# Visibly Pushdown Languages [*]

Rajeev Alur
University of Pennsylvania
alur@cis.upenn.edu

P. Madhusudan
University of Illinois, Urbana-Champaign
madhu@cs.uiuc.edu

August 22, 2005

**Abstract**

We propose the class of *visibly pushdown languages* as embeddings of context-free languages that is rich enough to model program analysis questions and yet is tractable and robust like the class of regular languages. In our definition, the input symbol determines when the pushdown automaton can push or pop, and thus the stack depth at every position. We show that the resulting class VPL of languages is closed under union, intersection, complementation, renaming, concatenation, and Kleene-$*$, and problems such as inclusion that are undecidable for context-free languages are EXPTIME-complete for visibly pushdown automata. Our framework explains, unifies, and generalizes many of the decision procedures in the program analysis literature, and allows algorithmic verification of recursive programs with respect to many context-free properties including access control properties via stack inspection and correctness of procedures with respect to pre and post conditions. We demonstrate that the class VPL is robust by giving three alternative characterizations: a logical characterization using the monadic second order (MSO) theory over words augmented with a binary matching predicate, a correspondence to regular tree languages, and a restricted class of context-free grammars. We also consider visibly pushdown languages of infinite words and show that the closure properties, MSO-characterization and the characterization in terms of regular trees carry over. The main difference with respect to the case of finite words turns out to be determinizability: nondeterministic Büchi visibly pushdown automata are strictly more expressive than deterministic Muller visibly pushdown automata.

## 1 Introduction

Pushdown automata naturally model the control flow of sequential computation in typical programming languages with nested, and potentially recursive, invocations of program modules such as procedures and method calls. Consequently, a variety of program analysis, compiler optimization, and model checking questions can be formulated as decision problems for pushdown automata. For instance, in contemporary software model checking tools, to verify whether a program $P$ (written in C, for instance) satisfies a regular correctness requirement $\varphi$ (written in linear temporal logic, for instance), the verifier first abstracts the program into a pushdown model $P^a$ with finite-state control, compiles the negation of the specification into a finite-state automaton $A_{\neg\varphi}$ that accepts all computations that violate $\varphi$ and algorithmically checks that the intersection of the languages of $P^a$ and $A_{\neg\varphi}$ is empty. The problem of checking regular requirements of pushdown models has

[*]A preliminary version of this paper appears in *Proc. 36th ACM Symposium on Theory of Computing* (STOC'04), pp. 202-211, 2004. This research was supported in part by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382.

been extensively studied in recent years leading to efficient implementations and applications to program analysis [RHS95, BEM97, BR00, ABE$^+$05, HJM$^+$02, EKS03, CW02].

While many analysis problems such as identifying dead code and accesses to uninitialized variables can be captured as regular requirements, many others require inspection of the stack or matching of calls and returns, and are context-free. Even though the general problem of checking context-free properties of pushdown automata is undecidable, algorithmic solutions have been proposed for checking many different kinds of non-regular properties. For example, access control requirements such as "a module $A$ should be invoked only if the module $B$ belongs to the call-stack," and bounds on stack size such as "if the number of interrupt-handlers in the call-stack currently is less than 5, then a property $p$ holds" require inspection of the stack, and decision procedures for certain classes of stack properties already exist [JMT99, CW02, EKS03, CMM$^+$04]. A separate class of non-regular, but decidable, properties includes the recently proposed temporal logic Caret that allows matching of calls and returns and can express the classical correctness requirements of program modules with pre and post conditions, such as "if $p$ holds when a module is invoked, the module must return, and $q$ holds upon return" [AEM04]. This suggests that the answer to the question "which class of properties are algorithmically checkable against pushdown models?" should be more general than "regular." In this paper, we propose *visibly pushdown languages* as an answer with desirable closure properties, tractable decision problems, multiple equivalent characterizations, and adequate for formulating program analysis questions.

The key feature of checkable requirements, such as stack inspection and matching calls and returns, is that the stacks in the model and the property are correlated: while the stacks are not identical, the two synchronize on when to push and when to pop, and are always of the same depth. We formalize this intuition by defining *visibly pushdown automata* (Vpa). Such an automaton operates over words over an alphabet that is partitioned into three disjoint sets of calls, returns, and local symbols. While reading a *call* symbol, the automaton must push one symbol, while reading a *return* symbol, it must pop one symbol (if the stack is non-empty), and while reading an *local* symbol, it can only update its control state. A language over a partitioned alphabet is a *visibly pushdown language* if there is such an automaton that accepts it. While modeling programs as context-free languages, we have to choose a finite alphabet of observations (for instance, an observation may denote that a particular variable is read), and a mapping from states (or transitions) of the program to observations. To model programs as visibly pushdown languages, this observation must include whether the current transition is a call to a module or a return from a module. A correctness requirement is another pushdown automaton over the alphabet with the same partitioning, and hence refers to the calls and returns of the model, and updates its stack in a synchronized manner. It is easy to see that all regular requirements, stack inspection properties, and correctness with respect to pre and post conditions (in fact, all of Caret definable properties) are visibly pushdown languages.

After introducing the class Vpl of visibly pushdown languages, we show that it has many of the desirable properties that regular languages have. Vpl is closed under union, intersection, renaming, and complementation. Given a nondeterministic Vpa, one can construct an equivalent deterministic one, and thus Vpl is a subset of deterministic context-free languages. Problems such as universality, inclusion, and equivalence are Exptime-complete for Vpas. We show three alternate characterizations of Vpl. First, every word over the partitioned alphabet can be viewed as the infix traversal of a corresponding labeled binary tree, where the subword between a call and a matching return is encoded in the left sub-tree of the call, and the suffix following the return is encoded in the right sub-tree of the call. With this correspondence, we show that the class Vpl coincides with the *regular* tree languages. Second, we augment the classical MSO—the monadic second order theory over natural numbers with successor and unary predicates (which gives an alternative logical characterization of regular languages), with a binary *matching* predicate $\mu(x, y)$ that holds if $y$ is a matching return for the call $x$. We show that the resulting theory $\mathrm{MSO}_\mu$ is expressively equivalent to VPL. Finally, we identify restrictions on the productions of context-free grammars to get a grammar-based characterization of VPL.

Analysis of liveness requirements such as "every write operation must be followed by a read operation" is formulated using automata over infinite words, and the theory of $\omega$-regular languages is well developed with many of the counterparts of the results for regular languages (c.f. [Tho90, VW94]). Consequently, we also define VPAs augmented with acceptance conditions such as *Büchi* and *Muller*, that accept visibly pushdown $\omega$-languages. We establish that the resulting class $\omega$-VPL is closed under union, intersection, renaming, and complementation. Decision problems for $\omega$-VPAs have the same complexity as the corresponding problems for VPAs. As in the finite case, the class $\omega$-VPL can be characterized by regular languages of infinite trees with exactly one infinite path, as well as by $\mathrm{MSO}_\mu$. The significant difference in the infinite case is that nondeterministic automata are strictly more expressive than the deterministic ones: the language "the stack is repeatedly bounded" (that is, for some $n$, the stack depth is at most $n$ in infinitely many positions) can be easily characterized using a nondeterministic Büchi $\omega$-VPA, and we prove that no deterministic Muller $\omega$-VPA accepts this language. However, we show that nondeterministic Büchi $\omega$-VPA can be complemented and hence problems such as checking for inclusion are still decidable.

**Related work.**

The idea of making calls and returns in a recursive program visible to the specification language for writing properties appears implicitly in [JMT99] which proposes a logic over stack contents to specify security constraints, and in [EKS03] which augments linear temporal logic with regular valuations over stack contents, and explicitly in our recent work on the temporal logic CARET that allows modalities for matching calls and returns [AEM04]. There is an extensive literature on pushdown automata, context-free languages, deterministic pushdown automata, and context-free $\omega$-languages (c.f. [ABB97]). The most related work is McNaughton's parenthesis languages with a decidable equivalence problem [McN67]. A parenthesis language is produced by a context-free grammar where each application of a production introduces a pair of parentheses, delimiting the scope of production. These parentheses can be viewed as visible calls and returns. Knuth showed that parentheses languages are closed under union, intersection, and difference (but not under complementation, primarily because parenthesis languages can consist of only well parenthesized words), and it is decidable to check whether a context-free language is a parenthesis language [Knu67]. These proofs are grammar-based and complex, and connection to pushdown automata was not studied. Furthermore, parenthesis languages are a strict subclass of visibly pushdown languages, even when restricted to languages of well-bracketed words, and the class of parenthesis languages is not closed under Kleene-$*$. Recently, balanced grammars are defined as a generalization of parenthesis languages by allowing several kinds of parentheses and regular languages in the right hand sides of productions [BB02]. It turns out that this class of languages is also a strict subclass of VPL. In the program analysis context, the notion of having unmatched returns (as in VPL) is useful as calls to procedures may not return.

It has been observed that propositional dynamic logic can be augmented with some restricted class of context-free languages, and *simple-minded* pushdown automata, which may be viewed as a restricted class of VPAs, have been proposed to explain the phenomenon [HKT00].

Finally, there is a logical characterization of context free languages using quantifications over matchings [LST94]. Also, properties expressing boundedness of stack, and repeatedly boundedness, have received a lot of attention recently [CDT02, BSW03].

## 2 Visibly pushdown languages

### 2.1 Definition via pushdown automata

A *pushdown alphabet* is a tuple $\widetilde{\Sigma} = \langle \Sigma_c, \Sigma_r, \Sigma_\ell \rangle$ that comprises three disjoint finite alphabets—$\Sigma_c$ is a finite set of *calls*, $\Sigma_r$ is a finite set of *returns* and $\Sigma_\ell$ is a finite set of *local actions.* For any such $\widetilde{\Sigma}$, let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_\ell$.

