Communicating Visibly pushdown Processes [★]

Stefan D. Bruda [*] **Md Tawhid Bin Waez** [*]

[*] *Department of Computer Science, Bishop's University, Sherbrooke, Quebec J1M 1Z7, Canada (e-mail: {bruda|mtbwaez}@cs.ubishops.ca)*

Abstract: We investigate the possibility of constructing a fully compositional concurrent process algebra (dubbed Communicating Visibly pushdown Processes or CVP for short) based on visibly pushdown automata. CVP is a superset of CSP, thus combining all the good properties of finite-state algebrae with context-free features. In particular, CVP includes support for parallel composition (which has however a restricted semantics by necessity). In addition, CVP supports self-embedding recursion, stack analysis, the specification of pre- and post-conditions, and the analysis of internal properties of a module. CVP lays the basis of algebraic software verification for infinite-state processes such as application software.

Keywords: formal software engineering, software design, software verification, process algebra, infinite-state process algebra, operational semantics, visibly pushdown automaton, visibly pushdown language

1. INTRODUCTION

Pushdown automata model naturally the control flow of sequential computation in typical programming languages with nested, potentially recursive invocations of program modules. Many non-regular properties are therefore required for software verification. Such properties cannot be handled in practice using finite-state process algebrae or standard verification techniques such as model checking; therefore they generate for all practical purposes an infinite state space. Context-free process algebrae such as basic process algebra—or BPA—(Bergstra and Klop (1988)) can specify these properties. However, most software use many parallel components (such as multiple threads). In addition, many formal software verification techniques—such as may/must testing (De Nicola and Hennessy (1983))—use tests that run in parallel with the process under test; so concurrency is required for software verification, but cannot be provided by context-free algebrae since context-free languages are not closed under intersection (Lewis and Papadimitriou (1998)).

Concurrent process algebrae (Hoare (1988); Milner (1980, 1984)) describe a process using eight major operators: event prefix, choice, recursion, parallel composition, hiding, renaming, sequential composition, and interrupt. These algebrae are useful tools in software specification and verification techniques such as model-based testing (Broy et al. (2005)). A prefix or suffix of a process is also a process, so the domain language must be closed under prefix and suffix. Each operator requires certain prerequisite closure properties of the domain language: closure under union, Kleene star, intersection and shuffle, concealment, renaming, concatenation, prefix are required for choice, recursion, parallel composition, hiding, renaming, sequential composition, and interrupt, respectively. Regular languages have all these closure properties. Balanced languages (or regular hedge languages) are not closed under prefix or suffix. Deterministic context-free languages are not closed under union, intersection, and concatenation. Context-free languages are not closed under intersection. In all, regular languages are the only domain used for concurrent process algebrae and thus concurrent process algebrae cannot specify non-regular properties. As a result, algebrae are not a dominant tool in software verification.

The formal verification arena was enhanced by the recent introduction of visibly pushdown languages—or VPL—(Alur and Madhusudan (2004)) which lie between balanced languages and deterministic context-free languages. VPL have all the appealing properties that the regular languages enjoy: deterministic acceptors are as expressive as their nondeterministic counterparts; they are closed under union, intersection, complementation, concatenation, Kleene star, and prefix; membership, emptiness, language inclusion, and language equivalence are all decidable. VPL are accepted by visibly pushdown automata (vPDA) whose stack behaviour is determined by the input. A vPDA operates over an alphabet partitioned into three disjoint sets of call, return, and local symbols. Any input symbol can change the control state but calls and returns can also change the stack content. A vPDA must push [pop] one symbol on [off] the stack while reading a call [return]. We can model the execution of a recursive module using a VPL by representing the invocation of a module by a call event, the return from a module by a return event, and all the other internal actions by local events. The potential of a VPL-based concurrent process algebra is high, as VPL have most of the required closure properties.

From a software engineering point of view, vPDA have been mostly studied in terms of logic-based conformance

[★] 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.

testing—namely, model checking (Alur and Madhusudan (2004); Alur et al. (2004, 2007)). vPDA and its subclass visibly BPA (Srba (2006)) are also studied, but to a lesser extent in terms of behavioural equivalence such as bisimulation relations. There is to our knowledge no work on process algebra to represent complex, concurrent vPDA systems. We note that a good concurrent, fully compositional process algebra is the main tool in the virtually unstudied area of vPDA algebraic-based conformance testing.

We show here how the operators of a concurrent process algebra along with a new operator called "abstract" can be applied in the VPL setting. The underlying labelled transition system (LTS) of a vPDA-based process algebra is an infinite-state machine. Every state of such an LTS is represented by the combination of a vPDA state and the stack content associated to that state. We apply our technique on the operators of CSP (Brookes et al. (1984)), a finite-state process algebra (a random choice, our formalism works with all the finite-state process algebrae). We are thus proposing a vPDA-based process algebra called Communicating Visibly pushdown Processes (CVP) as a superset of CSP; when all the input symbols are locals then CVP is equivalent to CSP. Different process algebrae adopt different denotational and axiomatic approaches for specification and verification. For example, CCS has been studied under bisimulation and testing semantics (Milner (1980)), CSP under the trace and failure variants of testing semantics (Brookes et al. (1984)), and ACP under bisimulation and branching bisimulation semantics (Bergstra and Klop (1985)). In all cases one needs to establish an operational semantics first. We are laying the foundation of VPL-based algebraic specification and verification by introducing a structural operational semantics for CVP.

VPL-based process algebrae have some major advantages over any existing process algebra, as they can model the execution of parallel recursive modules and can run (recursive) test cases in parallel with a (recursive) process. In addition, the extraction of the internal trace of a module from the trace of a process that contains that module is possible, as is the extraction of the stack from a trace. Finally, one can specify recursive properties such as pre/post conditions, inter-procedural properties, and stack inspection properties. These specification features will become apparent in the verification phase which is in the works but out of the scope of this paper. These additional non-regular verification properties make CVP suitable for formal software verification.

The drawback of CVP lies in the area of parallel composition, which has by necessity a restricted semantics (for indeed VPL are not closed under shuffle). Therefore CVP is a promising, but possibly just an incipient step in the algebraic analysis of application software.

2. VISIBLY PUSHDOWN AUTOMATA AND LABELLED TRANSITION SYSTEMS

A vPDA (Alur and Madhusudan (2004)) is a tuple $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$. Φ is a finite set of states, $\Phi_{in} \subseteq \Phi$ is a set of initial states, $\Phi_F \subseteq \Phi$ is the set of final states, Γ is the (finite) stack alphabet that contains a special bottom-of-stack symbol $\perp$, and $\Omega \subseteq (\Phi \times \Gamma^*) \times \widetilde{\Sigma} \times (\Phi \times \Gamma^*)$ 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 $((P, \gamma), a, (Q, \delta)) \in \Omega$ (also written $(P, \gamma) \xrightarrow{a} (Q, \delta)$) must have the following form: if $a \in \Sigma_l \cup \{\varepsilon\}$ then $\gamma = \delta = \varepsilon$, else if $a \in \Sigma_c$ then $\gamma = \varepsilon$ and $\delta = \mathfrak{a}$ (where $\mathfrak{a}$ is the stack symbol pushed for a), else if $a \in \Sigma_r$ then if $\gamma = \perp$ then $\gamma = \delta$ (hence vPDA allow unmatched return symbols) else $\gamma = \mathfrak{a}$ and $\delta = \varepsilon$ (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 a vPDA 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 $\gamma_0 = \perp$, $q_0 \in \Phi_{in}$, $(q_{j-1i}, \varepsilon) \xrightarrow{\varepsilon} (q_{ji}, \varepsilon) \in \Omega$ for all $1 \leq i \leq k$, $1 \leq j \leq m_i$, and $(q_{i-1m_{i-1}}\gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Omega$ for every $1 \leq i \leq k$ and for some prefixes γ'_{i-1} and γ'_i of γ_{i-1} and γ_i, respectively. Whenever $q_{km_k} \in \Phi_F$ 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.

