Pumping Visibly Pushdown Languages*

STEFAN D. BRUDA
Department of Computer Science
Bishop's University
2600 College St
Sherbrooke, Quebec J1M 1Z7
CANADA
stefan@bruda.ca

Abstract: Visibly pushdown languages are a subclass of context-free languages and are particularly well suited for specification and verification of application software. We find that, in addition to the pumping theorem inherited from context-free languages, visibly pushdown languages have some specific pumping properties. These properties have consequences in the recursive constructs of a process algebra based on visibly pushdown languages.

Key-words: Visibly pushdown languages, Visibly pushdown automata, Visibly pushdown grammars, Pumping properties, Formal methods, Process algebrae

1 Introduction

The formal verification arena has been enhanced by the recent introduction of the class of visibly pushdown languages (VPL) [2], a subclass of context-free languages. VPL are particularly well suited for modelling software analysis, and they are also tractable and robust like the class of regular languages (and therefore they have almost all the prerequisites to support a fully compositional process algebra).

Visibly pushdown languages are accepted by visibly pushdown automata (vPDA) whose stack behaviour is determined by the input symbols. A visibly pushdown automaton operates on a word over an alphabet that is partitioned into three disjoint sets of call, return, and local symbols. Any input symbol can change the control state but call and return symbols also change the stack content: While reading a call symbol the automaton must push one symbol on the stack, and while reading a return symbol it must pop one symbol (unless the stack is empty).

VPL are closed under intersection, union, complementation, renaming, concatenation and Kleene star, just like regular languages. A number of decision problems such as universality, language equivalence and language inclusion, which are not decidable for context-free languages, become EXPTIME-complete for VPL. Visibly pushdown languages seem quite natural for verification of pre/post conditions or for inter-procedural flow properties. In particular, requirements that can be verified in this manner include all regular properties but also non-regular properties such as partial correctness, total correctness, local properties, access control, and stack limits [1].

Our previous (and ongoing) work includes an algebraic approach to program specification and verification based on VPL [4] or related formalisms [5]. In the process, we developed several results regarding the structure of words in a VPL. We believe that such insights are useful to the research community at large, so we summarize them in this paper.

The contribution of this work is thus a deeper characterization of the structure of VPL. The most interesting properties specified using a vPDA-based process algebra will undoubtedly be recursive in a non-regular sense (for indeed, regular recursion has been present in finite-state process algebrae for a long time). We will call such a recursion "self-embedding" (see Section 3 for details). Since this is going to be the most important feature of any application of VPL, we study the limits of self-embedding recursion. We find a pumping result for visibly pushdown languages which illustrates the structure of recursive visibly pushdown automata-based processes and has also significant practical consequences.

2 Preliminaries

Let $\widetilde{\Sigma} = \Sigma_c \uplus \Sigma_r \uplus \Sigma_l$ be a partition over some alphabet Σ. For convenience, we put $\Sigma_m = \Sigma_c \cup \Sigma_r$. Given

*This research was supported by the Natural Sciences and Engineering Research Council of Canada.

ISBN: 978-1-61804-028-2

some word $w \in A^*$ and some $A' \subseteq A$, we denote by $w_{A'}$ the restriction of w on A' (that is, $w_{A'}$ is obtained by erasing from w all the symbols outside A'). We denote the empty word, and only the empty word by ε.

A visibly pushdown automaton (vPDA) [2] is a tuple $M = (\Phi, \Phi_{in}, \widetilde{\Sigma} = \Sigma_c \uplus \Sigma_r \uplus \Sigma_l, \Gamma, \Omega, \Phi_F)$, where Φ is a finite set of states, $\Phi_{in} \subseteq \Phi$ is a set of initial states, $\Phi_F \subseteq \Phi$ is the set of final states, Γ is the (finite) stack alphabet that contains a special bottom-of-stack symbol $\perp$, and Ω is the transition relation $\Omega \subseteq (\Phi \times \Gamma^*) \times \widetilde{\Sigma} \times (\Phi \times \Gamma^*)$. Σ_l is the set of local symbols, Σ_c is the set of call symbols and Σ_r is the set of return symbols.

