June 2004

Visibly Pushdown Languages

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

P. Madhusudan
University of Pennsylvania

Recommended Citation

Rajeev Alur and P. Madhusudan, "Visibly Pushdown Languages", *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC '04)* , 202-211. June 2004. http://dx.doi.org/10.1145/1007352.1007390

Visibly Pushdown Languages

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 two alternative characterizations: a logical characterization using the monadic second order (MSO) theory over words augmented with a binary matching predicate, and a correspondence to regular tree languages. 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.

Keywords
Context-free languages, pushdown automata, verification, logic, regular tree languages, omega-languages, algorithms

Visibly Pushdown Languages *

Rajeev Alur P. Madhusudan

Department of Computer and Information Science
University of Pennsylvania
{ alur, madhusudan }@cis.upenn.edu

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 two alternative characterizations: a logical characterization using the monadic second order (MSO) theory over words augmented with a binary matching predicate, and a correspondence to regular tree languages. 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.

*Supported in part by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382.

STOC'04, June 13–15, 2004, Chicago, Illinois, USA.

Categories and Subject Descriptors

F.1.1 [**Computation by abstract devices**]: Models of Computation—Automata; F.3.1 [**Logics and meanings of programs**]: Specifying and verifying and reasoning about programs; F.4.3 [**Mathematical logic and formal languages**]: Formal languages—Classes defined by grammars or automata, Decision problems

General Terms

Languages, Verification, Algorithms, Theory

Keywords

Context-free languages, pushdown automata, verification, logic, regular tree languages, ω-languages

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 φ (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 φ 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 been extensively studied in recent years leading to efficient implementations and applications to program analysis [21, 5, 6, 3, 15, 14, 13].

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 [17, 13, 14, 12]. 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" [2]. 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 internal 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 *internal* 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 two 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 MSO_μ is expressively equivalent to 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 ω-regular languages is well developed with most of the counterparts of the results for regular languages (c.f. [23, 24]). Consequently, we also define VPAs augmented with acceptance conditions such as *Büchi* and *Muller*, that accept visibly pushdown ω-languages. We establish that the resulting class ω-VPL is closed under union, intersection, renaming, and complementation. Decision problems for ω-VPAs have the same complexity as the corresponding problems for VPAs. As in the finite case, the class ω-VPL can be characterized by regular languages of infinite trees with exactly one infinite path, as well as by MSO_μ. 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 ω-VPA, and we prove that no deterministic Muller ω-VPA accepts this language. However, we show that nondeterministic Büchi ω-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 [17] which proposes a logic over stack contents to specify security constraints, and in [14] 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 [2]. There is an extensive literature on pushdown automata, context-free languages, deterministic pushdown automata, and context-free ω-languages (c.f. [1]). The most related work is McNaughton's parenthesis languages with a decidable equivalence problem [20]. 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 [18]. 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 [4]. 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 [16].

Finally, there is a logical characterization of context free languages using quantifications over matchings [19]. Also, properties expressing boundedness of stack, and repeatedly boundedness, have received a lot of attention recently [10, 8].

2. VISIBLY PUSHDOWN LANGUAGES

2.1 Definition via pushdown automata

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

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 internal 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 visibly pushdown automaton on finite words over $\langle \Sigma_c, \Sigma_r, \Sigma_{int} \rangle$ 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, Γ is a finite stack alphabet that contains a special bottom-of-stack symbol $\perp$, $\delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\perp\})) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_{int} \times Q)$, and $Q_F \subseteq Q$ is a set of final states.*

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

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

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

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

[Internal] If a_i is an internal 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, i.e. 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 internal actions; however this is not a restriction as for any VPA with stack alphabet Γ that can read the top of the stack, one can build a VPA (with a stack alphabet $\Gamma' = (\Gamma \times \Gamma) \cup \{\perp\}$) 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, ϵ-transitions are not allowed.

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 Σ. However, visibly pushdown languages are a strict subclass of context-free languages. For example, the language $\{a^n b a^n \mid n \in \mathbb{N}\}$ is not visibly pushdown for any partition of the alphabet $\Sigma = \{a, b\}$ into calls, returns and internal actions. However, notice that $\{a^n b^n \mid n \in \mathbb{N}\}$ is visibly pushdown if $\Sigma_c = \{a\}$ and $\Sigma_r = \{b\}$, but not otherwise.