A labelled transition system—or LTS—(Broy et al. (2005)) is a triple (Θ, Σ, Δ), where Θ is a set of states, Σ is a finite set of events, and $\Delta \subseteq \Theta \times \Sigma \times \Theta$ is the transition relation. The structural operational semantics of an algebra is given in terms of LTS, so we need to establish an equivalence between vPDA and LTS. We proceed as follows:

Definition 1. Given any visibly pushdown automaton $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$, the labelled transition system $[\![M]\!]$ is defined as follows: $[\![M]\!] = ((\Phi \cup \{H, I\}) \times \Gamma^*, \widetilde{\Sigma} \cup \{\tau\}, \Delta, (I, \perp))$, where $I, H \notin \Phi$. The transition relation of $[\![M]\!]$ is $\Delta \subseteq ((\Phi \cup \{I\}) \times \Gamma^*) \times (\widetilde{\Sigma} \cup \{\tau\}) \times ((\Phi \cup \{H\}) \times \Gamma^*)$ and is defined as follows: $\Delta = \{((q, \gamma), a, (q', \gamma')) : ((q, \gamma), a, (q', \gamma')) \in \Omega\} \cup \{((I, \perp), \tau, (q, \perp)) : q \in \Phi_{in}\} \cup \{((q, \gamma), \tau, (H, \gamma)) : q \in \Phi_F\} \cup \{((q, \gamma), \tau, (q, \gamma)) : q \notin \Phi_F, \forall a \in \widetilde{\Sigma} \cup \{\tau\} : (q, \gamma) \overset{a}{\not\longrightarrow} \}$. □

A state of $[\![M]\!]$ is labelled with a state of M as well as the stack content associated with that state in the given computation. We include in Δ the transitions corresponding to the transition of the visibly pushdown automaton being modelled. $[\![M]\!]$ should be capable of starting from any state $\Phi_{in} \times \{\perp\}$; to create a unique initial state we introduce a brand new state $(I, \perp)$ and we add to Δ the set of transitions that get us nondeterministically to one of the initial states of M. We invent the final state H that has no outgoing transitions and is reachable from any final state of M via τ transitions. Such a state is useful in the construction of the LTS corresponding to the concatenation of two VPL. Informally, given two LTS with initial (final) states I' and I'' (H' and H''), respectively, the LTS corresponding to the concatenation of the two languages will have I' as initial state, H'' as final state, and all the transitions in the two original LTS plus transitions of form $((H', \gamma), \tau, (I'', \gamma))$. Such a construction will be made formal later. Its correctness will follow from the closure of VPL under concatenation and the availability of ε-transitions that do not change the stack content (Alur and

Madhusudan (2004)). Non-final states with no outgoing transitions gain a loop that performs an internal action (so that they cannot participate in a complete trace).

The following result establishes the labelled transition system $[\![M]\!]$ thus constructed as the semantic model of the visibly pushdown automaton M. Indeed, we can establish a very strong, bisimilarity-like equivalence:

Theorem 1. Let $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$ be a visibly pushdown automaton and let $[\![M]\!]$ be the tuple $((\Phi \cup \{H, I\}) \times \Gamma^*, \widetilde{\Sigma} \cup \{\tau\}, \Delta, (I, \bot))$ as constructed in Definition 1. Then M and $[\![M]\!]$ are *bisimilar*, in the sense that there exists a relation $\sim \subseteq \Phi \times (\Phi \times \Gamma^*)$ such that for every pair $p \sim (q, \gamma)$ and for any $a \in \Sigma \cup \{\varepsilon\}$ and $a' \in \Sigma \cup \{\tau\}$ such that either $a = a'$ or $a = \varepsilon$ and $a' = \tau$:

(1) Whenever $q \neq I$ and $q' \neq H$, $(p, \alpha) \xrightarrow{a} (p', \alpha') \in \Omega$ implies that $(q, \gamma) \xrightarrow{a'} (q', \gamma')$ such that $\gamma = \alpha\delta$, $\gamma' = \alpha'\delta$ for some $\delta \in \Gamma^*$ and $p' \sim (q', \gamma')$. Conversely,

(2) Whenever $q \neq I$, $q' \neq H$, $(q, \gamma) \xrightarrow{a'} (q', \gamma')$ implies that either
   (a) $q = q'$, $\gamma = \gamma'$, $a' = \tau$, and $(q, \gamma) \not\xrightarrow{a''} (q'', \gamma'')$ for any $q'' \neq q$, $\gamma \neq \gamma''$, $a'' \in \Sigma \cup \{\tau\}$, or
   (b) $(p, \alpha) \xrightarrow{a} (p', \alpha') \in \Omega$ with $\gamma = \alpha\delta$, $\gamma' = \alpha'\delta$ for some $\delta \in \Gamma^*$ and $p' \sim (q', \gamma')$;

(3) $p \in \Phi_F$ iff $(q, \gamma) \xrightarrow{\tau} (H, \gamma)$, and $p \in \Phi_{in}$ iff $(I, \bot) \xrightarrow{\tau} (q, \bot)$

Proof. Items 1 and 3 follow immediately from the definition of $[\![\cdot]\!]$.

We consider the labelled transition system in its unfolded form, i.e., as a tree. The leaves of the tree are either states of form (H, γ), or states (q, γ) that have no outgoing transitions except $(q, \gamma) \xrightarrow{\tau} (q, \gamma)$ (introduced by the definition of $[\![M]\!]$ only for those states q that are not final states of M and for which (q, γ) has no outgoing transitions). The latter are the only unfolded states.