We define pushdown automata over $\widetilde{\Sigma}$. Intuitively, the pushdown automaton is restricted such that it pushes onto the stack only when it reads a call, it pops the stack only at returns, and does not use the stack when it reads local actions. The input hence controls the kind of operations permissible on the stack—however, there is no restriction on the symbols that can be pushed or popped. We call such an automaton a *visibly pushdown automaton*, defined as follows:

**Definition 1 (Visibly pushdown automaton)** *A (nondeterministic) visibly pushdown automaton on finite words over $\langle \Sigma_c, \Sigma_r, \Sigma_\ell \rangle$ is a tuple $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ where $Q$ is a finite set of states, $Q_{in} \subseteq Q$ is a set of initial states, $\Gamma$ is a finite stack alphabet that contains a special bottom-of-stack symbol $\bot$, $\delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\bot\})) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_\ell \times Q)$, and $Q_F \subseteq Q$ is a set of final states.*

A transition $(q, a, q', \gamma)$, where $a \in \Sigma_c$ and $\gamma \neq \bot$, is a push-transition where on reading $a$, $\gamma$ is pushed onto the stack and the control changes from state $q$ to $q'$. Similarly, $(q, a, \gamma, q')$ is a pop-transition where $\gamma$ is read from the top of the stack and popped (if the top of stack is $\bot$, then it is read but not popped), and the control state changes from $q$ to $q'$. Note that on local actions, there is no stack operation.

A stack is a nonempty finite sequence over $\Gamma$ ending in the bottom-of-stack symbol $\bot$; let us denote the set of all stacks as $St = (\Gamma \setminus \{\bot\})^*.\{\bot\}$. For a word $w = a_1 \dots a_k$ in $\Sigma^*$, a run of $M$ on $w$ is a sequence $\rho = (q_0, \sigma_0), \dots (q_k, \sigma_k)$, where each $q_i \in Q$, $\sigma_i \in St$, $q_0 \in Q_{in}$, $\sigma_0 = \bot$ and for every $1 \leq i \leq k$ the following holds:

**[Push]** If $a_i$ is a call, then for some $\gamma \in \Gamma$, $(q_i, a_i, q_{i+1}, \gamma) \in \delta$ and $\sigma_{i+1} = \gamma.\sigma_i$.

**[Pop]** If $a_i$ is a return, then for some $\gamma \in \Gamma$, $(q_i, a_i, \gamma, q_{i+1}) \in \delta$ and either $\gamma \neq \bot$ and $\sigma_i = \gamma.\sigma_{i+1}$, or $\gamma = \bot$ and $\sigma_i = \sigma_{i+1} = \bot$.

**[Local]** If $a_i$ is a local action, then $(q_i, a_i, q_{i+1}) \in \delta$ and $\sigma_{i+1} = \sigma_i$.

A run $\rho = (q_0, \sigma_0), \dots (q_k, \sigma_k)$ is accepting if the last state is a final state, that is, if $q_k \in Q_F$. A word $w \in \Sigma^*$ is accepted by a VPA $M$ if there is an accepting run of $M$ on $w$. The language of $M$, $L(M)$, is the set of words accepted by $M$.

VPAs cannot even read the top of the stack on local actions; however this is not a restriction as for any VPA with stack alphabet $\Gamma$ that can read the top of the stack, one can build a VPA (with a stack alphabet $\Gamma' = (\Gamma \times \Gamma) \cup \{\bot\}$) that keeps track of the top of the stack in its control state. Note that acceptance of VPAs is defined by final-state and not by emptiness of stack as the latter is too restrictive. Also, $\epsilon$-transitions are not allowed, but the expressiveness will not increase if we allow $\epsilon$-transitions that do not push or pop.

**Definition 2 (Visibly pushdown languages)** *A language of finite words $L \subseteq \Sigma^*$ is a visibly pushdown language (*VPL*) with respect to $\widetilde{\Sigma}$ (a $\widetilde{\Sigma}$-*VPL*) if there is a* VPA *$M$ over $\widetilde{\Sigma}$ such that $L(M) = L$.*

Note that, by definition, a visibly pushdown language over $\widetilde{\Sigma}$ is a context-free language over $\Sigma$. If $L$ is a regular language over $\Sigma$, then $L$ is a VPL irrespective of the partitioning of $\Sigma$ into calls, returns, and local symbols. If $\Sigma^c = \{a\}$ and $\Sigma^r = \{b\}$, then the language $\{a^n b^n \mid n \geq 0\}$ is a VPL, but the language $\{b^n a^n \mid n \geq 0\}$ is not a VPL with respect to this partitioning. The set of words with equal number of $a$ and $b$ symbols is a context-free language, but is not a VPL not matter what partitioning we choose.

While visibly pushdown languages are a strict subclass of context-free languages, for every context-free language, we can associate a visibly pushdown language over a different alphabet in the following way.

**Proposition 1 (Embedding context-free languages as VPLs)** *If $L$ is a context-free language over an alphabet $\Sigma$, then there exists a* VPL *$L'$ over the pushdown alphabet $\langle \Sigma \times \{c\}, \Sigma \times \{r\}, \Sigma \times \{\ell\} \rangle$ such that $L = h(L')$, where $h$ is a renaming function that maps each symbol $(a, s)$, for $s \in \{c, r, \ell\}$, to $a$.*

**Proof:** Let $P$ be a pushdown automaton over $\Sigma$ and let us assume, without loss of generality, that on reading a symbol, $P$ pushes or pops at most one stack symbol, and acceptance is defined using final states. Let $\Sigma_c = \Sigma \times \{c\}$, $\Sigma_r = \Sigma \times \{r\}$ and $\Sigma_\ell = \Sigma \times \{\ell\}$. Now consider the visibly pushdown automaton over $\langle \Sigma_c, \Sigma_r, \Sigma_\ell \rangle$ obtained by transforming $P$ such that every transition on $a$ that pushes onto the stack is transformed to a transition on $(a, c)$, transitions on $a$ that pop the stack are changed to transitions on $(a, r)$ and the remaining $a$-transitions are changed to transitions over $(a, \ell)$. Then a word $w = a_1 a_2 \ldots a_k$ is accepted by $P$ iff there is some augmentation $w'$ of $w$, $w' = (a_1, b_1)(a_2, b_2) \ldots (a_k, b_k)$, where each $b_i \in \{c, r, \ell\}$, such that $w'$ is accepted by $M$. Thus $M$ accepts the words in $L(P)$ annotated with information on how $P$ handles the stack. It follows that $L(P) = f(L(M))$. ■

We now briefly sketch how to model formal verification problems using VPL. Suppose we are given a boolean program $P$ (that is, a program where all the variables have finite types) with procedures (or methods) that can call one another. We choose a suitable pushdown alphabet $(\Sigma_c, \Sigma_r, \Sigma_\ell)$, and associate a symbol with every transition of $P$ with the restriction that calls are mapped to $\Sigma_c$, returns are mapped to $\Sigma_r$, and all other statements are mapped to $\Sigma_\ell$. Then, $P$ can be viewed as a generator for a visibly pushdown language $L(P)$. The specification is given as another VPL $S$ over the same alphabet, and the program is correct iff $L(P) \subseteq S$. Requirements that can be verified in this manner include all regular properties, and non-regular properties such as: partial correctness (if $p$ holds when a procedure is invoked, then, if the procedure returns, $q$ holds upon return), total correctness (if $p$ holds when a procedure is invoked, then the procedure must return and $q$ must hold at the return state), local properties (the abstract computation within a procedure obtained by skipping over subcomputations corresponding to calls to other procedures satisfies a regular property, for instance, every request is followed by a response), access control (a procedure $P_i$ can be invoked only if another procedure $P_j$ is in the current stack), and interrupt stack limits (whenever the number of interrupts in the call-stack is bounded by a given constant, a property $p$ holds). In fact, all properties from [JMT99, EKS03, AEM04] are VPLs.

## 2.2 Closure properties

Let us define first a renaming-operation. A renaming of $\widetilde{\Sigma}$ to $\widetilde{\Sigma}'$ is a function $f : \Sigma \to \Sigma'$, such that $f(\Sigma_c) \subseteq \Sigma'_c$, $f(\Sigma_r) \subseteq \Sigma'_r$ and $f(\Sigma_\ell) \subseteq \Sigma'_\ell$. A renaming $f$ is extended to words over $\Sigma$ in the natural way: $f(a_1 \dots a_k) = f(a_1) \dots f(a_k)$.

Recall that context-free languages are closed under union, renaming, concatenation and Kleene-$*$, but not under intersection. Visibly pushdown automata are however closed under all these operations:

**Theorem 1 (Closure)** *Let $L_1$ and $L_2$ be visibly pushdown languages with respect to $\widetilde{\Sigma}$. Then, $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1.L_2$ and $L_1^*$ are visibly pushdown languages with respect to $\widetilde{\Sigma}$. Also, if $f$ is a renaming of $\widetilde{\Sigma}$ to $\widetilde{\Sigma}'$, then $f(L_1)$ is a visibly pushdown language with respect to $\widetilde{\Sigma}'$.*

**Proof:** Given $L_1$ and $L_2$ accepted by VPAs $M_1$ and $M_2$, closure under union follows by taking the union of the states and transitions of $M_1$ and $M_2$ (assuming they are disjoint) and taking the new set of initial states (final states) to be the union of the initial states (final states) of $M_1$ and $M_2$.

$L_1 \cap L_2$ can be accepted by a VPA $M$ that has as its set of states the product of the states of $M_1$ and $M_2$, and as its stack alphabet the product of the stack alphabets of $M_1$ and $M_2$. When reading a call, if $M_1$ pushes $\gamma_1$ and $M_2$ pushes $\gamma_2$, then $M$ pushes $(\gamma_1, \gamma_2)$. The set of initial (final) states is the product of the initial (final) states of $M_1$ and $M_2$. Note that we crucially use the fact that $M_1$ and $M_2$, being VPAs, synchronize on the push and pop operations on the stack.

Given $L$ accepted by VPA $M$ and a renaming $f$, $f(L)$ can be accepted by simply transforming each transition of $M$ on $a$ to a transition on $f(a)$.

Given $L_1$ and $L_2$, we can design a VPA that accepts $L_1 \cdot L_2$ by nondeterministically guessing a split of the input word $w$ into $w_1$ and $w_2$. The VPA simulates $w_1$ on $M_1$ and $w_2$ on $M_2$ using different stack-alphabets; when simulating $M_2$, the stack-alphabet for $M_1$ is treated as bottom-of-stack.

A slightly more involved construction can be done to accept $L^*$. Let $M = (Q, Q_{in}, \Gamma, Q_F)$ be a VPA that accepts $L$. We build the automaton $M^*$ as follows. $M^*$ simulates $M$ step by step, but when $M$ changes its state to a final state, $M^*$ can nondeterministically update its state to an initial state, and thus, restart $M$. Upon this switch, $M^*$ must treat the stack as if it is empty, and this requires tagging its state so that in a tagged state the top can be assumed to be $\perp$ ignoring the actual content of the stack. More precisely, the state-space of $M^*$ is $Q \uplus Q'$, its stack alphabet is $\Gamma \uplus \Gamma'$, and its initial states are $Q_{in}$. Its transitions are as follows

**(Local)** For each local transition $(q, a, p) \in \delta$, $M^*$ contains the transitions $(q, a, p)$ and $(q', a, p')$, and if $p \in Q_F$, then the transitions $(q, a, r')$ and $(q', a, r')$ for each $r \in Q_{in}$.

