule:

$$\frac{\begin{array}{c} P_\Gamma \vdash S(tr) \\ Q_{\Gamma'} \vdash T(tr) \end{array}}{(P_A \|_B Q)_{\Gamma \cdot \Gamma'} \vdash S(tr \restriction A^\checkmark) \wedge T(tr \restriction B^\checkmark) \wedge \sigma(tr) \subseteq (A \cup B)^\checkmark}.$$

Intuitively, parallel composition corresponds to conjunction: both the constraints S and T hold in the parallel composition (on their respective interfaces).

The parallel composition $(P_{\{a^i, b, c^i, d, e^i, f^i\}} \|_{\{g^{n+i}, c^{n+i}, d, e^{n+i}, h, i^{n+i}\}} Q)_{((\gamma_1, \ldots, \gamma_n) \cdot (\gamma_{n+1}, \ldots, \gamma_{n+m}))}$ between the two processes P_ε and Q_ε defined above will satisfy

$$\begin{array}{l} S(tr \restriction \{a^i, b, c^i, d, e^i, f^i\}^\checkmark) \wedge T(tr \restriction \{g^{n+i}, c^{n+i}, d, e^{n+i}, h, i^{n+i}\}^\checkmark) \\ \wedge \sigma(tr) \subseteq \{a^i, b, c^i, d, e^i, f^i, g^{n+i}, c^{n+i}, e^{n+i}, h, i^{n+i}\}^\checkmark \end{array}$$

which reduces to

$$\begin{array}{l} |\mathfrak{S}(tr)_{(i/\gamma)}|_{a^i} \wedge |\mathfrak{S}(tr)_{(n+i/\delta)}|_{g^{n+i}} = 0 \\ \Rightarrow |\mathfrak{S}(tr)_{(i/\gamma)}|_{c^i} = 0 \wedge \sigma(tr) \subseteq \{a^i, b, c^i, d, e^i, f^i, g^{n+i}, c^{n+i}, e^{n+i}, h, i^{n+i}\}^\checkmark \end{array}$$

The rule for *hiding* is developed as follows. A trace of the process $P_\Gamma \setminus A$ is simply a trace of P_Γ less all the events in the interface A. Therefore, for every trace of $P_\Gamma \setminus A$ there is a corresponding trace of P_Γ.

$$\frac{P_\Gamma \vdash S(tr)}{P_\Gamma \setminus A \vdash \exists tr_1 : tr_1 \setminus A = tr \wedge S(tr_1)}.$$

The process P_ε defined above satisfies the non-regular specification that there will be no occurrence of event d in the internal trace of the module which is invoked by call event a^i:

$$P_\varepsilon \vdash S(tr) = (tr = tr'.a^i.tr'' \Rightarrow |\mathfrak{A}(tr'')|_d = 0)$$

Therefore the process $P_\varepsilon \setminus \{c^i\}$ satisfies the following specification:

$$S'(tr) = (\exists tr_1 : tr_1 \setminus \{c^i\} = tr \wedge (tr_1 = tr'.a^i.tr'' \Rightarrow |\mathfrak{A}(tr'')|_d = 0))$$

A trace of the *abstract* construct $\overline{P_\Gamma}$ is obtained from the trace of P_Γ by removing all the traces of the sub-modules of the top level module, resulting in the following inference rule:

$$\frac{P_\Gamma \vdash S(tr)}{\overline{P_\Gamma} \vdash \exists tr_1 : \mathfrak{A}(tr_1) = tr \wedge S(tr_1)}$$

.

The process P_ε defined above satisfies the partial correctness property that if b is in the trace when a module is called, and if the module returns then there will be a d in the trace:

$$P_\varepsilon \vdash S(tr) = ((tr = tr'.a^i.tr'' \wedge |tr'|_b \neq 0 \wedge a^i \in \widetilde{\Sigma}_c^i) \Rightarrow (\mathfrak{C}(a^i.tr'', a^i) = \langle\rangle \vee tr_1.|\mathfrak{M}(tr'')|_d \neq 0))$$

Therefore $\overline{P_\varepsilon}$ will satisfy

$$\begin{aligned} S'(tr) = \exists tr_1 : \mathfrak{A}(tr_1) = tr \wedge (\quad & (tr = tr'.a^i.tr'' \wedge |tr'|_b \neq 0 \wedge a^i \in \widetilde{\Sigma}_c^i) \\ & \Rightarrow (\mathfrak{C}(a^i.tr'', a^i) = \langle\rangle \vee tr_1.|\mathfrak{M}(tr'')|_d \neq 0) \quad) \end{aligned}$$