The proof of Item 2 then proceeds by induction over the tree structure of $[\![M]\!]$ as follows: Bisimilarity for leaves is established by Item 3 and Item 2(a) of the definition of $\sim$, respectively. Consider now some non-leaf state (q, γ) with its outgoing transitions $(q, \gamma) \xrightarrow{a'} (q', \gamma')$. For $q \neq I$ and $q' \neq H$, every such a transition comes from a transition in M of form $(q, \alpha) \xrightarrow{a} (q', \alpha')$, with α and α' suitable prefixes of γ and γ', respectively. Such transitions must exist in the original automaton (since it generated the (LTS) transition under scrutiny in the first place), and $q' \sim (q', \gamma')$ by induction hypothesis. □

3. COMMUNICATING VISIBLY PUSHDOWN PROCESSES

A communicating visibly pushdown (or CVP) process is an agent which interacts with its environment (itself regarded as a process) by performing certain events drawn from a visibly pushdown alphabet $\widetilde{\Sigma} = \Sigma_l \uplus \Sigma_c \uplus \Sigma_r$. The underlying semantics of CVP consists in labelled transition systems where states represent CVP processes. The syntax of CVP will be based on the following description:

$$S ::= x : A \rightarrow S(x) \,|\, S \Box R \,|\, S \sqcap R \,|\, X \,|\, S\,{}_A\|_B R \,|$$
$$S \setminus A \,|\, \overline{S} \,|\, f(S) \,|\, f^{-1}(S) \,|\, S; R \,|\, S \triangle R$$

where S and R range over CVP processes (to be substantiated later), x over $\widetilde{\Sigma}$, A and B over $2^{\widetilde{\Sigma}}$, f over the set $\{f : \widetilde{\Sigma} \rightarrow \widetilde{\Sigma} : \forall a \in \widetilde{\Sigma}$: $f(a), f^{-1}(a) \in \Sigma_c$ $[\Sigma_l,\ \Sigma_r]$ iff $a \in \Sigma_c$ $[\Sigma_l,\ \Sigma_r] \wedge f^{-1}(a)$ is finite $\wedge$ $f(a) = \checkmark$ iff $a = \checkmark$ $\wedge$ $f(a) = \bot$ iff $a = \bot\}$ of $\widetilde{\Sigma}$-transformations. The prefix choice $x : A \rightarrow S(x)$ is a process which may engage any $x \in A$ and then its behaviour depends on that choice. $S \sqcap R$ denotes a process which may behave as either S or R, independently of its environment. $S \Box R$ denotes a process which may behave as either S or R, the choice being influenced by the environment, provided that such influence is exerted on the first occurrence of the visibly pushdown event of the composite process. $S\,{}_A\|_B R$ denotes a process which behaves like the parallel composition of S and R with the following restrictions: any visibly pushdown event performed by the composition must lie in $A \cup B$; the composition may then perform a visibly pushdown event a only if $a \in A \setminus B$ and S may perform a, or $a \in B \setminus A$ and R may perform a, or $a \in A \cap B$ and both S and R may perform (synchronously) a. $S; R$ denotes the sequential composition of S followed by R and $S \setminus A$ is the process which behaves like S except that all occurrences of $a \in A$ are rendered invisible to the environment. $\overline{\cdot}$ is a new operator ("abstract") which hides the sub-modules of a module (further substantiated later). The process $f(S)$ and $f^{-1}(S)$ derive their behaviour from that of S in that if S may perform the visibly pushdown event a then $f(S)$ may perform $f(a)$ while $f^{-1}(S)$ may engage in any visibly pushdown event b such that $f(b) = a$. $S \triangle R$ denotes R interrupting the process S: R may begin execution at any point throughout the execution of S; the performance of the first external event of R is the point at which control passes from S to R and then S is discarded. A process name X may be used as a component process in a process definition. It is bound by the definition $X = S$ where S is an arbitrary process which may include process name X.

The CVP processes $STOP$, $SKIP$ and $a \rightarrow S$ are special instances of the prefix choice construct: $STOP = x : \emptyset \rightarrow STOP$, $SKIP = x : \{\checkmark\} \rightarrow STOP$, $a \rightarrow S = x : \{a\} \rightarrow S$. The special event $\checkmark$ denotes termination. $S \underset{A}{\|} R$ is a special form of $S\,{}_A\|_B R$; it synchronize only on those visibly pushdown events appearing in A (S and R interleave for any $a \notin A$). $S \,|||\, R$ (the unrestricted interleaving of S and R) and is also a special case of $S\,{}_A\|_B R$.

The discussion in Section 2 shows that we can represent an LTS state corresponding to a CVP process as P_γ, where P is the current vPDA state and γ is the current stack content. We then refine the CVP syntax as follows:

$$P_\gamma ::= x : A \rightarrow P(x)_\gamma \,|\, P_\gamma \Box Q_\delta \,|\, P_\gamma \sqcap Q_\delta \,|\, N_\gamma \,|\, P_\gamma\,{}_A\|_B Q_\delta \,|$$
$$P_\gamma \setminus A \,|\, \overline{P_\gamma} \,|\, f(P_\gamma) \,|\, f^{-1}(P_\gamma) \,|\, P_\gamma; Q_\delta \,|\, P_\gamma \triangle Q_\delta$$

where P, Q, N range over vPDA states and γ and δ represent some (necessarily finite) prefixes of the current stack content. In settling the form of γ and δ we note that the transitions of a vPDA (and the associated LTS) depend at most on the top of the stack. Therefore, we sometimes need to mention syntactically the top of the stack only, while other times (before and after a local transition, before a call transition, and after a return transition on a non-empty stack) we do not need to mention any part of the stack. The choice operators split one process into alternatives, but these alternatives start from the same stack. We thus reach the final syntax of CVP:

$$\begin{aligned} P_{\mathfrak{a}} ::= {} & x : A \rightarrow P(x)_{\mathfrak{a}} \mid P_{\mathfrak{a}} \square Q_{\mathfrak{a}} \mid P_{\mathfrak{a}} \sqcap Q_{\mathfrak{a}} \mid N_{\mathfrak{a}} \mid P_{\mathfrak{a}}\,{}_A\|^l_B Q_{\mathfrak{b}} \mid \\ & P_{\mathfrak{a}}\,{}_A\|^o_B Q_{\mathfrak{b}} \mid P_{\mathfrak{a}} \setminus A \mid \overline{P_{\mathfrak{a}}} \mid f(P_{\mathfrak{a}}) \mid f^{-1}(P_{\mathfrak{a}}) \mid \\ & P_{\mathfrak{a}} ; Q_{\mathfrak{b}} \mid P_{\mathfrak{a}} \triangle Q_{\mathfrak{b}} \end{aligned}$$

where $\mathfrak{a}$ and $\mathfrak{b}$ range over $\Gamma \cup \{\varepsilon\}$. The operator $\|$ has been replaced by two new operators for reasons that will become evident in Section 4.2.

4. THE OPERATIONAL SEMANTICS OF CVP

We often use the subscripts l, c and r to denote the sets of local, call and return events, respectively. Any $A \in 2^{\widetilde{\Sigma}}$ is then the union of three disjoint sets A_l, A_c, A_r. A CVP operation is allowed between two CVP processes only when their partitions do not overlap[1] (else the main restriction of VPL over context-free language is violated). For any $a \in \Sigma_l$, $a \in A \cap B$ $[a \in A \setminus B]$ is equivalent with $a \in A_l \cap B_l$, or $a \in A_c \cap B_c$, or $a \in A_r \cap B_r$ $[a \in A_l \setminus B_l$, etc.]. *Matched* call-returns are defined by the specification, while *balanced* call-returns are determined at run-time: A return balance 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. We will use henceforth a itself as the symbol pushed on the stack by a CVP process that performs $a \in \Sigma_c$, setting in effect $\Gamma = \Sigma_c \cup \{\bot\}$. This helps to extract the stack from a trace and is done without loss of generality, for indeed the matching calls of a return event are determined at specification time by specifying which stack symbols can be popped by the given return (e.g., if $a, b \in \Sigma_c$ and $c \in \Sigma_r$ pops either a or b, then c is the matching return of both a and b). The stack grows to the left, hence the bottom-of-stack $\bot$ gets the rightmost place.