**(Push)** For each push transition $(q, a, p, \gamma) \in \delta$, $M^*$ contains the transitions $(q, a, p, \gamma)$ and $(q', a, p, \gamma')$, and if $p \in Q_F$, then the transitions $(q, a, r', \gamma)$ and $(q', a, r', \gamma)$, for each $r \in Q_{in}$.

**(Pop)** For each pop transition $(q, a, \gamma, p) \in \delta$, $M^*$ contains the transitions $(q, a, \gamma, p)$ and $(q, a, \gamma', p')$, and if $p \in Q_F$, then the transitions $(q, a, \gamma, r')$ and $(q, a, \gamma', r')$, for each $r \in Q_{in}$. For each pop transition $(q, a, \perp, p) \in \delta$, $M^*$ contains the transitions $(q', a, \gamma, p')$ for each $\gamma \in \Gamma \cup \Gamma'$, and if $p \in Q_F$, also the transitions $(q', a, \gamma, r')$ for each $\gamma \in \Gamma \cup \Gamma'$ and $r \in Q_{in}$.

Note that from a tagged state, upon a push, $M^*$ pushes a tagged symbol on the stack so that at any point, the relevant portion of the stack is from the top down to the first tagged symbol. This allows restoring the tagging of the state while popping a tagged symbol. The set of final states for $M^*$ is $Q_F \uplus Q'_F$. It is easy to check that $L(M^*) = L^*$. ■

Note that the restriction that $f$ maps calls to calls, returns to returns, and local actions to local actions, is important for closure of VPLs under renaming. VPLs are not closed under unrestricted renaming functions since such renaming allows embedding arbitrary context-free languages (see Proposition 1).

VPAs can also be *determinized.* A VPA $(Q, Q_{in}, \Gamma, \delta, Q_F)$ is said to be deterministic if $|Q_{in}| = 1$ and for every $q \in Q$:

- for every $a \in \Sigma_\ell$, there is at most one transition of the kind $(q, a, q') \in \delta$,
- for every $a \in \Sigma_c$, there is at most one transition of the form $(q, a, q', \gamma) \in \delta$, and
- for every $a \in \Sigma_r, \gamma \in \Gamma$, there is at most one transition of the form $(q, a, \gamma, q') \in \delta$.

**Theorem 2 (Determinization)** *For any* VPA *$M$ over $\widetilde{\Sigma}$, there is a deterministic* VPA *$M'$ over $\widetilde{\Sigma}$ such that $L(M') = L(M)$. Moreover, if $M$ has $n$ states, we can construct $M'$ with $O(2^{n^2})$ states and with stack alphabet of size $O(2^{n^2} \cdot |\Sigma_c|)$.*

**Proof:** Let $L$ be accepted by a VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$. We construct an equivalent deterministic VPA as follows.

The main idea behind the proof is to do a subset construction but postpone handling the push-transitions that $M$ does; instead, we store the call actions and simulate the push-transitions corresponding to them later, namely at the time of the corresponding pop-transition. The construction will have a component $S$ that is a set of "summary" edges that keeps track of what state transitions are possible from a push-transition to the corresponding pop-transition. Using the summary information, the set of reachable states is updated.

Let $w = w_1 a_1 w_2 a_2 w_3$, where in $w_1$ every call is matched by a return, but there may be unmatched returns; $w_2$ and $w_3$ are words in which all calls and returns are matched; and $a_1$ and $a_2$ are calls (that don't have matching returns in $w$). Then after reading $w$, the VPA we construct will have as its stack $(S_2, R_2, a_2)(S_1, R_1, a_1)\bot$ and its control state will be $(S, R)$. Here $S_2$ contains all the pairs $(q, q')$ such that the VPA $M$ can get on $w_2$ from $q$ with stack $\bot$ to $q'$ with stack $\bot$. Similarly $S_1$ is the summary for $w_1$ and $S$ is the summary for $w_3$. The set $R_1$ is the set of states reachable by $M$ from any initial state on $w_1$, $R_2$ is the set of states reachable from any initial state on $w_1 a_1 w_2$ and $R$ is the set of states reached by the $M$ after $w$. We maintain such a property of the stack and control-state as an invariant.

If now a call $a_3$ occurs, we push $(S, R, a_3)$, update $R$ using all possible transitions on $a_3$ to get $R'$, and go to state $(S', R')$ where $S' = \{(q, q) \mid q \in Q\}$ is the initialization of the summary. On local actions we update the $R$-component. If a return $a_2'$ occurs, we pop $(S_2, R_2, a_2)$, and update $S_2$ and $R_2$ using the current summary $S$ along with a push-transition on $a_2$ and a corresponding pop-transition on $a_2'$.

Let $L$ be accepted by a VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$. We construct an equivalent deterministic VPA $M' = (Q', Q'_{in}, \Gamma', \delta', Q'_F)$ as follows.

Let $Q' = 2^{Q \times Q} \times 2^Q$. If $Id_Q$ denotes the set $\{(q, q) \mid q \in Q\}$, then $Q'_{in} = \{(Id_Q, Q_{in})\}$. The stack alphabet $\Gamma'$ is the set of elements $(S, R, a)$, where $(S, R) \in Q'$ and $a \in \Sigma_c$. The transition relation $\delta'$ is given by:

**(Local)** For every $a \in \Sigma_\ell$, $((S, R), a, (S', R')) \in \delta'$ where $S' = \{(q, q') \mid \exists q'' : (q, q'') \in S, (q'', a, q') \in \delta\}$, $R' = \{q' \mid \exists q \in R : (q, a, q') \in \delta\}$.

**(Call)** For every $a \in \Sigma_c$, $((S, R), a, (Id_Q, R'), (S, R, a)) \in \delta'$ where $R' = \{q' \mid \exists q \in R, \gamma \in \Gamma : (q, a, q', \gamma) \in \delta\}$.

**(Return)**

- For every $a \in \Sigma_r$, $((S,R), a, (S', R', a'), (S'', R'')) \in \delta'$ if $(S'', R'')$ satisfies the following: Let $\mathit{Update} = \{(q, q') \mid \exists q_1, q_2 \in Q, \gamma \in \Gamma : (q, a', q_1, \gamma) \in \delta, (q_1, q_2) \in S, (q_2, a, \gamma, q') \in \delta\}$. Then, $S'' = \{(q, q') \mid \exists q_3 : (q, q_3) \in S', (q_3, q') \in \mathit{Update}\}$ and $R'' = \{q' \mid \exists q \in R', (q, q') \in \mathit{Update}\}$.
- For every $a \in \Sigma_r$, $((S,R), a, \perp, (S', R')) \in \delta'$ if $S' = \{(q, q') \mid \exists q'' : (q, q'') \in S, (q'', a, \perp, q') \in \delta\}$, $R' = \{q' \mid \exists q \in R : (q, a, , \perp, q') \in \delta\}$.

The set of final states is $Q'_F = \{(S,R) \mid R \cap Q_F \neq \emptyset\}$.

Intuitively, the $R$-component keeps track of the current set of reachable states. When a call-action occurs, $R$ is propagated on all push-transitions and will be used to determine acceptance provided there is no matching return. If there is a matching return, then the summary of this call-return segment will be computed in the $S$-component and the $R$-component in the state before the call occurred will be updated with this summary. ∎

Since deterministic VPAs can be complemented by complementing the set of final states, we have:

**Corollary 1** *The class of visibly pushdown languages is closed under complementation. That is, if $L$ is a $\widetilde{\Sigma}$-VPL, then $\bar{L}$ is also a $\widetilde{\Sigma}$-VPL.*

## 2.3 Decision problems

Turning now to decidability of decision problems for VPAs, observe that since a VPA is a PDA, emptiness is decidable in time $O(n^3)$ where $n$ is the number of states in the VPA. The universality problem for VPAs is to check whether a given VPA $M$ accepts all strings in $\Sigma^*$. The inclusion problem is to find whether, given two VPAs $M_1$ and $M_2$, $L(M_1) \subseteq L(M_2)$. Though both are undecidable for PDAs, they are decidable for VPAs:

**Theorem 3** *The universality problem and the inclusion problem are* EXPTIME*-complete.*

**Proof:** Decidability and membership in EXPTIME for inclusion hold because, given VPAs $M_1$ and $M_2$, we can take the complement of $M_2$, take its intersection with $M_1$ and check for emptiness. Universality reduces to checking inclusion of the language of the fixed 1-state VPA $M_1$ accepting $\Sigma^*$ with the given VPA $M$. We now show that universality is EXPTIME-hard (hardness of inclusion follows by the above reduction).

The reduction is from the membership problem for alternating linear-space Turing machines (TM) and is similar to the proof in [BEM97] where it is shown that checking pushdown systems against linear temporal logic specifications is EXPTIME-hard.

Given an input word for such a fixed TM, a run of the TM on the word can be seen as a binary tree of configurations, where the branching is induced by the universal transitions. Each configuration can be coded using $O(n)$ bits, where $n$ is the length of the input word. Consider an infix traversal of this tree, where every configuration of the tree occurs twice: when it is reached from above for the first time, we write out the configuration and when we reach it again from its left child we write out the configuration in reverse. This encoding has the property that for any parent-child pair, there is a place along the encoding where the configuration at the parent and child appear consecutively. We then design, given an input word to the TM, a VPA that accepts a word $w$ iff $w$ is either a wrong encoding (i.e. does not correspond to a run of the TM on the input word) or $w$ encodes a run that is not accepting. The VPA checks if the word satisfies the property

|  | Closure under |  |  |  |  |
|---|---|---|---|---|---|
|  | Union | Intersection | Complement | Concat. | Kleene-$*$ |
| Regular | Yes | Yes | Yes | Yes | Yes |
| CFL | Yes | No | No | Yes | Yes |
| DCFL | No | No | Yes | No | No |
| VPL | Yes | Yes | Yes | Yes | Yes |

Figure 1: Closure Properties

|  | Decision problems for automata |  |  |
|---|---|---|---|
|  | Emptiness | Universality/Equivalence | Inclusion |
| NFA | NLOGSPACE | PSPACE | PSPACE |
| PDA | PTIME | Undecidable | Undecidable |
| DPDA | PTIME | Decidable | Undecidable |
| VPA | PTIME | EXPTIME | EXPTIME |

Figure 2: Summary of decision problems

that a configuration at a node is reversed when it is visited again using the stack. The VPA can also guess nondeterministically a parent-child pair and check whether they correspond to a wrong evolution of the TM, using the finite-state control. Thus the VPA accepts $\Sigma^*$ iff the Turing machine does not accept the input. ∎

The following table summarizes and compares closure properties and decision problems for CFLs, deterministic CFLs (DCFLs), VPLs and regular languages. In the context of reducing program analysis questions to inclusion problems for VPL, note that the complexity is polynomial in the model and exponential only in the specification.