The trace tr of a *renamed* process $f(P_\Gamma)$ corresponds to a renamed trace $f(tr_1)$ for some tr_1 of P_Γ. The inference rule for translating specifications through a forward renaming is

$$\frac{P_\Gamma \vdash S(tr)}{f(P_\Gamma) \vdash \exists tr_1 : S(tr_1) \wedge f(tr_1) = tr}$$

A particular specification S can be translated through the renaming function f to a specification R. This will hold if $R(tr)$ can be shown to translate to S correctly: $\forall tr : (S(tr) \Rightarrow R(f(tr)))$. If tr is a trace of $f^{-1}(P_\Gamma)$, then $f(tr)$ is a trace of P_Γ, so it must satisfy whatever specification P_Γ is known to satisfy. We thus reach the following inference rule:

$$\frac{P_\Gamma \vdash S(tr)}{f^{-1}(P_\Gamma) \vdash S(f(tr))}$$

Any given trace of the *sequential composition* $P_\Gamma; Q_{\Gamma'}$ is either a non-terminated trace of P_Γ, or a terminated trace of P_Γ followed by a trace of $Q_{\Gamma'}$. The corresponding proof rule goes like this:

$$\frac{\begin{array}{c} P_\Gamma \vdash S(tr) \\ Q_{\Gamma'} \vdash T(tr) \end{array}}{P_\Gamma; Q_{\Gamma'} \vdash \neg(\checkmark \in \sigma(tr)) \wedge S(tr) \vee \exists tr_1, tr_2 : tr = tr_1 tr_2 \wedge S(tr_1\langle\checkmark\rangle) \wedge T(tr_2)}$$

Any trace of the *interrupt* $P_\Gamma \triangle Q_{\Gamma'}$ is either a terminated trace of P_Γ or a non-terminated trace of P_Γ followed by a trace of $Q_{\Gamma'}$, hence the following inference rule:

$$\frac{\begin{array}{c} P_\Gamma \vdash S(tr) \\ Q_{\Gamma'} \vdash T(tr) \end{array}}{P_\Gamma \triangle Q_{\Gamma'} \vdash S(tr) \vee \exists tr_1, tr_2 : tr = tr_1 tr_2 \wedge \neg term(tr_1) \wedge S(tr_1) \wedge T(tr_2)}.$$

A *recursive* process N_Γ is defined by the equation $(N = P)_\Gamma$ or equivalently $(N = F(N))_\Gamma$. Therefore the following inductive rule is sufficient to define that $N_\Gamma \vdash S(tr)$:

$$\frac{\forall Y_\Gamma : (Y_\Gamma \vdash S(tr) \Rightarrow F(Y)_\Gamma \vdash S(tr))}{N_\Gamma \vdash S(tr)} \left[S(\langle\rangle)\right]$$

The traces of N_Γ are those of $\bigcup_{i\in\mathbb{N}} \text{traces}((F^i(STOP))_\Gamma)$, and all the finite unwindings of $(F(Y))_\Gamma$ begin from the process *STOP*. The inductive hypothesis is that $(F^i(STOP))_\Gamma \vdash S(tr)$. The side conditions $S(\langle\rangle)$ corresponds to the base case, since it is equivalent to $STOP \vdash S(tr)$, which in turn is equivalent to $(F^0(STOP))_\Gamma \vdash S(tr)$.

10 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 it is reasonable to 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 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 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. To substantiate this claim we introduce CMVP, a fully compositional multi-stack visibly pushdown process algebra, as a superset of CSP. The definition of the semantics of MvPDA in terms of labelled transition systems establishes MVPL as the domain language for CMVP. The operational and trace semantics for CMVP are also derived from the semantics of MvPDA.

Although CSP and other classical process algebras have proven useful for the specification and verification of hardware, communication protocols, and drivers, they are not able to adequately specify and verify more complex application software. Indeed, application software has a large number of distinct finite states, and so finite-state mechanisms are no longer practical in this context. CMVP therefore opens up a new domain in formal methods (and more specifically algebraic methods such as model-based testing) for the specification and verification of application software. Its context-free features should be adequate for specifying and verifying these complex systems.