The semantics of prefix choice is shown in Figure 1(a). The prefix can be introduced by eight kinds of syntactic rules: $P = a \rightarrow P'$, $P = b \rightarrow P'_b$, $P_c = c \rightarrow P'$, $P_\bot = d \rightarrow P'_\bot$, $P_e = STOP$, $P_e = SKIP$, $P = STOP$, and $P = SKIP$. From the semantics of prefix choice we can recognize that a is local, b is a call, c is a balanced return, and d is an unbalanced return. $P_e = STOP$ $[P_e = SKIP]$ requires that the process enter the $STOP$ $[SKIP]$ state only when the vPDA state is P and the top of the stack is e. On the other hand, $P = STOP$ $[P = SKIP]$ does not impose any constrain on the top of the stack.

In general, we can specify any system with as well as without an explicit partitioning of its events. However, if we do not provide an explicit partition, then we can write a process as a sequence of events (like in CSP) only when all the events are locals. On the other hand, we can write any finite process (including processes with calls and returns) as a sequence (desirable in a large system), provided that we specify a partition. For example, we can write the process that produces $a^n badc^{n+1}$ as its trace without an explicit partition: $P = a \rightarrow P_a$, $P = b \rightarrow Q$, $Q = a \rightarrow R_a$, $R = d \rightarrow S$, $S_a = c \rightarrow S$, $S_\bot = STOP$. The process can also be written as $\Sigma_l = \{b, d\}$, $\Sigma_c = \{a\}$, $\Sigma_r = \{c\}$; $P = a \rightarrow P_a$, $P = b \rightarrow a \rightarrow d \rightarrow Q_a$, $Q_a = c \rightarrow Q$, $Q_\bot = STOP$ or $P = a_c \rightarrow P_a$, $P = b \rightarrow a_c \rightarrow d \rightarrow Q_a$, $Q_a = c_r \rightarrow Q$, $Q_\bot = STOP$.

$$\frac{}{(x : A \rightarrow P(x))_\gamma \xrightarrow{a} P(a)_\gamma} \left[a \in A_l\right]$$

$$\frac{}{(x : A \rightarrow P(x))_\gamma \xrightarrow{a} P(a)_{a\gamma}} \left[a \in A_c\right]$$

$$\frac{}{(x : A \rightarrow P(x))_{a\gamma} \xrightarrow{a} P(a)_\gamma} \left[a \in A_r\right]$$

$$\frac{}{(x : A \rightarrow P(x))_\bot \xrightarrow{a} P(a)_\bot} \left[a \in A_r\right]$$

(a)

$$\frac{}{P_\gamma \xrightarrow{\tau} Q_\gamma} \qquad \frac{}{P_\gamma \sqcap Q_\gamma \xrightarrow{\tau} P_\gamma} \qquad \frac{}{P_\gamma \sqcap Q_\gamma \xrightarrow{\tau} Q_\gamma}$$

(b) (c)

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{\begin{array}{c} P_\gamma \square Q_\gamma \xrightarrow{\tau} P'_\gamma \square Q_\gamma \\ Q_\gamma \square P_\gamma \xrightarrow{\tau} Q_\gamma \square P'_\gamma \end{array}} \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{\begin{array}{c} P_\gamma \square Q_\gamma \xrightarrow{a} P'_\gamma \\ Q_\gamma \square P_\gamma \xrightarrow{a} P'_\gamma \end{array}} \qquad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{\begin{array}{c} P_\gamma \square Q_\gamma \xrightarrow{a} P'_{a\gamma} \\ Q_\gamma \square P_\gamma \xrightarrow{a} P'_{a\gamma} \end{array}}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{\begin{array}{c} P_{a\gamma} \square Q_{a\gamma} \xrightarrow{a} P'_\gamma \\ Q_{a\gamma} \square P_{a\gamma} \xrightarrow{a} P'_\gamma \end{array}} \qquad \frac{P_\bot \xrightarrow{a} P'_\bot}{\begin{array}{c} P_\bot \square Q_\bot \xrightarrow{a} P'_\bot \\ Q_\bot \square P_\bot \xrightarrow{a} P'_\bot \end{array}} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{\begin{array}{c} P_\gamma \square Q_\gamma \xrightarrow{\checkmark} P'_\gamma \\ Q_\gamma \square P_\gamma \xrightarrow{\checkmark} P'_\gamma \end{array}}$$

(d)

Fig. 1. Prefix choice (a); internal transition (b); internal choice (c); external choice (d).

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{N_\gamma \xrightarrow{\tau} P'_\gamma} \left[N = P\right] \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{N_\gamma \xrightarrow{a} P'_\gamma} \left[N = P\right]$$

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{N_\gamma \xrightarrow{a} P'_{a\gamma}} \left[N = P\right] \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{N_{a\gamma} \xrightarrow{a} P'_\gamma} \left[N_a = P_a\right]$$

$$\frac{P_\bot \xrightarrow{a} P'_\bot}{N_\bot \xrightarrow{a} P'_\bot} \left[N_\bot = P_\bot\right] \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{N_\gamma \xrightarrow{\checkmark} P'_\gamma} \left[N = P\right]$$

Fig. 2. Recursion.

A CVP process can perform the internal event τ (not noticeable to the environment), which can change the current vPDA state but cannot affect the vPDA stack. The behaviour of the τ transition is described in Figure 1(b).

The semantics of internal and external choice are shown in Figures 1(c) and 1(d). The composite process has a stack content similar to the component processes. A process that chooses (once!) between ‘]’ and ‘$\rangle$’ as balanced return for ‘[’ can be defined as follows: $P = [\rightarrow P_[, P = Q \square R, Q_[= \,] \rightarrow Q, R_[= \rangle \rightarrow R, Q_\bot = STOP, R_\bot = STOP$. Note that ‘]’ and ‘$\rangle$’ are both matching returns of ‘[’ but only one is used as balanced return, depending on the environment.

A CSP recursive process creates a loop among LTS states while CVP recursive processes create loops among vPDA states only. During each recursive loop a CVP process will change the stack by the same amount, as in each loop it

[1] Meaning that $\Sigma'_x \cap \Sigma''_y = \emptyset$ for all $x \neq y$ for two partitions $\widetilde{\Sigma}' = \{\Sigma'_c, \Sigma'_r, \Sigma'_l\}$ and $\widetilde{\Sigma}'' = \{\Sigma''_c, \Sigma''_r, \Sigma''_l\}$.