## 3 A logical characterization

Given a word $w$ over a pushdown alphabet, recall that there is a natural notion of associating each call in $w$ with its matching return, if it exists. We can now define a logic over words over $\widetilde{\Sigma}$ that has in its signature this matching relation $\mu$.

Fix $\widetilde{\Sigma}$. A word $w$ over $\Sigma$ can be treated as a structure over the universe $U = \{1, \ldots, |w|\}$ that denotes the set of positions and a set of unary predicates $Q_a$, for each $a \in \Sigma$, where $Q_a(i)$ is true iff $w[i] = a$. Also, we have a binary relation $\mu$ over $U$ that corresponds to the matching relation of calls and returns: $\mu(i, j)$ is true iff $w[i]$ is a call and $w[j]$ is its matching return. Let us fix a countable infinite set of first-order variables $x, y, \ldots$ and a countable infinite set of monadic second-order (set) variables $X, Y, \ldots$.

**Definition 3 (Monadic second order logic with matching relation)** *The* monadic second-order logic *(MSO$_\mu$) over $\widetilde{\Sigma}$ is defined as:*

$$\varphi := Q_a(x) \mid x \in X \mid x \leq y \mid \mu(x, y) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi$$

*where $a \in \Sigma$, $x$ is a first-order variable and $X$ is a set variable.*

The models are words over $\Sigma$. The semantics is the natural semantics on the structure for words defined above. The first-order variables are interpreted over the positions of $w$, and the second-order variables range over subsets of positions. A sentence is a formula which has no free variables.

For example, if $\widetilde{\Sigma} = \langle \{a\}, \{b\}, \{d\} \rangle$, then the formula

$$\forall x.\ (\, Q_a(x) \ \Rightarrow\ \exists y.\, \exists z.\ (\, Q_b(y) \ \wedge\ Q_d(z) \ \wedge\ x \leq z \ \wedge\ z \leq y \ \wedge\ \mu(x,y)))$$

says "every call must have a corresponding return with an intervening $d$ action".

The set of all words that satisfy a sentence $\varphi$ is denoted $L(\varphi)$ and we say $\varphi$ defines this language.

**Theorem 4** *A language $L$ over $\widetilde{\Sigma}$ is a* VPL *iff there is an MSO$_\mu$ sentence $\varphi$ over $\widetilde{\Sigma}$ that defines $L$.*

**Proof:** The proof follows a similar style as in proving that MSO (without $\mu$) over words defines the same class as that of regular words (see [Tho90]).

First we show that for any sentence $\varphi$, $L(\varphi)$ is a VPL. Let us assume that in all formulas, each variable is quantified at most once. Consider any *formula* $\psi(x_1, \ldots, x_m, X_1, \ldots, X_n)$ (i.e. with free variables $Z = \{x_1, \ldots, x_m, X_1, \ldots, X_n\}$). Then consider the alphabet $\widetilde{\Sigma^Z}$ where $\Sigma^Z_s = \{(a, V) \mid a \in \Sigma_s, V : Z \rightarrow \{0,1\}$ is a valuation function$\}$, where $s \in \{c, r, \ell\}$. Then a word $w'$ over $\Sigma^Z$ encodes a word $w$ along with a valuation for the variables (provided singleton variables get assigned to exactly one position). Let $L(\psi)$ denote the set of words $w'$ over $\Sigma^Z$ such that the underlying word $w$ satisfies $\psi$ under the valuation defined by $w'$. Then we show, by structural induction, that $L(\psi)$ is a VPL.

The property that first-order variables are assigned exactly once can be checked using the finite control of a VPA. The atomic formulas $x \in X$, $Q_a(x)$ and $x \leq y$ are easy to handle.

To handle the atomic formula $\mu(x, y)$, we build a VPA that pushes the input calls onto the stack, and pops the top of the stack whenever it sees a return. The VPA accepts the string if it reads a return $(a, v)$ where $v$ assigns $y$ to 1 and the popped symbol is of the kind $(a', v')$ where $v'$ assigns $x$ to 1.

Disjunction and negation can be dealt with using the fact that VPLs are closed under union and complement. Also, existential quantification corresponds to restricting the valuation functions to exclude a variable and can be done by renaming the alphabet. Thus we obtain a VPA over $\widetilde{\Sigma}$ that accepts precisely the language $L(\varphi)$.

For the converse, consider a VPA $M = (Q, q_{in}, \Gamma, \delta, Q_F)$ where $Q = \{q_1, \ldots q_n\}$ and $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$. The corresponding MSO$_\mu$ formula will express that there is an accepting run of $M$ on the word and will be of the form:

$$\exists X_{q_1} \ldots \exists X_{q_n} \ \ \exists C_{\gamma_1} \ldots \exists C_{\gamma_k} \ \ \exists R_{\gamma_1} \ldots \exists R_{\gamma_k}$$

$$\varphi(X_{q_1}, \ldots X_{q_n}, C_{\gamma_1}, \ldots C_{\gamma_k}, R_{\gamma_1}, \ldots, R_{\gamma_k})$$

where $X_q$ stands for the positions where the run is in state $q$, and $C_\gamma$ and $R_\gamma$ stand for the positions where $\gamma$ is pushed and popped from the stack, respectively. We can write conditions in $\varphi$ that ensure that the variables $X_q$, $C_\gamma$ and $R_\gamma$ indeed define a run; the only interesting detail here is to ensure that when a stack symbol $\gamma$ is pushed (i.e. when $C_\gamma$ holds), at the corresponding return $R_\gamma$ must hold. We can state this using the $\mu$-relation by demanding that for every $x$ and $y$, if $\mu(x, y)$ holds, then there is a $\gamma$ such that $x \in C_\gamma$ and $y \in R_\gamma$. Also, $\varphi$ demands that if $y \in R_\gamma$ and there is no $x$ such that $\mu(x, y)$ holds, then $\gamma = \perp$. ∎

## 4 Relation to regular tree languages

In this section we describe a mapping from words over $\widetilde{\Sigma}$ to a particular class of trees, called *stack*-trees such that visibly pushdown languages correspond to regular sets of *stack*-trees. For technical convenience, we do not represent the empty-word $\epsilon$ as a tree.

It is well known that context-free grammars and regular tree languages are related: the derivation trees of a CFG form a regular tree language and for any regular tree language, the *yield* language of the trees is a CFL [CDG+02]. However, the tree language for a CFL is determined by the grammar and not the language itself; in this section we associate regular tree languages with VPLs.

A $\Sigma$-labeled binary tree is a structure $T = (V, \lambda)$, where $V \subseteq \{0,1\}^*$ is a finite prefix-closed language, and $\lambda : V \to \Sigma$ is a labeling function. The set $V$ represents the nodes of the tree and the edge-relation is implicit: the edges are the pairs of the form $(x, x.i)$, where $x, x.i \in V$, $i \in \{0,1\}$; $\epsilon$ is the root of the tree. Let $\mathcal{T}_\Sigma$ denote the set of all $\Sigma$-labeled trees.

Fix $\widetilde{\Sigma}$; we now define a map $\eta : \Sigma^* \to \mathcal{T}_\Sigma$. $\eta(w)$ for any $w \in \Sigma^*$ is defined inductively as follows:

- If $w = \epsilon$, $\eta(\epsilon)$ is the empty tree—i.e. with an empty set of vertices.
- If $w = cw'$, where $c$ is a call, then there are two cases:
  If the first position in $w$ (labeled $c$) has a matching return, let $w = cw_1rw_2$, where the matching return is the position after $w_1$. Then $\eta(w)$ has its root labeled $c$, the subtree rooted at its 0-child is isomorphic to $\eta(w_1)$, and the subtree rooted at its 1-child is isomorphic to $\eta(rw_2)$.

  If the first letter in $w$ does not have a matching return, then $\eta(w)$ has its root labeled $c$, has no right-child, and the subtree rooted at its 0-child is isomorphic to $\eta(w')$.
- If $w = aw'$, where $a$ is a local action or a return, $\eta(w)$ has its root labeled $a$, has no 0-child, and the subtree rooted at its 1-child is isomorphic to $\eta(w')$.

The trees that correspond to non-empty words in $\Sigma^+$ are called *stack*-trees and the set of *stack*-trees is denoted by $STree = \eta(\Sigma^+)$. If $T = \eta(w)$, then the labels on an infix traversal of $T$ recovers $w$. Hence $\eta$ is a 1-1 correspondence between words in $\Sigma^+$ and $STree$. In a stack tree, however, a call and its matching return are encoded next to each other (if $w[i]$ is a call and $w[j]$ is the corresponding return, then the node encoding $w[j]$ is the 1-child of the node encoding $w[i]$).

The tree encoding can be easily understood from the exmaple of Figure 3 which shows the tree corresponding to the word $a_1a_2a_3a_4a_5a_6a_7a_8$, where $a_1, a_3, a_5 \in \Sigma_\ell$, $a_2, a_4 \in \Sigma_c$, and $a_6, a_7, a_8 \in \Sigma_r$.