For every context-free language, we can associate a visibly pushdown language over a different alphabet in the following way. Let P be a pushdown automaton over Σ and let us assume that on reading any letter, P pushes or pops at most one letter. Let $\Sigma_c = \Sigma \times \{c\}$, $\Sigma_r = \Sigma \times \{r\}$ and $\Sigma_{int} = \Sigma \times \{int\}$. Now consider the visibly pushdown automaton over $\langle \Sigma_c, \Sigma_r, \Sigma_{int} \rangle$ 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, int). Then a word $w = a_1 a_2 \dots a_k$ is accepted by P iff there is some augmentation w' of w, $w' = (a_1, b_1)(a_2, b_2) \dots (a_k, b_k)$, where each $b_i \in \{c, r, int\}$, such that w' is accepted by M. Thus M accepts the words in $L(P)$ annotated with information on how P handles the stack.

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 $(\Sigma_c, \Sigma_r, \Sigma_{int})$, and associate a symbol with every transition of P with the restriction that calls are mapped to Σ_c, returns are mapped to Σ_r, and all other statements are mapped to Σ_{int}. 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 [17, 14, 2] 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 $f(\Sigma_{int}) \subseteq \Sigma'_{int}$. A renaming f is extended to words over Σ 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 γ_1 and M_2 pushes γ_2, then M pushes (γ_1, γ_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^*. Essentially, the automaton will break the word into arbitrarily many subwords nondeterministically and simulate the automaton for L on each such word. To handle the stack correctly, it must augment the stack alphabet so that it can demark positions on the stack corresponding to different words. While this demarking can be disturbed because of pop operations, the automaton can detect this and keep track of this event in its finite control and push the demarker again at the next push operation. □

Note that the restriction that f maps calls to calls, returns to returns, and internal actions to internal actions, is important for closure of VPLs under renaming. VPLs are not closed under unrestricted renaming functions. Indeed, if Σ' is any alphabet and $\widetilde{\Sigma} = \langle \Sigma' \times \{c\}, \Sigma' \times \{r\}, \Sigma' \times int \rangle$, and $f : \Sigma \to \Sigma'$ is an unrestricted renaming function which maps each (a, s) to a (where $s \in \{c, r, int\}$), then, using the construction we sketched earlier, we can show:

PROPOSITION 1. *Let L' be a context-free language over Σ'. Then, there is a visibly pushdown language L over $\widetilde{\Sigma}$ such that $f(L) = L'$.*

PROOF. *Take a pushdown automaton accepting L' and change it to a visibly pushdown automaton over $\widetilde{\Sigma}$, by restricting the transitions according to the input read.* □

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 $a \in \Sigma_{int}$, 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 w_1, 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 $(S_2, R_2, a_2)(S_1, R_1, a_1)\perp$ 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 $\perp$ to q' with stack $\perp$. 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 internal 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 Γ' is the set of elements (S, R, a), where $(S, R) \in Q'$ and $a \in \Sigma_c$. The transition relation δ' is given by:

(Internal) For every $a \in \Sigma_{int}$, $((S, R), a, (S', R')) \in \delta'$ 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 $Update = \{(q,q') \mid \exists q_1,q_2 \in Q, \gamma \in \Gamma : (q,a',q_1,\gamma) \in \delta, (q_1,q_2) \in S, (q_2,a,\gamma,q') \in \delta\}$. Then, $S'' = \{(q,q') \mid \exists q_3 : (q,q_3) \in S', (q_3,q') \in Update\}$ and $R'' = \{q' \mid \exists q \in R', (q,q') \in Update\}$.
- 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 Σ^*. 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 Σ^* 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 [5] 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 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 Σ^* 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.

	Closure under				
	$\cup$	$\cap$	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

	Decision problems for automata		
	Emptiness	Univ./Equiv.	Inclusion
Regular	NLOG	PSPACE	PSPACE
CFL	PTIME	Undecidable	Undecidable
DCFL	PTIME	Decidable	Undecidable
VPL	PTIME	EXPTIME	EXPTIME

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 μ.

Fix $\widetilde{\Sigma}$. A word w over Σ 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 μ 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 $X, Y, \ldots$.

Then the *monadic second-order logic* (MSO_μ) 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 Σ. 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 $\varphi = (\forall x.\ Q_a(x) \Rightarrow \exists y.(Q_b(y) \wedge \mu(x, y)))$ says "every call must have a corresponding return".

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

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

PROOF. The proof follows a similar style as in proving that MSO (without μ) over words defines the same class as that of regular words (see [23]).