We introduced the new abstract operator that allows for the abstraction of modules from the process environment. Variants of the CMVP abstract operator can be easily defined; for instance a variant that terminates a process when the abstracted module terminates or a variant that hides only a specific submodule can be developed. We also define functions on CMVP traces that extract stack and module information. Useful properties for software verification such as the access control of a module, stack limits, concurrent stack properties, internal properties of a module, and pre-/post-conditions of a module can be specified using these functions. Finally, the CMVP module extract function allows the trace of a module to be easily extracted from the trace of a process. The internal traces of a module can be identified by combining the module extract and abstract functions. It is thus possible to specify and verify the internal properties of modules.

This paper thus lays down the foundation that will enable MvPDA (rather than finite automata) to be the basis of concurrent process algebraic tools and theories, hence allowing for the formal specification and verification of application software.

References

[1] R. ALUR, M. ARENAS, P. BARCELO, K. ETESSAMI, N. IMMERMAN, AND L. LIBKIN, *First-order and temporal logics for nested words*, in Proceedings of the 22nd IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2007, pp. 151–160.

[2] 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.

[3] 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.

[4] J. C. M. BAETEN AND W. P. WEIJLAND, *Process Algebra*, Cambridge University Press, 1990.

[5] J. A. BERGSTRA AND J. W. KLOP, *Algebra of communicating processes with abstraction*, Theoretical Computer Science, 37 (1985), pp. 77–121.

[6] ——, *Process theory based on bisimulation semantics*, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, vol. 354 of Lecture Notes in Computer Science, Springer, 1988, pp. 50–122.

[7] S. BROOKES, C. A. R. HOARE, AND A. W. ROSCOE, *A theory of communicating sequential processes*, Journal of the ACM, 31 (1984), pp. 560–599.

[8] S. D. BROOKES, A. W. ROSCOE, AND D. J. WALKER, *An operational semantics for CSP*, Technical Report, (1988).

[9] M. BROY, B. JONSSON, J.-P. KATOEN, M. LEUCKER, AND A. PRETSCHNER, eds., *Model-Based Testing of Reactive Systems: Advanced Lectures*, vol. 3472 of Lecture Notes in Computer Science, Springer, 2005.

[10] S. D. BRUDA AND M. T. BIN WAEZ, *Communicating Visibly pushdown Processes*, in Proceedings of the 17th International Conference on Control Systems and Computer Science, vol. 1, Bucharest, Romania, May 2009, pp. 507–514.

[11] ——, *Unrestricted and disjoint operations over multi-stack visibly pushdown languages*, in Proceedings of the 6th International Conference on Software and Data Technologies (ICSOFT 2011), vol. 2, Seville, Spain, July 2011, pp. 156–161.

[12] D. CAROTENUTO, A. MURANO, AND A. PERON, *2-visibly pushdown automata*, in Developments in Language Theory, Springer, 2007, pp. 132–144.

[13] E. M. CLARKE, O. GRUMBERG, AND D. A. PELED, *Model Checking*, MIT Press, 1999.

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

[15] C. A. R. HOARE, *Communicating sequential processes*, Commun. ACM, 21 (1978), pp. 666–677.

[16] J.-P. KATOEN, *Labelled transition systems*, in Broy et al. [9], pp. 615–616.

[17] S. LA TORRE, P. MADHUSUDAN, AND G. PARLATO, *2-vpas are not determinizable*. http://www.cs.uiuc.edu/ madhu/vpa/wrong-proof-CMP07.html, 2007.

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

[19] S. LA TORRE, P. MADHUSUDAN, AND G. PARLATO, *The language theory of bounded context-switching*, in LATIN 2010: Theoretical Informatics, Springer, 2010, pp. 96–107.

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

[21] P. MADHUSUDAN, *Private communication*, 2008.

[22] A. J. R. G. MILNER, *A Calculus of Communicating Systems*, vol. 92 of Lecture Notes in Computer Science, Springer, 1980.

[23] R. MILNER, *A complete inference system for a class of regular behaviours*, Journal of Computer and System Sciences, 28 (1984), pp. 439–466.

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

[25] J. SRBA, *Beyond language equivalence on visibly pushdown automata*, Logical Methods in Computer Science, 1 (2009), pp. 1–22.