Let us now define regular tree languages using automata over trees. A tree-automaton on $\Sigma$-labeled trees is a tuple $\mathcal{A} = (Q, Q_{in}, \Delta)$ where $Q$ is a finite set of states, $Q_{in} \subseteq P$ is the set of initial states and $\Delta = \langle \Delta_{01}, \Delta_0, \Delta_1, \Delta_\emptyset \rangle$ is a set of four transition relations—$\Delta_D$ encodes transitions for nodes $u$ where $D$ is the set of children that $u$ has:

- $\Delta_{01} \subseteq P \times \Sigma \times P \times P$
- For $i = 0, 1$, $\Delta_i \subseteq P \times \Sigma \times P$
- $\Delta_\emptyset \subseteq P \times \Sigma$

Let $T = (V, \lambda)$ be a $\Sigma$-labeled tree. A run of $\mathcal{A}$ over $T$ is a $Q$-labeled tree $T_\rho = (V, \lambda_\rho)$ where $\lambda_\rho(\epsilon) \in Q_{in}$ and for every node $v \in V$:

- If $v$ has both children, then $(\lambda_\rho(v), \lambda(v), \lambda_\rho(v.0), \lambda_\rho(v.1)) \in \Delta_{01}$

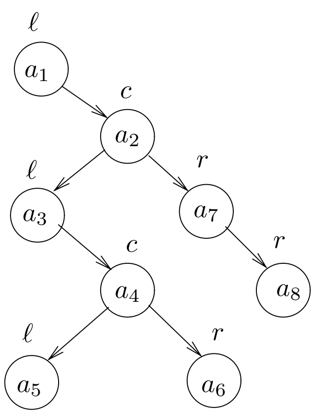

Figure 3: From words over pushdown alphabet to stack-trees

- If $v$ has one child, say the $i$-child, then $(\lambda_\rho(v), \lambda(v), \lambda_\rho(v.i)) \in \Delta_i$
- If $v$ is a leaf, then $(\lambda_\rho(v), \lambda(v)) \in \Delta_\emptyset$

Tree automata are usually defined using a set of final states; this has been absorbed into the $\Delta_\emptyset$ component of the transition relation. A tree $T$ is accepted by a tree automaton $\mathcal{A}$ iff there is a run of $\mathcal{A}$ over $T$; the set of trees accepted by $\mathcal{A}$ is the language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$. A set of $\Sigma$-labeled trees $\mathcal{L}$ is regular if there is some tree automaton such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$. The set of trees *STree* can be shown to be regular. We can show the following:

**Theorem 5** *Let $\mathcal{L}$ be a set of stack trees. Then $\eta^{-1}(\mathcal{L})$ is a* VPL *iff $\mathcal{L}$ is regular.* ■

**Proof:** Let $L = \eta^{-1}(\mathcal{L})$ be accepted by a VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$. Then we build a tree automaton $\mathcal{A} = (P, P_{in}, \Delta)$ of size polynomial in $|M|$ accepting $\mathcal{L}$, where $P = (Q \times Q) \cup (Q \times Q \times (\Gamma \setminus \{\bot\}))$ and $P_{in} = \{(q_{in}, q_f) \mid q_{in} \in Q_{in}, q_f \in Q_F\}$. The tree automaton simulates the VPA on the word corresponding to the tree. A state of the form $(q, q')$ means that the current state of the VPA that is being simulated is $q$ and $q'$ is the (guessed) state where the run of the VPA will be when it meets the next unmatched return or when the word ends (i.e. $q'$ is the guessed state of the VPA when it has finished reading the leaf on the 1-branch from the current node). The state $(q, q', \gamma)$ stands for the current state of VPA being $q$ and the commitment to end in $q'$ but additionally that the next node to be read must be a return and the top of the stack is $\gamma$.

We start with any state $(q_{in}, q_f)$, where $q_{in} \in Q_{in}$, $q_f \in Q_F$, which means that we want to end in a final state when we finish the word. The main idea of the construction is in the call—when reading a call in a state $(q, q')$ that is going to return (i.e. the node has both children), if $(q, a, q_1, \gamma)$ is a transition in the VPA, the tree automaton can split into two copies $(q_1, q_2)$ and $(q_2, q', \gamma)$ to the 0-child and 1-child respectively, for any $q_2 \in Q$. Here the tree automaton is guessing that the portion of the word within the call will start at $q_1$ and end at $q_2$ and that at the corresponding return, the VPA will take the state from $q_2$ to $q'$. The state $(q_2, q', \gamma)$ will see the return, simulate a transition from $q_2$ that pops $\gamma$ and continue. Note that the tree automaton is in general nondeterministic, even if the VPA is deterministic.

The transition relation $\Delta$ is defined as follows:

**Calls:** For each push-transition $(q, a, q_1, \gamma) \in \delta$, where $a$ is a call, we have, for every $q' \in Q$, the transitions:

**Calls that immediately return:** $((q, q'), a, (q_1, q', \gamma)) \in \Delta_1$

**Calls that return:** For every $q_2 \in Q$, $((q, q'), a, (q_1, q_2), (q_2, q', \gamma)) \in \Delta_{01}$

**Calls that do not return:** $((q, q'), a, (q_1, q')) \in \Delta_0$

**Returns that have matching calls:** For each pop-transition $(q, a, \gamma, q_1) \in \delta$, where $a$ is a return and $\gamma \neq \bot$, we have, for every $q' \in Q$, the transitions $((q, q', \gamma), a, (q_1, q')) \in \Delta_1$

**Local actions or returns without matching calls:** If $a \in \Sigma_\ell$ and $(q, a, q_1) \in \delta$, or, $a \in \Sigma_r$ and $(q, a, \bot, q_1) \in \delta$, then we have, for every $q' \in Q$, the transitions $((q, q'), a, (q_1, q')) \in \Delta_1$.

$\Delta_\emptyset$ contains the pairs $((q, q'), a)$ where $a \in \Sigma$ and, either $(q, a, q') \in \delta$ or $(q, a, q', \gamma) \in \delta$ or $(q, a, \bot, q') \in \delta$. Also, $\Delta_\emptyset$ contains the pairs $((q, q', \gamma), a)$ where $a$ is a return and $(q, a, \gamma, q') \in \delta$.

Intuitively the set of final states (i.e. $\Delta_\emptyset$) checks whether the commitment to end in the state $q'$ is indeed met. Since the initial state is of the form $(q_{in}, q_f)$ and the component $q_f$ stays in the second-component along the right-most path of the tree, we ensure that the entire run ends in $q_f$, i.e. in a final state. It is easy to show that $\mathcal{A}$ accepts exactly $\mathcal{L}$.

For the converse, let $\mathcal{A} = (P, P_{in}, \Delta)$ accept $\mathcal{L}$. Intuitively, we construct a Vpa $M$ accepting $\eta^{-1}(\mathcal{L})$ of size polynomial in $|A|$ as follows. The Vpa on a word $w$, simulates a run of $\mathcal{A}$ on $\eta(w)$. At every call $a \in \Sigma_c$, $M$ guesses whether the call will have a matching return or not. If it will, then the Vpa picks a transition $(q, a, q_0, q_1) \in \Delta_{01}$ and pushes $q_1$ onto the stack and continues simulating $q_0$. Just before the matching return is seen, we are at a leaf and $M$ will check if the current state and the label of the leaf is in $\Delta_\emptyset$. When the matching return is seen, the current state is reset to the popped state $q_1$ and $M$ continues to simulate $\mathcal{A}$ on the right-branch from the call. The other cases are analogous. To make sure the guesses made on the structure of the tree is indeed correct is a bit more involved.

Formally, $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$, where $Q = ((P \cup \{acc\}) \times (P \cup \{*, \bot\})) \cup \{fin\}$, $Q_{in} = \{(p, \bot) \mid p \in P_{in}\}$ and $\Gamma = ((P \cup \{*\}) \times (P \cup \{*, \bot\})) \cup \{\bot\}$.

A state $(p, p')$ stands for the fact that the current state of the tree automaton being simulated is $p$ and $p'$ is the top of the stack. If $p = acc$, then this means that a leaf of the tree has just been read and that we expect to see a return. At the right-most leaf where the word ends, the Vpa will be in the state *fin* if it has successfully checked that there is an accepting run of the tree automaton.

The stack at any point is of the form $(p_n, p'_n)(p_{n-1}, p'_{n-1}) \ldots (p_1, p'_1)\bot$ where $p'_1 = \bot$ and every $p'_{i+1} = p_i$, i.e. we keep track of the stack content as a chain where each element of the stack also records what symbol is below it. Using this structure, the Vpa can keep track of the top-most symbol of the stack in its finite control. The $*$ symbol is a special symbol pushed when the Vpa reads a call and guesses that it is not going to have a matching return.

The transition relation is defined as follows:

**Local actions:** For every $a \in \Sigma_\ell$, $(p, a, p_1) \in \Delta_1$, $p' \in P \cup \{*, \bot\}$, we have $((p, p'), a, (p_1, p')) \in \delta$. For every $a \in \Sigma_\ell$, $(p, a) \in \Delta_\emptyset$, we have $((p, *), a, fin) \in \delta$, $((p, \bot), a, fin) \in \delta$ and for every $p' \in P$, $((p, p'), a, (acc, p')) \in \delta$.

**Calls:** For every $a \in \Sigma_c$,

- For every $(p, a, p_0, p_1) \in \Delta_{01}$ and $p' \in P \cup \{*, \bot\}$, we have $((p, p'), a, (p_0, p_1), (p_1, p')) \in \delta$.

- For every $(p, a, p_0) \in \Delta_0$, $b \in \{*, \perp\}$, we have $((p, b), a, (p_0, *), (*, b)) \in \delta$.
- For every $(p, a, p_1) \in \Delta_1$ and $p' \in P \cup \{*, \perp\}$, we have $((p, p'), a, (acc, p_1), (p_1, p')) \in \delta$.
- For every $(p, a) \in \Delta_\emptyset$ and $b \in \{*, \perp\}$, we have $((p, b), a, \mathit{fin}, (*, b)) \in \delta$.

**Returns:** For every $a \in \Sigma_r$,

- For $(p, a, p_1) \in \Delta_1$, we have $((acc, p), a, (p, p'), (p_1, p')) \in \delta$ and $((p, \perp), a, \perp, (p_1, \perp)) \in \delta$.
- For $(p, a) \in \Delta_\emptyset$, we have $((acc, p), a, (p, p'), (acc, p')) \in \delta$ and $((p, \perp), a, \perp, \mathit{fin}) \in \delta$.