Every tuple $((P, \gamma), a, (Q, \delta)) \in \Omega$ (also written $(P, \gamma) \xrightarrow{a} (Q, \delta) \in \Omega$) must have the following form: if $a \in \Sigma_l \cup \{\varepsilon\}$ then $\gamma = \delta = \varepsilon$, else if $a \in \Sigma_c$ then $\gamma = \varepsilon$ and $\delta = a'$ (where a' is the stack symbol pushed for a), else if $a \in \Sigma_r$ then if $\gamma = \perp$ then $\gamma = \delta$ (hence visibly pushdown automata allow unmatched return symbols) else $\gamma = a'$ and $\delta = \varepsilon$ (where a' is the stack symbol popped for a).

In other words, a local symbol is not allowed to modify the stack, while a call always pushes one symbol on the stack. Similarly, a return symbol always pops one symbol off the stack, except when the stack is already empty. Note in particular that empty transitions (that is, transitions that do not consume any input) are allowed but are not permitted to modify the stack [2].

Whenever we have a pair of symbols c and r such that $c \in \Sigma_c$, $r \in \Sigma_r$, and $(P, \varepsilon) \xrightarrow{c} (Q, a), (R, a) \xrightarrow{r} (S, \varepsilon) \in \Omega$, the two symbols are called *matched*. A call (or return) that has no matched return (or call) is called *unmatched*. Note that some call or return can be both matched and unmatched at the same time in a given visibly pushdown automaton.

The notion of run, acceptance, and language accepted by a visibly pushdown automaton are defined as usual: A run of a visibly pushdown automaton M on some word $w = a_1 a_2 \ldots a_k$ is a sequence of configurations $(q_0, \gamma_0)(q_{01}, \gamma_0) \cdots (q_{0m_0}, \gamma_0)(q_1, \gamma_1)$ $(q_{11}, \gamma_1) \cdots (q_{1m_1}, \gamma_1)(q_2, \gamma_2) \cdots (q_k, \gamma_k)(q_{k1}, \gamma_k) \cdots$ (q_{km_k}, γ_k) such that $\gamma_0 = \perp$, $q_0 \in \Phi_{in}$, $(q_{ij-1}, \varepsilon) \xrightarrow{\varepsilon} (q_{ij}, \varepsilon) \in \Omega$ for all $1 \leq i \leq k$, $1 \leq j \leq m_i$, and $(q_{i-1m_{i-1}}\gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Omega$ for every $1 \leq i \leq k$ and for some prefixes γ'_{i-1} and γ'_i of γ_{i-1} and γ_i, respectively. Whenever $q_{km_k} \in \Phi_F$ the run is accepting; M accepts w iff there exists an accepting run of M on w. The language $\mathcal{L}(M)$ accepted by M contains exactly all the words w accepted by M.

It is possible for a call or a return which is both matched and unmatched in a visibly pushdown automaton to have only one characteristic in a particular word accepted by the visibly pushdown automaton (i.e., be either matched, or unmatched, but not both in that word). A word that is accepted by some visibly pushdown automaton can be *balanced* (meaning that all the calls and returns are matched). In addition, we say that a word is *call-balanced* if it has no unmatched calls and *return-balanced* if it has no unmatched returns. Note in passing that w is balanced [call-balanced, return-balanced] iff w_{Σ_m} is balanced [call-balanced, return-balanced].

A context-free grammar [7] is a tuple $G = (\Sigma, V, S, R)$. Σ and V are the set of terminals and nonterminals, respectively. $S \in V$ is the axiom, and $R \subseteq V \times (\Sigma \cup V)^*$ is the set of rewriting rules; a rule (A, w) is commonly written $A \rightarrow w$. The semantics of a grammar is given by the rewriting operator $\Rightarrow$ defined as follows: for any $u, v \in \Sigma \cup V$, $uAv \Rightarrow uwv$ iff $A \rightarrow w \in R$. The language $\mathcal{L}(G)$ generated by a grammar G is the set of exactly all the words $w \in \Sigma^*$ such that $S \Rightarrow^* w$, where $\Rightarrow^*$ denotes as usual the reflexive and transitive closure of $\Rightarrow$.

The pumping theorem for context-free languages (specifying that certain portions of words in a context-free language can be "pumped" as desired) is stated as follows:

Proposition 1 [7] *For any context-free language L there exists a constant n such that any word $w \in L$, $|w| > n$ can be written as $w = uvtxy$ with $vx \neq \varepsilon$ and $uv^ktx^ky \in L$ for every $k \geq 0$.* ■

A regular grammar is a context-free grammar with all the rules taken from the set $V \times (\Sigma^*(V \cup \{\varepsilon\}))$. Languages generated by regular grammars are called regular languages [7].