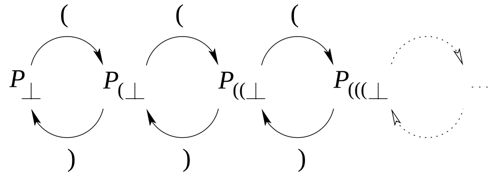

Fig. 3. Balanced brackets.

$$\frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma \setminus A \xrightarrow{\tau} P'_\gamma \setminus A}\ \left[a \in A\right] \quad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma \setminus A \xrightarrow{a} P'_\gamma \setminus A}\ \left[a \notin A\right]$$

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{P_\gamma \setminus A \xrightarrow{\tau} P'_\gamma \setminus A} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{P_\gamma \setminus A \xrightarrow{\checkmark} P'_\gamma \setminus A}$$

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{P_\gamma \setminus A \xrightarrow{a} P'_{a\gamma} \setminus A} \quad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{P_{a\gamma} \setminus A \xrightarrow{a} P'_\gamma \setminus A} \quad \frac{P_\bot \xrightarrow{a} P'_\bot}{P_\bot \setminus A \xrightarrow{\tau} P'_\bot \setminus A}$$

Fig. 4. Hiding $A \subseteq \Sigma_l$.

will visit the same vPDA states and will execute the same visible actions. If the amount of change is zero, then the process can be represented by a finite state machine; it also creates a loop among LTS states. One condition for a CVP process P_γ to be called recursive is that the process definition contains the vPDA state P. A second condition however is needed: P_γ must have enough elements in the stack so that the process reaches the recursive call. If we apply the otherwise recursive process definition $P_a = a \to P$ on $P_{\boldsymbol{a^n}\bot}$ then the recursive process will end its looping at the n-th iteration, when it will not get the top of the stack a required for the recursion to go forward.

The semantics of recursion is shown in Figure 2. Like CSP, CVP supports right- and left-embedding recursion, but it also supports self-embedding recursion such as balanced brackets: $P = (\to P_(, P_(=) \to P, P_\bot = STOP$. $P_\bot$ can produce an infinite number of LTS states and infinitely many possible traces (as shown in Figure 3) although we only have one vPDA state P. Consider now the process $Q = (\to) \to Q, Q_\bot = STOP$, which may have infinitely long traces. One can argue that the events '(' and ')' are behaving like locals in $Q_\bot$, as $Q_\bot$ can be represented by a finite state machine. However, in $P_\bot$ above the event '(' *must* be a call and the event ')' *must* be a return. Any CVP composition between $Q_\bot$ and $P_\bot$ is possible only if $(\in \Sigma_c$ and $) \in \Sigma_r$, for otherwise the partitions of $P_\bot$ and $Q_\bot$ will not coincide. It is therefore recommended that the partitioning be made in a domain-dependent way, and not in order to simplify the process definition.

Figure 4(a) shows the semantics of hiding. Note that we cannot hide everything, since VPL are not closed under such an operation: Indeed, consider the VPL $L = \{(caa)^n(rbbb)^n : n \geq 0\}$ over $\widetilde{\Sigma} = \{a, b\} \uplus \{c\} \uplus \{r\}$; hiding both c and r yields the language $\{a^{2n}b^{3n} : n \geq 0\}$, which cannot be a VPL even if we change the role of a (and make it a call) and b (making it a return). Similar phenomena occur when only c (or r) are hidden. We can however hide safely local symbols, for indeed ε-transitions are allowed in a vPDA in place of local symbols by definition. Therefore the hiding operator can only hide local symbols. Its semantics is shown in Figure 4.

$$\frac{P_{b\gamma} \xrightarrow{a} P'_{ab\gamma}}{\begin{array}{c}\overline{P_{b\gamma}} \xrightarrow{a} \overline{P'_{\tilde{a}b\gamma}} \\ \overline{P_{\tilde{b}\gamma}} \xrightarrow{\tau} \overline{P'_{\bar{a}\tilde{b}\gamma}} \\ \overline{P_{\bar{b}\gamma}} \xrightarrow{\tau} \overline{P'_{\bar{a}\bar{b}\gamma}}\end{array}} \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_{a\gamma}}{\begin{array}{c}\overline{P_{a\gamma}} \xrightarrow{a} \overline{P'_{a\gamma}} \\ \overline{P_{\tilde{a}\gamma}} \xrightarrow{\tau} \overline{P'_{\tilde{a}\gamma}} \\ \overline{P_{\bar{a}\gamma}} \xrightarrow{\tau} \overline{P'_{\bar{a}\gamma}}\end{array}} \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{\begin{array}{c}\overline{P_{a\gamma}} \xrightarrow{a} P'_\gamma \\ \overline{P_{\tilde{a}\gamma}} \xrightarrow{a} \overline{P'_\gamma} \\ \overline{P_{\bar{a}\gamma}} \xrightarrow{\tau} \overline{P'_\gamma}\end{array}}$$

$$\frac{P_\bot \xrightarrow{a} P'_\bot}{\overline{P_\bot} \xrightarrow{a} P'_\bot}\ [a \in A_r] \quad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{\overline{P_\gamma} \xrightarrow{\checkmark} P'_\gamma} \qquad \frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{\overline{P_\gamma} \xrightarrow{\tau} \overline{P'_\gamma}}$$

Fig. 5. Abstract.

Abstract—introduced over any other process algebra—hides all the sub-modules of a module. It is motivated by the abstract path in CARET and NWTL (Alur et al. (2004, 2007)). One can now hide the sub-modules from the environment, which cannot be accomplished by the hiding operator alone. Abstract produces the local trace of a module, so that one can specify internal properties of a recursive module. The semantics of abstract is presented in Figure 5. During the execution of a call, abstract pushes the corresponding stack symbol with two special markers: $\tilde{\cdot}$ for a call internal to the main module and $\bar{\cdot}$ for a call internal to a sub-module. If the top of the stack contains any special marker then every local event will be hidden; calls and returns are pushed to/popped off the stack but are otherwise hidden as well (except for top-level calls and returns in the module). If a return occurs when the top of the stack is not marked, the process will get out of abstract.

Let B (with call b and return f), and C (with call d and return e) be two modules. The top-level process P calls B and B calls C: $P = a \to Q$, $Q = b \to R_b$, $R = c \to S$, $S = d \to T_d$, $T = c \to U$, $U_d = e \to V$, $V_b = f \to W$, and $W = c \to STOP$. We can hide the sub-modules of B by using abstract: $P = a \to Q$, $Q = b \to \overline{R_b}$, $R = c \to S$, $S = d \to T_d$, $T = c \to U$, $U_d = e \to V$, $V_b = f \to W$, and $W = c \to STOP$. We actually hide sub-module C.

The semantics of forward and backward renaming is depicted in Figures 6(a) and 6(b). Renaming functions cannot change the VPL partition. Renaming can modify the matched call-returns of a process but cannot change a balanced [unbalanced] process into an unbalanced [balanced] one. There might be no "reverse" renaming that retrieves the original process or set of matched call-returns: If we apply a renaming $f\langle\rangle) =]$ on our previous example (illustrating choice), we get a process whose traces define the language $[^n]^n$; no renaming can give back the original.

Figures 7 shows the semantics of sequential composition and interrupt. Termination of the first process is indicated by the first event performed by the second process, so the stack of the second process remains $\bot$ during the lifetime of the first process.