The set $Q_F = \{\mathit{fin}\}$. ■

### Summary Automata

It is worth noting that we can get an equivalent characterization of visibly pushdown automata using finite-state word automata that can jump. Let us call such automata *summary automata.* A (nondeterministic) summary automaton has finitely many control states, no stack, and reads a word over a pushdown alphabet. While reading a local or a return symbol, a summary automaton works just like a standard automaton, but while reading a call, it can send a copy to the next symbol as well as to the matching return (if such a return exists). For example, consider the word $a_1 a_2 \dots a_8$ (see Figure 3 for the types of symbols). While reading $a_1$, the automaton updates its state and moves to $a_2$. At this point, it sends a copy to symbol $a_3$ as well as a copy to symbol $a_7$ (the states in these two copies are chosen according to the transition relation, that is, for a state $q$ and a call $a$, the transition relation contains pairs of states). This is similar to a tree automaton processing the stack-tree corresponding to the word as it also sends a copy to the left child (the next symbol) and the right child (the matching return) at a call symbol. The difference is that at a leaf such as $a_6$ the control of a summary automaton will simply move to the next symbol $a_7$, and this allows propagation of state across subtrees. The summary automaton needs to be nondeterministic when it jumps in order to relate what happens before the matching return to what happens after the matching return. This is analogous to the fact that deterministic tree automata do not capture all regular tree languages. However, note that if the property as a word language is regular, then a summary automaton does not need to jump, and can be deterministic, a tree automaton operating on the tree view still may need to be nondeterministic. We leave it as an exercise to define these automata formally and prove that they capture precisely the class VPL.

## 5 Grammar-based characterization

It is well known that context-free languages can be described either by pushdown automata or by context-free grammars. In this section, we identify a class of context-free grammars that corresponds to visibly pushdown languages.

A context-free grammar over an alphabet $\Sigma$ is a tuple $G = (V, S, P)$, where $V$ is a finite set of variables, $S \in V$ is a start variable, and $P$ is a finite set of productions of the form $X \rightarrow \alpha$ such that $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. The semantics of the grammar $G$ is defined by the derivation relation $\Rightarrow$ over $(V \cup \Sigma)^*$: for every production $X \rightarrow \alpha$ in $P$ and for all words $\beta, \beta' \in (V \cup \Sigma)^*$, $\beta X \beta' \Rightarrow \beta \alpha \beta'$ holds. The language $L(G)$ of the grammar $G$ consists of all words $w \in \Sigma^*$ such that $S \Rightarrow^* w$, that is, a word $w$ over $\Sigma$ is in the language of the grammar $G$ iff it can be derived from the start variable $S$ in one or more steps.

**Definition 4 (Visibly pushdown grammar)** *A context-free grammar $G = (V, S, P)$ over $\Sigma$ is a visibly pushdown grammar with respect to the partitioning $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_\ell)$ if the set $V$ of variables is partitioned into two disjoint sets $V^0$ and $V^1$, such that all the productions in $P$ are of one the following forms*

- *$X \to \varepsilon$;*
- *$X \to aY$ such that if $X \in V^0$ then $a \in \Sigma_\ell$ and $Y \in V^0$;*
- *$X \to aYbZ$ such that $a \in \Sigma_c$ and $b \in \Sigma_r$ and $Y \in V^0$ and if $X \in V^0$ then $Z \in V^0$.*

■

The variables in $V^0$ derive only well-matched words where there is a one-to-one correspondence between calls and returns. The variables in $V^1$ derive words that can contain unmatched calls as well as unmatched returns. In the rule $X \to aY$, if $a$ is a call or a return, then either it is unmatched or its matching return or call is not remembered, and the variable $X$ must be in $V^1$. In the rule $X \to aYbZ$, the symbols $a$ and $b$ are the matching calls and returns, with a well-matched word, generated by $Y \in V^0$, sandwitched in between, and if $X$ is required to be well-matched then that requirement propagates to $Z$.

Recall that a *Dyck language* consists of well-paranthesized strings of $k$ types of parantheses (c.f. [HU79]). It is known that every context-free language is a homomorphism of the intersection of a Dyck language with a regular language (in contrast, Proposition 1 asserts that every CFL is a homomorphism of a VPL). The Dyck language can be generated by a visibly pushdown grammar as follows. The pushdown alphabet is given by: $\Sigma_c = \{[_1, [_2, \ldots [_k\}$, $\Sigma_r = \{]_1, ]_2, \ldots ]_k\}$, $\Sigma_\ell$ is empty set. There is a single (start) variable $S$, and the productions are

$$S \to \varepsilon \ \mid \ [_1\, S\, ]_1\, S \ \mid \ [_2\, S\, ]_2\, S \ \mid \ \cdots \ \mid \ [_k\, S\, ]_k\, S$$

Observe that the rule $X \to aY$ is right-linear, and is as in regular grammars. The rule $X \to aYbZ$ requires $a$ and $b$ to be matching call and return symbols, and can be encoded by a visibly pushdown automaton that, while reading $a$, pushes the obligation that the matching return should be $b$, with $Z$ to be subsequently expanded. This intuition can be made precise:

**Theorem 6 (Visibly pushdown grammars)** *For a pushdown alphabet $\widetilde{\Sigma}$, a language is a* VPL *iff it has a visibly pushdown grammar.*

**Proof:** Fix a pushdown alphabet $\widetilde{\Sigma}$. Let $G = (V, S, P)$ be a visibly pushdown grammar. We build a VPA $M_G$ that accepts $L(G)$ as follows. The set of states of $M_G$ is $V$. The unique initial state is $S$. The stack alphabet is $V \times \Sigma_r$ along with the bottom of stack $\perp$ and a dummy symbol \$. The transitions of $M_G$ from a state $X$ on a symbol $a$ are as follows:

**Local $a \in \Sigma_\ell$:** $\delta$ contains $(X, a, Y)$ for each variable $Y$ such that $X \to aY$ is a production of $G$.

**Push $a \in \Sigma_c$:** $\delta$ contains $(X, a, Y, \$)$ for each variable $Y$ such that $X \to aY$ is a production of $G$; and $(X, a, Y, (b, Z))$ for each production $X \to aYbZ$ of $G$.

**Pop $a \in \Sigma_r$:** $\delta$ contains $(X, a, \perp, Y)$ and $(X, a, \$, Y)$ for each variable $Y$ such that $X \to aY$ is a production of $G$; and if $X$ is a nullable symbol (that is, $X \to \varepsilon$ is a production of $G$) and is in $V^0$, then for each variable $Y$, $\delta$ contains $(X, a, (a, Y), Y)$.

The first clause says that the automaton can update state from $X$ to $Y$ while consuming a local action $a$ according to the rule $X \to aY$. The second clause says that while reading a call, to simulate the rule $X \to aY$ (this can happen only when $X \in V^1$), the automaton pushes the dummy symbol \$, and updates the state to $Y$. To simulate the rule $X \to aYbZ$, the automaton changes the state to $Y$ while remembering the continuation of the rule by pushing the pair $(b, Z)$ onto the stack. The third clause handles returns. The return can be consumed using a rule $X \to aY$ when $X$ is in $V^1$. If the current state is nullable and in $V^0$, then the top of the stack contains the required continuation, and the symbol being read should be consistent with it. If neither of these conditions hold, then no transition is enabled, and the automaton will reject. The accepting condition for $M_A$ is that the state should be nullable, and the top of the stack should be either \$ or $\perp$ (which means that there is no requirement concerning matching return). Since acceptance for visibly pushdown automata is defined solely based upon final states, we need to modify the construction so that the state of $M_A$ includes a bit that shows whether or not the current top of the stack equals \$ or $\perp$.

In the other direction, consider a VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$. We will construct a visibly pushdown grammar $G_M$ that generates $L(M)$. For each state $q \in Q$, the set $V^1$ has two variables $X_q$ and $Y_q$; and for every pair of states $q, p$ and stack symbol $\gamma$, the set $V^0$ has a variable $Z_{q,\gamma,p}$. Intuitively, the variable $X_q$ says that the state is $q$ and the stack is empty; the variable $Y_q$ says that the state is $q$ and the stack is non-empty, but its contents are not relevant (this happens due to unmatched calls); and the variable $Z_{q,\gamma,p}$ says that the current state is $q$, the top of the stack is $\gamma$, and the state just before the next unmatched return (i.e., the state just before popping off the current top) is required to be $p$. The start variables are $X_q$ for each $q \in Q_{in}$ (to ensure that the start variable is unique, we can add a new variable $S$ and productions $S \to \alpha$ for every production $X \to \alpha$, for each such start variable $X$). The productions are as follows:

1. For each state $q$, if $q \in Q_F$, $P$ contains $X_q \to \varepsilon$ and $Y_q \to \varepsilon$.
2. For each state $q$ and each stack symbol $\gamma$, $P$ contains $Z_{q,\gamma,q} \to \varepsilon$.
3. For each local action $a$ and states $q, p$, if $\delta$ contains the transition $(q, a, p)$, $P$ contains $X_q \to aX_p$ and $Y_q \to aY_p$.
4. For each local action $a$, states $q, p, q'$, and stack symbol $\gamma$, if $\delta$ contains the transition $(q, a, p)$, $P$ contains $Z_{q,\gamma,q'} \to aZ_{p,\gamma,q'}$.
5. For each call $a$, return $b$, stack symbol $\gamma$, and states $q, q', p, p'$, if $\delta$ contains the transitions $(q, a, \gamma, p)$ and $(q', b, \gamma, p')$, $P$ contains $X_q \to aZ_{p,\gamma,q'}bX_{p'}$ and $Y_q \to aZ_{p,\gamma,q'}bY_{p'}$.
6. For each call $a$, return $b$, stack symbols $\gamma, \gamma'$, and states $q, q', p, p', r$, if $\delta$ contains the transitions $(q, a, \gamma, p)$ and $(q', b, \gamma, p')$, $P$ contains $Z_{q,\gamma',r} \to aZ_{p,\gamma,q'}bZ_{p',\gamma',r}$.
7. For each call $a$, states $q, p$, and stack symbol $\gamma$, if $\delta$ contains the transition $(q, a, \gamma, p)$, $P$ contains $X_q \to aY_p$ and $Y_q \to aY_p$.
8. For each return $a$ and states $q, p$, if $\delta$ contains the transition $(q, a, \perp, p)$, $P$ contains $X_q \to aX_p$.