A visibly pushdown grammar (VPG) [2, 6] is a context-free grammar $G = (\widetilde{\Sigma}, V = V_0 \uplus V_1, S, R = R_\varepsilon \uplus R_{reg} \uplus R_{bal})$, where $S \in V$, and the set of rewriting rules R is the union of the following sets:

$$\begin{aligned}
R_\varepsilon &\subseteq \{X \rightarrow \varepsilon : X \in V\} \\
R_{reg} &\subseteq \{X \rightarrow aY : X, Y \in V, X \in V_0 \text{ implies} \\
&\qquad a \in \Sigma_l \text{ and } Y \in V_0\} \\
R_{bal} &\subseteq \{X \rightarrow aYbZ : X, Z \in V, a \in \Sigma_c, b \in \Sigma_r, \\
&\qquad Y \in V_0, X \in V_0 \text{ implies } Z \in V_0\}
\end{aligned}$$

ISBN: 978-1-61804-028-2

A language is accepted by a vPDA iff it is generated by a VPG [2].

3 Some Pumping Properties of VPL

It is well known that any context-free language can be transformed into a visibly pushdown language by determining a suitable partition of the underlying alphabet [2]. Therefore, finding a pumping theorem for visibly pushdown languages in the same spirit as for context-free or regular languages [7] is not necessary. However, pumping results borrowed from context-free languages do not say anything about those cases in which the partition is already in place. In these cases it turns out that we can establish more pumping properties.

We first note that an infinite visibly pushdown language can be accepted only by a recursive visibly pushdown automaton. The term "recursive" is borrowed here from other areas (such as grammars or process algebrae) for convenience, but the term should be intuitive: A recursive visibly pushdown automaton has one recursive state (or more), that can be encountered infinitely often during an accepting run. A recursive state generates a family of runs, which in turn accept an infinite subset of the language. Given the nature of visibly pushdown automata, recursion can take two forms.

Suppose that a recursive run can encounter the same configuration (q, γ) infinitely often. We are then talking about *regular recursion*, which is similar to the recursion encountered in finite automata. Such a recursion satisfies all the properties known from regular languages (including the pumping theorem for regular languages) and is thus not very interesting. We will not consider this kind of recursion any further, for indeed it does not generate new issues over the ones already studied for regular languages; the stack is always bounded by a constant for regular recursion.

More generally, regular recursion occurs whenever a run encounters an infinite sequence of configurations $(q, \gamma_1)(q, \gamma_2)(q, \gamma_3)\cdots$ such that $|\gamma_i| \geq |\gamma_j|$ whenever $i > j$: Whenever the stack increases but does not decrease, the stack does not participate in the acceptance of the language (and can indeed be eliminated altogether by simple modifications). Now the stack grows unboundedly, but for all practical purpose we end up with the same kind of recursion (regular).

Suppose now that a recursive run can encounter an infinite sequence of configurations $(q, \gamma_1)(q, \gamma_2)(q, \gamma_3)\cdots$ that does not describe regular recursion. We call this kind of recursion *self-embedding*. When self-embedding recursion is present, the stack becomes potentially unbounded, but now the stack also plays a role in the acceptance of the input. This kind of recursion can thus create interesting phenomena, some of them explained in what follows.

The term self-embedding recursion is borrowed from context-free languages (the name itself coming from grammars rather than automata).

In grammatical terms, regular recursion is introduced by derivations of the form $A \Rightarrow^* w$ with $|w|_{\{A\}} \neq 0$ using only rules from R_{reg} and self-embedding recursion is introduced by derivations of the form $A \Rightarrow^* w$ with $|w|_{\{A\}} \neq 0$ using rules from R_{bal}.

Regular and self-embedding recursion are usually mixed in the definition of a VPL. Putting both regular and self-embedding recursion together, and based on the definition of VPG, we first establish the following general result about the form of a word in a VPL.

Theorem 2 *Given some VPG $G = (\widetilde{\Sigma}, V = V_0 \uplus V_1, S, R = R_\varepsilon \uplus R_{reg} \uplus R_{bal})$ and some $A \in V$, the words generated by A have the form $w = u_1v_1u_2v_2 \ldots u_nv_nu_{n+1}$ for some $n \geq 0$, where u_i are regular words over Σ and v_i are balanced words over $\widetilde{\Sigma}$.*

Proof. If $A \in V_0$, then w is balanced by the definition of a VPG and so the proof is established (since ε is obviously regular). Indeed, a nonterminal in V_0 can only introduce regular strings of local symbols (using rules from R_{reg}) or matched pairs of call and return symbols (via rules from R_{bal}); any combination of these yield balanced words.