4.1 Closure Properties

It is immediate that CVP is closed under all the operators presented above. To show this, we proceed by structural induction:

STOP, *SKIP* are obviously CVP processes. It is also easy to see that CVP is closed under: prefix choice (which just follows the definition of a transition in the associated

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{f(P)_\gamma \xrightarrow{\tau} f(P')_\gamma} \qquad \frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{f^{-1}(P)_\gamma \xrightarrow{\tau} f^{-1}(P')_\gamma}$$

$$\frac{P_\gamma \xrightarrow{a} P'_\gamma}{f(P)_\gamma \xrightarrow{f(a)} f(P')_\gamma} \qquad \frac{P_\gamma \xrightarrow{f(a)} P'_\gamma}{f^{-1}(P)_\gamma \xrightarrow{a} f^{-1}(P')_\gamma}$$

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{f(P)_\gamma \xrightarrow{f(a)} f(P')_{f(a)\gamma}} \qquad \frac{P_\gamma \xrightarrow{f(a)} P'_{f(a)\gamma}}{f^{-1}(P)_\gamma \xrightarrow{a} f^{-1}(P')_{a\gamma}}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{f(P)_{f(a)\gamma} \xrightarrow{f(a)} f(P')_\gamma} \qquad \frac{P_{f(a)\gamma} \xrightarrow{f(a)} P'_\gamma}{f^{-1}(P)_{a\gamma} \xrightarrow{a} f^{-1}(P')_\gamma}$$

$$\frac{P_\perp \xrightarrow{a} P'_\perp}{f(P)_\perp \xrightarrow{f(a)} f(P')_\perp} \qquad \frac{P_\perp \xrightarrow{f(a)} P'_\perp}{f^{-1}(P)_\perp \xrightarrow{a} f^{-1}(P')_\perp}$$

$$\frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{f(P)_\gamma \xrightarrow{\checkmark} f(P')_\gamma} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{f^{-1}(P)_\gamma \xrightarrow{\checkmark} f^{-1}(P')_\gamma}$$

(a) (b)

Fig. 6. Forward renaming (a); backward renaming (b).

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{P_\gamma ; Q_\perp \xrightarrow{\tau} P'_\gamma ; Q_\perp} \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma ; Q_\perp \xrightarrow{a} P'_\gamma ; Q_\perp} \qquad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{P_\gamma ; Q_\perp \xrightarrow{a} P'_{a\gamma} ; Q_\perp}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{P_{a\gamma} ; Q_\perp \xrightarrow{a} P'_\gamma ; Q_\perp} \qquad \frac{P_\perp \xrightarrow{a} P'_\perp}{P_\perp ; Q_\perp \xrightarrow{a} P'_\perp ; Q_\perp} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{P_\gamma ; Q_\perp \xrightarrow{\tau} Q_\perp}$$

(a)

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{P_\gamma \triangle Q_\perp \xrightarrow{\tau} P'_\gamma \triangle Q_\perp} \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma \triangle Q_\perp \xrightarrow{a} P'_\gamma \triangle Q_\perp} \qquad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{P_\gamma \triangle Q_\perp \xrightarrow{a} P'_{a\gamma} \triangle Q_\perp}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{P_{a\gamma} \triangle Q_\perp \xrightarrow{a} P'_\gamma \triangle Q_\perp} \qquad \frac{P_\perp \xrightarrow{a} P'_\perp}{P_\perp \triangle Q_\perp \xrightarrow{a} P'_\perp \triangle Q_\perp} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{P_\gamma \triangle Q_\perp \xrightarrow{\checkmark} P'_\gamma}$$

$$\frac{Q_\perp \xrightarrow{\tau} Q'_\perp}{P_\gamma \triangle Q_\perp \xrightarrow{\tau} P_\gamma \triangle Q'_\perp} \qquad \frac{Q_\perp \xrightarrow{a} Q'_{a\perp}}{P_\gamma \triangle Q_\perp \xrightarrow{a} Q'_{a\perp}}$$

$$\frac{Q_\perp \xrightarrow{a} Q'_\perp}{P_\gamma \triangle Q_\perp \xrightarrow{a} Q' \perp} \ \left[a \in \{\Sigma_l, \Sigma_r\}\right]$$

(b)

Fig. 7. Sequential composition (a); interrupt (b).

vPDA), external choice (which is a prefix choice with more than one alternative; many transitions out of one state are clearly allowed), internal choice (we connect two LTS to a common start state via τ-transitions that does not change the stack), recursion (which generates a loop from some vPDA state P back into P; this does not introduce infinite vPDA states, and manipulates the stack according to the vPDA semantics introduced by the other transitions), and renaming (unchanged VPL partition).

Consider now two LTS L' and L'' with initial [final] vPDA states I' and I'' [H' and H'']. We assume without loss of generality that the stack alphabets of L' and L'' are disjoint. The LTS corresponding to the sequential composition of L' and L'' will have I' as initial vPDA state and H'' as final vPDA state. For all the states $H'_{\delta\perp}$ in L' we make a copy of L'' with identical transitions and with a state $P_{\gamma\delta\perp}$ for every state $P_{\gamma\perp}$ of L''. The copy works the same as the original, but unbalanced returns (introduced by rules of form $P_\perp = r \to Q_\perp$) may now want to match with symbols from δ (which they won't succeed); for every rule $P_\perp = r \to Q_\perp$ we add $\{P_a = r \to Q : a \in \delta\}$ to take care of such a case. We link the copy thus described to $H'_{\delta\perp}$ using a τ transition. Closure under interrupt proceeds with the same construction but we copy L'' for every state $Q_{\delta\perp}$ of L' (not just $H'_{\delta\perp}$).

Closure under abstract is given by the fact that such an operator replaces a whole portion of the LTS by a τ transition; however, the stack at the entry point of that portion is identical to the stack at the exit point by the definition of abstract, so such a τ transition is allowed by the definition of the vPDA: Given the equivalence between an LTS and the associated vPDA (Theorem 1), the just introduced τ-transition will correspond to an ε-transition that does not affect the stack. In all, CVP is indeed an algebra:

Theorem 2. CVP is closed under prefix choice, internal and external choice, recursion, hiding, abstract, renaming, sequential composition, and interrupt.

4.2 Parallel Composition and Its Limitations

Consider the languages $L_1 = \{c_1^n r_1^n : n \geq 0\}$ and $L_2 = \{c_2^n r_2^n : n \geq 0\}$ over $\widetilde{\Sigma}$ and let $L_1 \mathbin{|||} L_2 = \{w_1 v_1 w_2 v_2 \cdots w_m v_m : w_1 w_2 \cdots w_m \in L_1, v_1 v_2 \cdots v_m \in L_2$ for all $w_i, v_i \in \widetilde{\Sigma}^*\}$ be their shuffle. Any word $w \in L_1 \mathbin{|||} L_2$ has then the following properties: $|w|_{c_1} = |w|_{r_1}$, $|w|_{c_2} = |w|_{r_2}$, $|w'|_{c_1} \geq |w'|_{r_1}$, and $|w'|_{c_2} \geq |w'|_{r_2}$ for any non-empty prefix w' of w. Then a vPDA that accepts $L_1 \mathbin{|||} L_2$ cannot exist: clearly, both r_1 and r_2 must match c_1 and also c_2 (since $c_1 c_2 r_1 r_2$ is a valid shuffle of $c_1 r_1 \in L_1$ and $c_2 r_2 \in L_2$); however, this causes the above property not to be preserved (for indeed we cannot then prevent the inadvertent acceptance of words such as $c_1 c_1 c_2 r_2 r_2 r_2$). In all, VPL are not closed under shuffle.