First we show that for any sentence φ, $L(\varphi)$ 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 $\Sigma_s^Z = \{(a, V) \mid a \in \Sigma_s, V : Z \to \{0, 1\}$ is a valuation function$\}$, where $s \in \{c, r, int\}$. Then a word w' over Σ^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 Σ^Z such that the underlying word w satisfies ψ 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_γ and R_γ stand for the positions where γ is pushed and popped from the stack, respectively. We can write conditions in φ that ensure that the variables X_q, C_γ and R_γ indeed define a run; the only interesting detail here is to ensure that when a stack symbol γ is pushed (i.e. when C_γ holds), at the corresponding return R_γ must hold. We can state this using the μ-relation by demanding that for every x and y, if $\mu(x, y)$ holds, then there is a γ such that $x \in C_\gamma$ and $y \in R_\gamma$. Also, φ 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 ϵ 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 [9]. 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 Σ-labeled 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 $x, x.i \in V$, $i \in \{0, 1\}$; ϵ is the root of the tree. Let $\mathcal{T}_\Sigma$ denote the set of all Σ-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 an internal 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 Σ^+ 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 η is a 1-1 correspondence between words in Σ^+ 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 $w[i]$).

Let us now define regular tree languages using automata over trees. A tree-automaton on Σ-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—Δ_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 Σ-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 $(\lambda_\rho(v), \lambda(v), \lambda_\rho(v.0), \lambda_\rho(v.1)) \in \Delta_{01}$
- 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 Σ-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 $P = (Q \times Q) \cup (Q \times Q \times (\Gamma \setminus \{\perp\}))$ 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', γ) 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 γ.

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, γ) is a transition in the VPA, the tree automaton can split into two copies (q_1, q_2) and (q_2, q', γ) 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', γ) will see the return, simulate a transition from q_2 that pops γ and continue. Note that the tree automaton is in general nondeterministic, even if the VPA is deterministic.

The transition relation Δ 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 $\gamma \neq \perp$, we have, for every $q' \in Q$, the transitions $((q, q', \gamma), a, (q_1, q')) \in \Delta_1$

Internal actions or returns without matching calls: If $a \in \Sigma_{int}$ and $(q, a, q_1) \in \delta$, or, $a \in \Sigma_r$ and $(q, a, \perp, q_1) \in \delta$, 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 $(q, a, \perp, q') \in \delta$. 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 $Q = ((P \cup \{acc\}) \times (P \cup \{*, \perp\})) \cup \{\mathit{fin}\}$, $Q_{in} = \{(p, \perp) \mid p \in P_{in}\}$ and $\Gamma = ((P \cup \{*\}) \times (P \cup \{*, \perp\})) \cup \{\perp\}$.

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 $(p_n, p'_n)(p_{n-1}, p'_{n-1}) \dots (p_1, p'_1)\perp$ where $p'_1 = \perp$ 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:

Internal actions: For every $a \in \Sigma_{int}$, $(p, a, p_1) \in \Delta_1$, $p' \in P \cup \{*, \perp\}$, we have $((p, p'), a, (p_1, p')) \in \delta$.
For every $a \in \Sigma_{int}$, $(p, a) \in \Delta_\emptyset$, we have $((p, *), a, \mathit{fin}) \in \delta$, $((p, \perp), a, \mathit{fin}) \in \delta$ 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 $p' \in P \cup \{*, \perp\}$, 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 $Q_F = \{\mathit{fin}\}$. □

5. VISIBLY PUSHDOWN ω-LANGUAGES

We now consider extensions of the results in the previous sections to infinite words over $\widetilde{\Sigma}$. An ω-VPA is a tuple $M = (Q, Q_{in}, \Gamma, \delta, \mathcal{F})$ where Q, Q_{in}, Γ and δ are as in a VPA. For any infinite word $\alpha \in \Sigma^\omega$, a run is an ω-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 ρ is accepting, we consider the set $\mathit{inf}(\rho) \subseteq Q$ which is the set of all states that occur in ρ 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 ρ 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 ρ 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 α is accepted by M if there is some accepting run of M on α. The language of M, $L_\omega(M)$, is the set of all ω-words that it accepts. A language of infinite words $L \subseteq \Sigma^\omega$ is said to be an ω-VPL if there is some ω-VPA M such that $L = L_\omega(M)$.

The class of ω-regular languages is well studied and is defined using automata (without stack) on infinite words; the definition of ω-VPA extends this definition to visibly pushdown automata and the languages accepted by ω-VPAs is a subclass of ω-CFLs [11]. 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 ω-VPAs are equivalent. Hence we take the definition of an ω-VPL as that of being accepted by a nondeterministic Büchi ω-VPA. The notion of renaming can be extended to infinite words and we can show:

THEOREM 6 (CLOSURE). *Let L_1 and L_2 be ω-VPLs with respect to $\widetilde{\Sigma}$. Then, $L_1 \cup L_2$ and $L_1 \cap L_2$ are also ω-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 ω-VPLs over $\widetilde{\Sigma}$.*

However, ω-VPAs cannot be determinized. For automata (without stack) on ω-words it is well known that for any nondeterministic Büchi automaton, there is a deterministic Muller automaton that accepts the same language [22, 23]. However, this is not true for visibly pushdown automata over infinite words. In fact, consider the language L_{repbdd} consisting of all words α 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 α infinitely often is less than or equal to n [10]. We can then show that this language is not determinizable.