In any derivation starting from a start variable, the string contains only one trailing $X$ or $Y$ variable, which can be nullified by the first clause, provided the current state is accepting. The second clause allows nullifying a variable $Z_{q,\gamma,q'}$ when the current state $q$ is same as the target state $q'$, forcing the next symbol to be a return. Clauses 3 and 4 correspond to consuming local symbols consistent with the intended interpretation of the variables. Clauses 5 and 6 capture summarization. In state

$q$, while reading a call $a$, if the automaton can push $\gamma$ while updating its state to $p$, then we guess the matching return $b$ and the state $q'$ just before reading this matching return. The well-matched word sandwitched between is generated by the variable $Z_{p,\gamma,q'}$, and takes the automaton from $p$ to $q'$ with $\gamma$ on top. The variable following the matching return $b$ is consistent with some transition that updates state from $q'$ to $p'$ while reading $b$ and popping off $\gamma$. The clause 7 corresponds to the guess that the call being read has no matching return, and hence, it suffices to remember the state along with the fact that the stack is nonempty. The final clause allows processing of unmatched returns when the stack is empty. ■

## 6 Visibly pushdown $\omega$-languages

We now consider extensions of the results in the previous sections to infinite words over $\widetilde{\Sigma}$. An $\omega$-VPA is a tuple $M = (Q, Q_{in}, \Gamma, \delta, \mathcal{F})$ where $Q$, $Q_{in}$, $\Gamma$ and $\delta$ are as in a VPA. For any infinite word $\alpha \in \Sigma^\omega$, a run is an $\omega$-sequence $\rho = (q_0, \sigma_0)(q_1, \sigma_1)\ldots$ that is defined using the natural extension of the definition of runs on finite words. To determine whether a run $\rho$ is accepting, we consider the set $\mathit{inf}(\rho) \subseteq Q$ which is the set of all states that occur in $\rho$ infinitely often.

The acceptance condition $\mathcal{F}$ is an infinitary winning condition that can be of two kinds:

- Büchi acceptance condition: $\mathcal{F} = F \subseteq Q$ is a set of states; a run $\rho$ is accepting if $F$ is met infinitely often along the run, i.e. $\mathit{inf}(\rho) \cap F \neq \emptyset$.
- Muller acceptance condition: $\mathcal{F} = \{F_1, \ldots, F_k\}$, where each $F_i \subseteq Q$; a run $\rho$ is accepting if the set of states it meets infinitely often is an element of $\mathcal{F}$, i.e. $\mathit{inf}(\rho) \in \mathcal{F}$

An infinite word $\alpha$ is accepted by $M$ if there is some accepting run of $M$ on $\alpha$. The language of $M$, $L_\omega(M)$, is the set of all $\omega$-words that it accepts. A language of infinite words $L \subseteq \Sigma^\omega$ is said to be an $\omega$-VPL if there is some $\omega$-VPA $M$ such that $L = L_\omega(M)$.

The class of $\omega$-regular languages is well studied and is defined using automata (without stack) on infinite words; the definition of $\omega$-VPA extends this definition to visibly pushdown automata and the languages accepted by $\omega$-VPAs is a subclass of $\omega$-CFLs [CG77]. It is known that nondeterministic Büchi automata and Muller automata (without stack) can simulate each other—these constructions can be easily extended to show that nondeterministic Büchi and Muller $\omega$-VPAs are equivalent. Hence we take the definition of an $\omega$-VPL as that of being accepted by a nondeterministic Büchi $\omega$-VPA. The notion of renaming can be extended to infinite words and we can show:

**Theorem 7 (Closure)** *Let $L_1$ and $L_2$ be $\omega$-VPLs with respect to $\widetilde{\Sigma}$. Then, $L_1 \cup L_2$ and $L_1 \cap L_2$ are also $\omega$-VPLs with respect to $\widetilde{\Sigma}$. If $f$ is a renaming of $\widetilde{\Sigma}$ to $\widetilde{\Sigma}'$, then $f(L_1)$ is a visibly pushdown language with respect to $\widetilde{\Sigma}'$. Further, if $L_3$ is a VPL over $\widetilde{\Sigma}$, then $L_3.L_1$ and $(L_3)^\omega$ are $\omega$-VPLs over $\widetilde{\Sigma}$.*

**Proof:** The closure constructions for nondeterministic Büchi $\omega$-VPA are similar to the ones in Theorem 1. For the case of intersection, we need to make sure that the final states of both automata are visited infinitely often, and this can be done by adding a bit to the state as is done for intersection of Büchi automata. The construction for $L^\omega$ is similar to the construction for $L^*$ in Theorem 1. In this construction, for every transition leading to a state $p \in Q_F$, the state is nondeterministically reset to some initial state. To ensure membership in $L^\omega$, we need to require that this reset happens infinitely often, and this can be captured by a Büchi condition after adding a bit to the state. ■

However, $\omega$-VPAs cannot be determinized. For automata (without stack) on $\omega$-words it is well known that for any nondeterministic Büchi automaton, there is a deterministic Muller automaton that accepts the same language [Saf88, Tho90]. However, this is not true for visibly pushdown automata over infinite words. In fact, consider the language $L_{repbdd}$ consisting of all words $\alpha$ in $\{c, r\}^\omega$, (where $c$ is a call and $r$ is a return), that is repeatedly bounded—i.e. $\alpha \in L_{repbdd}$ if there is some $n \in \mathbb{N}$ such that the stack-depth on reading $\alpha$ infinitely often is less than or equal to $n$ [CDT02]. We can then show that this language is not determinizable.

**Theorem 8 (Deterministic vs. Nondet. $\omega$-VPAs)** *$\omega$-VPAs are not determinizable. In particular, the language $L_{repbdd}$ is an $\omega$-VPL but cannot be accepted by any deterministic Muller $\omega$-VPA.*

**Proof:** We can easily design a nondeterministic Büchi $\omega$-VPA that accepts $L_{repbdd}$. The $\omega$-VPA nondeterministically chooses a position in the word and checks whether the stack-depth at that position is the least stack-depth that occurs infinitely often. The $\omega$-VPA guesses this point by pushing a special symbol onto the stack and signals a Büchi acceptance state whenever the stack reaches that depth.

Now to show that no deterministic $\omega$-VPA can accept $L_{repbdd}$, assume the contrary and let $M = (Q, \{q_{in}\}, \Gamma, \delta, \mathcal{F})$ be a deterministic Muller automaton that accepts $L_{repbdd}$. Let $G_1 = (Q, \rightarrow)$ be the summary-graph of $M$ where $q \rightarrow q'$ iff there exists a word $w$ that is well-matched (every call has a matching return and vice versa) such that from $q$ and the empty stack, $M$ reaches $q'$ (and empty stack) on $w$ (note that $M$ hence also goes on $w$ from $(q, \sigma)$ to $(q', \sigma)$ for any stack $\sigma$). Note that if $q$ is a state reachable by $M$ on any arbitrary word, then there must be an edge from $q$ in $G_1$ (we can append an infinite word $w^\omega$ where $w$ is a well-matched word; then $M$ must have a run on it as it is repeatedly bounded and hence there will be a summary edge from $q$ on $w$). Also, since concatenation of well-matched words is well-matched, $G_1$ is transitive-closed.

Consider the strongly connected components (SCC) of $G_1$. A sink SCC of $G_1$ is a strongly connected component $S'$ such that every edge $(q, q') \in G_1$, if $q \in S'$ then $q'$ is also in $S'$.

Now let $G_2 = (Q, \Rightarrow)$ which is a super-graph of $G_1$ with additional *call-edges*: $(q, q')$ is a call-edge if there is a transition from $q$ to $q'$ in $M$ on a call $c \in \Sigma_c$. We now want to show:

**(*)** there is a sink SCC $S$ of $G_1$ and a state $q \in S$ reachable from $q_{in}$ in $G_2$ such that there is a cycle involving $q$ in $G_2$ that includes a call-edge.

If (*) is true, then we can show a contradiction. Consider a word that from $q_{in}$ reaches $q$ using the summary edges and call-edges in $G_2$ and then loops forever in $S$. This word is repeatedly bounded and hence must be accepting and hence $Q_S$, the union of all states reachable using summary edges in $S$ must be in the Muller set $\mathcal{F}$. Now consider another word that takes $M$ from $q_{in}$ to $q$ but now follows the cycle that involves $q$. Along the cycle, some are words corresponding to summary edges and some are calls; note that there will be no returns that match these calls. If $Q'$ is the set of states visited from going to $q$ to $q$ along the cycle, then we can show that $Q' \subseteq Q_S$ (after the cycle, if we feed enough returns to match the calls then we get a summary edge from $q$; however summary edges from $q$ go only to $S$ and hence the states seen must be a subset of $Q_S$). Now, consider the infinite word that first goes from $q_{in}$ to $q$, and then alternately takes the cycle in $G_2$ to reach $q$ and takes the set of all possible summary edges in $S$ to reach $q$ again. This word is not repeatedly bounded (as it has unmatched calls during the cycle in $G_2$) but the set of states seen infinitely often is $Q_S$ and is hence accepted, a contradiction.

Now let us show (*). Note that from any state, one can take summary edges in $G_1$ to reach a sink SCC of $G_1$. Let us take summary edges from $q_{in}$ to reach a state $q_1$ in a sink SCC $S_1$ of $G_1$

and take the call edge from $q_1$ to reach $q_1'$. If $q_1' \in S_1$, we are done as we have a cycle from $q_1$ to $q_1$ using a call-edge. Otherwise take summary edges in $G_1$ from $q_1'$ to reach a state $q_2$ in a sink SCC $S_2$. If $S_2 = S_1$, we are again done, else take a call-edge from $q_2$ and repeat till some sink SCC repeats. ■

Though $\omega$-VPAs cannot be determinized, they can be complemented.

**Theorem 9 (Complementability)** *The class of $\omega$-VPLs over $\widetilde{\Sigma}$ is closed under complement.*

**Proof:** Let $M = (Q, Q_{in}, \Gamma, \delta, F)$ be a $\omega$-VPA over $\widetilde{\Sigma}$. We can assume that there are transitions from every state on every letter, and that there is a run of $M$ on every word. Consider a word $\alpha \in \Sigma^\omega$. Then $\alpha$ can be factored into words where we treat a segment of letters starting at a call and ending at the matching return as a block. This factorization can be, say, of the form $\alpha = a_0a_1w_0a_2w_1w_2a_3\ldots$ where each $w_i$ is a finite word that is a properly matched word ($w_i$ starts with a call, ends with the matching return). The letters $a_i$ can be calls, returns or local actions but if some $a_i$ is a call, then $a_j$ $(j > i)$ cannot be a return. Consider the following word which can be seen as a pseudo-run of $M$ on $\alpha$: $\alpha' = a_0a_1S_0a_2S_1S_2a_3\ldots$ where each $S_i$ is the set of all triples $(q, q', f)$ where $q, q' \in Q$, $f \in \{0, 1\}$ such that there is some run $\rho$ of $M$ on $w_i$ starting at the state $q$ and ending at the state $q'$ and $\rho$ meets a state in $F$ iff $f = 1$.