This is unfortunate, as the full alphabetized parallel operator requires closure under shuffle to model those runs that are not synchronized (i.e., do not contains actions from the common interface $A \cap B$). We thus need to restrict this kind of runs, and we do it as follows: We first restrict the shuffling between two VPL words such that the scope of a balanced call-return of one word cannot cross another word's balanced call-return.

In a parallel composition any common event of the component processes must synchronize during execution; any two synchronized events of the component processes behave as a single event for the composite process. Therefore, the interface parallel composition between two balanced processes is possible if the set of call and return events of one process does not overlap with the set of call and return events of another process. Two balanced processes can then interface in parallel iff $A \cap B \subseteq \Sigma_l$ ($A \cap B$ contains the common events in the interface). Figure 8(a) shows the semantics of the restricted interface parallel operator thus described.

A possibly more general parallel composition operator that allows for the synchronization of calls and returns is also possible provided that we have a one-to-one relation between calls and their matching returns. Note that in

$$\frac{P_\gamma \xrightarrow{a} P'_\gamma \qquad Q_\gamma \xrightarrow{a} Q'_\gamma}{P_\gamma\,{}_A\|^l_B Q_\gamma \xrightarrow{a} P'_\gamma\,{}_A\|^l_B Q'_\gamma}\,\big[a \in A \cap B\big] \quad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{\begin{array}{c}P_\gamma\,{}_A\|^l_B Q_\gamma \xrightarrow{a} P'_\gamma\,{}_A\|^l_B Q_\gamma \\ Q_\gamma\,{}_B\|^l_A P_\gamma \xrightarrow{a} Q_\gamma\,{}_B\|^l_A P'_\gamma\end{array}}\,\big[a \notin A \cap B\big] \quad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{\begin{array}{c}P_\gamma\,{}_A\|^l_B Q_\gamma \xrightarrow{a} P'_{a\gamma}\,{}_A\|^l_B Q_{a\gamma} \\ Q_\gamma\,{}_B\|^l_A P_\gamma \xrightarrow{a} Q_{a\gamma}\,{}_B\|^l_A P'_{a\gamma}\end{array}}\,\big[a \notin A \cap B\big]$$

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{\begin{array}{c}P_\gamma\,{}_A\|^l_B Q_\gamma \xrightarrow{\tau} P'_\gamma\,{}_A\|^l_B Q_\gamma \\ Q_\gamma\,{}_A\|^l_B P_\gamma \xrightarrow{\tau} Q_\gamma\,{}_A\|^l_B P'_\gamma\end{array}} \quad \frac{P_{b\gamma} \xrightarrow{a} P'_\gamma}{\begin{array}{c}P_{b\gamma}\,{}_A\|^l_B Q_{b\gamma} \xrightarrow{a} P'_\gamma\,{}_A\|^l_B Q_\gamma \\ Q_{b\gamma}\,{}_B\|^l_A P_{b\gamma} \xrightarrow{a} Q_\gamma\,{}_B\|^l_A P'_\gamma\end{array}}\,\big[a \notin A \cap B\big] \quad \frac{P_\perp \xrightarrow{\checkmark} P'_\perp \qquad Q_\perp \xrightarrow{\checkmark} Q'_\perp}{P_\perp\,{}_A\|^l_B Q_\perp \xrightarrow{\checkmark} P'_\perp\,{}_A\|^l_B Q'_\perp}$$

(a)

$$\frac{P_\gamma \xrightarrow{a} P'_\gamma \qquad Q_\gamma \xrightarrow{a} Q'_\gamma}{P_\gamma\,{}_A\|^o_B Q_\gamma \xrightarrow{a} P'_\gamma\,{}_A\|^o_B Q'_\gamma}\,\big[a \in A \cap B\big] \quad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma} \qquad Q_\gamma \xrightarrow{a} Q'_{a\gamma}}{P_\gamma\,{}_A\|^o_B Q_\gamma \xrightarrow{a} P'_{a\gamma}\,{}_A\|^o_B Q'_{a\gamma}}\,\big[a \in A \cap B\big] \quad \frac{P_{b\gamma} \xrightarrow{a} P'_\gamma \qquad Q_{b\gamma} \xrightarrow{a} Q'_\gamma}{P_{a\gamma}\,{}_A\|^o_B Q_{b\gamma} \xrightarrow{a} P'_\gamma\,{}_A\|^o_B Q'_\gamma}\,\big[a \in A \cap B\big]$$

$$\frac{P_\gamma \xrightarrow{a} P'_\gamma}{\begin{array}{c}P_\gamma\,{}_A\|^o_B Q_\gamma \xrightarrow{a} P'_\gamma\,{}_A\|^o_B Q_\gamma \\ Q_\gamma\,{}_B\|^o_A P_\gamma \xrightarrow{a} Q_\gamma\,{}_B\|^o_A P'_\gamma\end{array}}\,\big[a \in A \setminus B\big] \quad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{\begin{array}{c}P_\gamma\,{}_A\|^o_B Q_\gamma \xrightarrow{a} P'_{a\gamma}\,{}_A\|^o_B Q_{a\gamma} \\ Q_\gamma\,{}_B\|^o_A P_\gamma \xrightarrow{a} Q_{a\gamma}\,{}_B\|^o_A P'_{a\gamma}\end{array}}\,\big[a \in A \setminus B\big] \quad \frac{P_{b\gamma} \xrightarrow{a} P'_\gamma}{\begin{array}{c}P_{b\gamma}\,{}_A\|^o_B Q_{b\gamma} \xrightarrow{a} P'_\gamma\,{}_A\|^o_B Q_\gamma \\ Q_{b\gamma}\,{}_B\|^o_A P_{b\gamma} \xrightarrow{a} Q_\gamma\,{}_B\|^o_A P'_\gamma\end{array}}\,\big[a \in A \setminus B\big]$$

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{\begin{array}{c}P_\gamma\,{}_A\|^o_B Q_\gamma \xrightarrow{\tau} P'_\gamma\,{}_A\|^o_B Q_\gamma \\ Q_\gamma\,{}_B\|^o_A P_\gamma \xrightarrow{\tau} Q_\gamma\,{}_B\|^o_A P'_\gamma\end{array}} \quad \frac{P_\perp \xrightarrow{\checkmark} P'_\perp \qquad Q_\perp \xrightarrow{\checkmark} Q'_\perp}{P_\perp\,{}_A\|^o_B Q_\perp \xrightarrow{\checkmark} P'_\perp\,{}_A\|^o_B Q'_\perp}$$

(b)

Fig. 8. Restricted alphabetized parallel between two balanced CVP processes: with $A \cap B \subseteq \Sigma_l$ (a); with a one-to-one relation between calls and their matching returns.

some sense this does not loose generality as we can always convert any relation between matching call-returns into a one-to-one relation via a suitable renaming (let a and b be matching returns of d, e and g; we can simply rename a and b to ab and d, e, and g to deg); however, this could cause a dramatic (and unintended) transformation of the whole system. If the one-to-one relation between calls and matching returns exists, then whenever a call is synchronized [unsynchronized] then the matching return must also be synchronized [unsynchronized]. Recall that a matching in one process is not allowed to cross the scope of a matching in the other process. Figure 8(b) shows the semantics of the restricted alphabetized parallel composition operator as outlined above.