THEOREM 7 (DETERMINISTIC VS. NONDET. ω-VPAS). *ω-VPAs are not determinizable. In particular, the language L_{repbdd} is an ω-VPL but cannot be accepted by any deterministic Muller ω-VPA.*

PROOF. We can easily design a nondeterministic Büchi ω-VPA that accepts L_{repbdd}. The ω-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 ω-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 ω-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, σ) to (q', σ) for any stack σ). 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^ω 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 ω-VPAs cannot be determinized, they can be complemented.

THEOREM 8 (COMPLEMENTABILITY). *The class of ω-VPLs over $\widetilde{\Sigma}$ is closed under complement.*

PROOF. Let $M = (Q, Q_{in}, \Gamma, \delta, F)$ be a ω-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 α 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 $\alpha = a_0 a_1 w_0 a_2 w_1 w_2 a_3 \ldots$ 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 internal 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 α: $\alpha' = a_0 a_1 S_0 a_2 S_1 S_2 a_3 \ldots$ 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 ρ of M on w_i starting at the state q and ending at the state q' and ρ 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 $PR = (\Sigma_r \cup \Sigma_{int} \cup \mathcal{S})^\omega \cup (\Sigma_r \cup \Sigma_{int} \cup \mathcal{S})^*.(\Sigma_c \cup \Sigma_{int} \cup \mathcal{S})^\omega$ 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 Σ-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 α is accepted by M iff the pseudo-run corresponding to α 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 ([23]).

We now construct a nondeterministic Muller ω-VPA that, on reading α, generates the pseudo-run of α online and checks whether it is in L_R. The factorization of α 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 ω-VPL using MSO_μ which is now interpreted over infinite words, using the fact that ω-VPLs are closed under union, complement and renaming:

THEOREM 9 (MSO_μ-CHARACTERIZATION). *A language L of infinite strings over $\widetilde{\Sigma}$ is an ω-VPL iff there is an MSO_μ sentence φ over $\widetilde{\Sigma}$ that defines L.*

The emptiness problem is decidable in polynomial time [7] 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[23].[1] We can then show:

THEOREM 10 (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 ω-VPL over $\widetilde{\Sigma}$ iff $\mathcal{L}$ is regular.*

6. 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 have many applications and warrants further investigation. A characterization via restricted context-free grammars should be feasible, but has not been explored here. In the case of ω-languages, the gap between deterministic and nondeterministic variants raises new questions. On the logical side, identifying expressively complete fragments of MSO_μ (and its first-order counterpart) that can be model checked at low cost would be fruitful. Finally, it would be interesting to study games on pushdown structures where the winning condition is an ω-VPL.

Acknowledgment: We thank Kousha Etessami and Mihalis Yannakakis for fruitful discussions.

[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. REFERENCES

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

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. To appear in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS'04, Spain*, LNCS 2988. Springer, 2004.

[3] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of the 13th International Conference on Computer Aided Verification*, LNCS 2102, pages 207–220. Springer, 2001.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] A. Bouquet, O. Serre and I. Walukiewicz. Pushdown games with unboundedness and regular conditions. To appear in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, December 2003.

[9] 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.

[10] T. Cachat, J. Duparc and W. Thomas. Solving pushdown games with a Σ_3 winning condition. In Proceedings of the 11th Annual Conference of the European Association for Computer Science Logic, CSL 2002, volume 2471 of Lecture Notes in Computer Science, pages 322-336. Springer, 2002.

[11] R.S. Cohen and A.Y. Gold. Theory of omega-Languages. I. Characterizations of omega-Context-Free Languages. *JCSS* 15(2): 169–184, 1977.

[12] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt driven programs. In *Proceedings of the 10th International Symposium on Static Analysis*, volume LNCS 2694, pages 109–126, 2003.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] C. Lautemann, T. Schwentick and D. Thérien. Logics For context-free languages. In *Proceedings of Computer Science Logic, 8th International Workshop, CSL '94, Poland*, LNCS 933, pages 205–216. Springer, 1994.

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

[21] 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.

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

[23] 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.

[24] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.