Let $\mathcal{S}$ denote the set of all sets $S$ where $S$ contains triples of the form $(q, q', f)$ where $q, q' \in Q$ and $f \in \{0, 1\}$; the summary edges used above hence are in $\mathcal{S}$. Then $PR = (\Sigma_r \cup \Sigma_\ell \cup \mathcal{S})^\omega \cup (\Sigma_r \cup \Sigma_\ell \cup \mathcal{S})^*.(\Sigma_c \cup \Sigma_\ell \cup \mathcal{S})^\omega$ denotes the set of all possible pseudo-runs. We can now construct a nondeterministic Büchi automaton (with no stack) that accepts all *accepting pseudo-runs*; a pseudo-run is accepting if there is a run of $M$ that runs over the $\Sigma$-segments of the word in the usual way, and on letters $S \in \mathcal{S}$, updates the state using a summary edge in $S$ and either meets $F$ infinitely often or uses summary edges of the form $(q, q', 1)$ infinitely often. Note that a word $\alpha$ is accepted by $M$ iff the pseudo-run corresponding to $\alpha$ is accepting. Let the language of accepting pseudo runs be $L_R$. Now we construct a deterministic Muller automaton $\mathcal{A}_R$ that accepts the *complement* of $L_R$ ([Tho90]).

We now construct a nondeterministic Muller $\omega$-VPA that, on reading $\alpha$, generates the pseudo-run of $\alpha$ online and checks whether it is in $L_R$. The factorization of $\alpha$ into segments is done nondeterministically and the summary edges are computed online using the stack (as in the proof of determinization of VPAs on finite words). The resulting automaton can be converted to a Büchi automaton and accepts the complement of $L_\omega(M)$. ■

We can also characterize the class $\omega$-VPL using MSO$_\mu$ which is now interpreted over infinite words, using the fact that $\omega$-VPLs are closed under union, complement and renaming:

**Theorem 10 (MSO$_\mu$-characterization)** *A language $L$ of infinite strings over $\widetilde{\Sigma}$ is an $\omega$-VPL iff there is an MSO$_\mu$ sentence $\varphi$ over $\widetilde{\Sigma}$ that defines $L$.*

**Proof:** The proof of Theorem 4 can be easily adopted to the infinite case. For translation from MSO$_\mu$ formulas to $\omega$-VPA we use the same encoding; the requirement that the first-order variables are assigned exactly once can be enforced using a Büchi acceptance condition, and disjunction, negation, and existential quantification are handled using corresponding constructions for $\omega$-VPAs. The translation from $\omega$-VPA to MSO$_\mu$ uses the same encoding for capturing the runs of the automaton by a formula, only the conjunct corresponding to the acceptance requirement for the run needs to be modified. ■

The emptiness problem is decidable in polynomial time [BS92] and we can show that the universality and inclusion problems are EXPTIME-complete.

The connection to regular tree languages also extends to the infinite-word setting. Given a word $\alpha \in \Sigma^\omega$, we can associate with it a unique tree, $\eta(\alpha)$, where matching calls and returns occur together with the segment in between them encoded as a finite tree on the 0-child of the call. This class of infinite trees has the property that the right-most path (the path going down from the root obtained by taking the 1-child whenever it exists and taking the 0-child otherwise) is the only infinite path in the tree. We can define the class of regular *stack*-trees using Büchi tree automata on trees[Tho90].[1] We can then show:

**Theorem 11 (Relation to regular stack-trees)** *Let $\mathcal{L}$ be a set of infinite stack trees over $\widetilde{\Sigma}$. Then $\eta^{-1}(\mathcal{L})$ is an $\omega$-VPL over $\widetilde{\Sigma}$ iff $\mathcal{L}$ is regular.*

**Proof:** Let us consider translation from a Büchi VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ to a Büchi tree automaton $\mathcal{A}$. The construction of Theorem 5 requires some modifications: we need to account for the fact one path is infinite, and thus, has no "end" state; and we need to keep track of whether or not the runs corresponding to call subtrees encounter a Büchi state. More precisely, the set of states for the tree automaton is $P = Q \cup (Q \times (\Gamma \setminus \{\perp\}) \times \{0,1\}) \cup (Q \times Q \times \{0,1\}) \cup (Q \times Q \times (\Gamma \setminus \{\perp\}))$. A state $q$ means that the current node is on the (unique) infinite path in the tree, and the VPA state is $q$. A state $(q, q', b)$ means that the current state of the VPA is $q$, $q'$ is the guessed state where the run of the VPA will be when it meets the next unmatched return, and if the bit $b$ is set to 1, then this run is required to visit a Büchi state in $Q_F$. A state $(q, q', \gamma)$ records the current state $q$, the top symbol $\gamma$, and expects the current node to be a return leaf that can lead to the state $q'$. Finally, a state $(q, \gamma, b)$ means that the current node is on the infinite path, it must be a return, $q$ is the current state of the VPA, $\gamma$ is the current top symbol, and $b$ records whether the corresponding left subtree was required to encounter a Büchi state. The initial states are $Q_{in}$, and the Büchi set for the infinite path in the tree is $Q_F \cup (Q \times \Gamma \times \{1\})$.

In state $q$ while reading a call symbol $a$, for each push-transition $(q, a, q_1, \gamma)$ of $M$, the $\Delta_{01}$ relation for the tree automaton (the case corresponding to "Calls that return") contains, for every $q' \in Q$ and $b \in \{0,1\}$, $(q, a, (q_1, q', b), (q', \gamma, b))$. Thus, the tree automaton guesses the state $q'$ at the end of the run corresponding to the left subtree as well as whether this run visits a state in $Q_F$. If $b$ is set to 1, then in the left child, it is treated as an obligation to visit a Büchi state, and in the right child it is treated as an accepting state for the accepting condition on the infinite path in the tree. In a state $(q, \gamma, b)$, while reading $a$, $a$ must be a return, and the state is updated to $q'$ according to some pop-transition $(q, a, \gamma, q')$ of $M$. In state $(q, q', b)$, while reading a call symbol $a$, for each push-transition $(q, a, q_1, \gamma)$ of $M$, the $\Delta_{01}$ relation for the tree automaton (the case corresponding to "Calls that return") contains, for every $q_2 \in Q$, $((q, q', b), a, (q_1, q_2, b'), (q_2, q', \gamma))$. If $b = 0$ then $b'$ is 0, and if $b = 1$ then either $b'$ is 1 or $q'$ is in $Q_F$. The remaining details are analogous to the finite case.

The construction from tree automata to VPA does not require any noteworthy modifications. ■

[1]Usually regular classes of trees are defined using Muller conditions as they are more powerful than Büchi conditions; however, since we deal with trees that have only one infinite path, the two definitions coincide.

## 7 Conclusions

The class of visibly pushdown languages proposed in this paper explains, unifies, and extends the known classes of properties that can be algorithmically checked against pushdown models of recursive programs. While it requires the model to render its calls and returns visible, this seems natural for formulating program analysis questions. We have shown the class to be *robust* with many of the desirable properties as those of regular languages. Based on our results, we believe that this is indeed a basic class that can be useful in contexts other than software verification. For instance, an XML document can be viewed as a word over a pushdown alphabet where the opening and closing tags are matching calls and returns, and attributes are local actions, and a Vpa can be used to describe the set of XML documents satisfying a query.

There has already been some research that builds on this paper. Games on pushdown structures where the winning condition is an $\omega$-Vpl have been studied and shown to be decidable [LMS04]. A characterization of Vpls using syntactic congruences on words has also been developed, and used for minimizing Vpas [AKMV05]. The decidability result for contextual equivalence for third order Idealized Algol also uses the results of this paper [MW05]. Finally, visibly pushdown versions of branching-time logics and fixpoint calculi are also under investigation [ACM05].

**Acknowledgments**

We thank Kousha Etessami and Mihalis Yannakakis for fruitful discussions.

## References

[ABB97] J. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In *Handbook of Formal Languages*, volume 1, pages 111–174. Springer, 1997.

[ABE$^+$05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems*, 27(4):786–818, 2005.

[ACM05] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. 2005.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04: Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, LNCS 2988, pages 467–481. Springer, 2004.

[AKMV05] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *Automata, Languages and Programming: Proceedings of the 32nd ICALP*. Springer, 2005.

[BB02] J. Berstel and L. Boasson. Balanced grammars and their languages. In *Formal and Natural Computing: Essays Dedicated to Grzegorz Rozenberg*, LNCS 2300, pages 3–25. Springer, 2002.

[BEM97] A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97: Concurrency Theory, Eighth International Conference*, LNCS 1243, pages 135–150. Springer, 1997.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR'92: Concurrency Theory, Third International Conference*, LNCS 630, pages 123–137. Springer, 1992.

[BSW03] A. Bouquet, O. Serre, and I. Walukiewicz. Pushdown games with unboundedness and regular conditions. In *FSTTCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd International Conference*, LNCS 2914, pages 88–99. Springer, 2003.

[CDG$^+$02] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft, Available at `http://www.grappa.univ-lille3.fr/tata/`, 2002.

[CDT02] T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a $\sigma_3$ winning condition. In *Proceedings of the 11th Annual Conference of the European Association for Computer Science Logic, CSL 2002*, LNCS 2471, pages 322–336. Springer, 2002.

[CG77] R.S. Cohen and A.Y. Gold. Theory of omega-languages. i. characterizations of omega-context-free languages. *Journal of Computer and System Sciences*, 15(2):169–184, 1977.

[CMM$^+$04] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt driven programs. *Information and Computation*, 194(2):144–174, 2004.

[CW02] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[EKS03] J. Esparza, A. Kucera, and S. S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003.

[HJM$^+$02] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[JMT99] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[Knu67] D.E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11(3):269–289, 1967.

[LMS04] C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference*, LNCS 3328, pages 408–420. Springer, 2004.

[LST94] A. Lautemann, T. Schwentick, and D. Thérien. Logics for context-free languages. In *Proceedings of Computer Science Logic, 8th International Workshop, CSL 94*, LNCS 933, pages 205–216. Springer, 1994.

[McN67] R. McNaughton. Parenthesis grammars. *Journal of the ACM*, 14(3):490–500, 1967.

[MW05] A.S. Murawski and I. Walukiewicz. Third-order Idealized Algol with iteration is decidable. In *FOSSACS 2005: Foundations of Software Science and Computational Structures, 8th International Conference*, LNCS 3441, pages 202–218. Springer, 2005.

[RHS95] T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[Saf88] S. Safra. On the complexity of $\omega$-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, 1988.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, 1990.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.