The following is immediate from the construction of the two parallel composition operators, for indeed the semantics of these operators build explicitly an LTS that can be trivially transformed into the form required in Theorem 1:

Theorem 3. Balanced CVP processes are closed under the operators $\|^l$ and $\|^o$.

5. CONCLUSIONS

CVP, a superset of CSP is the first fully compositional visibly pushdown process algebra. We introduced the semantics of a vPDA in terms of labelled transition systems, which establishes the relation between VPL and CVP. Using this vPDA semantics we presented an operational semantics for CVP. We thus lay the basis of a near future where most of the concurrent process algebraic tools and theories will be based on vPDA instead of finite automata, so that application software will become amenable to formal verification.

5.1 Advantages of CVP over Other Process Algebrae

First, CVP is based on a formalism that goes above regular languages and into the context-free realm. For instance, CVP represents recursive modules (or functions), which is not possible in finite-state process algebrae. In addition, CVP can also represent multi-threaded modules; this is possible in the finite state realm, but not in the context-free domain. CVP is at a fortunate crossroad where representing recursive, multi-threaded modules is possible.

One of the major advantages of CVP over context free process algebrae is that by analyzing the trace of a process an observer can determine the content of the composite stack of that CVP process. This is possible because of the visible nature of CVP; in a CVP trace the number of call events is equal to the number of stack symbols pushed onto the stack and the number of balanced return events is equal to the number of stack symbols popped off the stack. Given the one-to-one relation between the set of call events and the set of stack symbols one can also reconstruct the current stack from the trace and initial stack. Stack inspection properties (e.g. a module A should be invoked only within the context of a module B, with no interleaving call to an overriding module C—Alur et al. (2004)) can thus be specified at trace level.

In CVP, the environment notices when a module starts (by a call event) and terminates (by a return event). As a result, one is able to specify and verify partial and total correctness of a module; pre-conditions of that module can be checked at the starting point, and post-conditions at the end point.

Using the abstract operator the designer can hide the sub-modules from the environment. One can go further and easily create one's own variant of abstract operator, for instance a variant that only hides a sub-module, or hide sub-modules along with their top-level call and return events, or terminate the process just after the end of the module.

By using the abstract operator one can easily extract the internal trace of a module. Unlike context free and regular process algebrae, CVP will thus be able to specify the internal properties of a module (e.g., every a should be followed by b in the same module).

5.2 Disadvantages of CVP

The limitations of CVP manifest themselves in the hiding and parallel composition operators.

Hiding is useful for isolating the internal behaviour of a process from its external interface; in CVP only local behaviour can be hidden. We believe however that the inability to hide calls and returns can be overcome using suitable renamings, so we do not view the limitations of hiding as a significant problem.

Some of the restrictions of parallel composition are not significant. Indeed, the inability of the operator $\|^l$ to synchronize over calls or returns is of little consequence to system design: if one needs to synchronize calls (or returns) then one can synchronize instead special local symbols placed just before or after the calls that need to be synchronized. The one-to-one relation required by the operator $\|^o$ can also be realized by suitable (and however cumbersome) renamings, as mentioned before. Overall, the operator $\|^l$ is in our opinion the most usable.

We also note that parallel composition is applicable to balanced processes only. This is not a big restriction per se (since well-behaved processes are balanced anyway), but this restriction is not expressible syntactically. We intend to investigate such a syntactic restriction.

The more severe limitation of parallel composition stems from the restriction that the scope of a balanced call-return cannot overlap with another call-return. From incipient work on trace semantics and model-based testing theory for CVP we have reasons to believe that this limitation does not eliminate the usefulness of parallel composition. However this work is at a very early stage and the usefulness of the CVP parallel composition operators is not yet established definitely.

5.3 Future

The features of CVP presented in Section 5.1 are very powerful in conjunction with denotational semantics; indeed, most of the internal structure of a CVP process can be reconstructed based on traces. We thus intend to focus on developping such a denotational semantics, including testing semantics based on traces and then failure traces, toward a verification framework for CVP. It is our belief that in the near future vPDA-based process algebrae (or derivatives) will dominate over context free and regular process algebrae.

While developping the denotational semantics field, we will asses the utility of the parallel composition operators. Should they be found wanting, we will extend our study to multi-stack visibly pushdown automata. These automata are more complex, but may generate a less restricted parallel composition. Even in such a case this paper is useful, as the techniques used here are readily transferable to multiple stacks.

ACKNOWLEDGEMENTS

We are grateful to Madhusudan Parthasarathy for noticing an error in Bruda and Bin Waez (2008) and thus directing us to the correct closure properties (or lack thereof) of VPL under hiding and shuffle.

REFERENCES

Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., and Libkin, L. (2007). First-order and temporal logics for nested words. In *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science*, 151–160. IEEE Computer Society.

Alur, R., Etessami, K., and Madhusudan, P. (2004). A temporal logic of nested calls and returns. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 04)*, volume 2988 of *Lecture Notes in Computer Science*, 467–481. Springer.

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

Bergstra, J.A. and Klop, J.W. (1985). Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1), 77–121.

Bergstra, J.A. and Klop, J.W. (1988). Process theory based on bisimulation semantics. In J.W. de Bakker, W. de Roever, and G. Rozenberg (eds.), *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, 50–122. Springer.

Brookes, S., Hoare, C.A.R., and Roscoe, A.W. (1984). A theory of communicating sequential processes. *Journal of the ACM*, 31(3), 560–599.

Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., and Pretschner, A. (eds.) (2005). *Model-Based Testing of Reactive Systems: Advanced lectures*. Number 3472 in Lecture Notes in Computer Science. Springer.

Bruda, S.D. and Bin Waez, M.T. (2008). Visibly pushdown languages are closed under prefix, shuffle, and hiding. In *Computational Engineering in Systems Applications: Selected Papers from the WSEAS Conferences in Heraklion, Greece*, 204–209. Heraklion, Crete Island, Greece.

De Nicola, R. and Hennessy, M. (1983). Testing equivalences for processes. *Theoretical Computer Science*, 34, 83–133.

Hoare, A.R. (1988). *Communicating Sequential Processes*. Prentice-Hall.

Lewis, H.R. and Papadimitriou, C.H. (1998). *Elements of the Theory of Computation*. Prentice-Hall, 2nd edition.

Milner, A.J.R.G. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer.

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

Srba, J. (2006). Visibly pushdown automata: From language equivalence to simiulation and bisimulation. In *Annual Conference on Computer Science Logic (CSL 06)*, volume 4207 of *Lecture Notes in Computer Science*, 89–103. Springer.