Suppose then that $A \in V_1$. An inductive argument establishes that A yields a word w as above as follows:

1. We begin by generating the regular prefix u_1 of w using rules from R_{reg}. Once we use some rule from R_{bal} we end the generation of u_1. We have $A \Rightarrow^* u_1B_1$ and obviously u_1 is regular. This word can contain call and return symbols, but their matching (if any) is not "remembered" in the grammar.

2. Once we use one rule from R_{bal} the generation of u_1 is complete. Indeed, by applying such a rule

ISBN: 978-1-61804-028-2

we end up with $A \Rightarrow^* u_1aYbA_1 \Rightarrow^* uaybA_1$. y is always balanced because it comes from $Y \in V_0$ (See the case of $A \in V_0$ above), so ayb is balanced. We put $v_1 = ayb$ and so we have $A \Rightarrow^* u_1v_1A_1$.

3. It is however possible that we do not use any rule from R_{bal} and use instead a rule from R_ε to get rid of B_1 thus obtaining u_{n+1}. This ends the derivation as it erases the sole nonterminal in the word.

4. Replace now in Items 1 and 2 above A with A_i, A_1 with A_{i+1}, u_1 with u_i, and v_1 with v_i. We obtain that $A_i \Rightarrow^* u_iv_iA_{i+1}$ and so $A \Rightarrow u_1v_1u_2v_2\ldots u_iv_iA_{i+1}$. A_{i+1} either continues in the same manner (Items 1 and 2), or gets replaced by a string of terminals as per Item 3. At some point however A_{i+1} needs to follow Item 3 for the derivation to end.

It s quite obvious that no other derivation is possible beside the derivations described above. Indeed, in the rules from R_{bal} Y is always in V_0, and for any $X \in V_0$ there is no $Y \in V_1$ such that $X \Rightarrow^* uYw$. That is, a nonterminal in V_0 never yields a nonterminal in V_1 (while the converse is obviously not true). It follows that the only possible derivations are as outlined above. ■

Self-embedding recursion creates the more complex pumping theorem for context-free languages (which is still pertinent for visibly pushdown languages, as mentioned above). However, more specific pumping results can be established for visibly pushdown languages. We start by establishing the form of VPL words that can be pumped.

Theorem 3 *Consider some visibly pushdown automaton M and two words w_1 and w_2 such that $w_1w_2 \in \mathcal{L}(M)$. Then $w_1^nw_2^n \in \mathcal{L}(M)$ for any $n > 0$ only if w_1 is return-balanced, w_2 is call-balanced, neither w_1 nor w_2 are balanced, and the unmatched calls in w_1 as well as the unmatched returns in w_2 are not introduced recursively. Furthermore, $w_1^nw_2^n \in \mathcal{L}(M)$ for any $n > 0$ only if $w_1^nw_2^n$ is balanced for any $n > 0$.*

Proof. After w_1^n is accepted, M needs to remember n (so that to recognize exactly w_2^n afterward). Given the nature of M's storage, n can only be remembered as stack height. Therefore, each and every iteration of w_1 must add χ symbols to the stack for some $\chi > 0$. Conversely, every iteration of w_2 must remove χ symbols from the stack for some $\chi > 0$, and the stack must become empty after the n-th iteration of w_2—indeed, there is no other way to know that we reached n iterations of w_2 than by detecting the emptiness of the stack.

Adding to the stack at every iteration of w_1 clearly happens whenever w_1 is return-balanced but not balanced. If on the other hand w_1 is call-balanced (including w_1 being balanced) then the stack does not increase in any iteration of w_1. Suppose now that w_1 is not balanced in any way and that the number of (unmatched) returns is larger than the number of (unmatched) calls. Then, since the unmatched returns must precede the unmatched calls the stack will increase in the first iteration of w_1; however, in the second (and subsequent) iterations those unmatched returns will pop off the unmatched calls and the stack ceases to grow, which is not acceptable.

In all, w_1 cannot be balanced. In addition, it must be either return-balanced, or having a larger number of (unmatched) calls than (unmatched) returns.

By a similar argument, the stack height of M needs to decrease at every iteration of w_2 and thus w_2 cannot be balanced and must be either call-balanced or having a larger number of returns than calls.

Denote the number of unmatched calls [returns] in w_1 by c_1 [r_1] and the number of unmatched calls [returns] in w_2 by c_2 [r_2]. We have then that $0 \leq r_1 < c_1$ and $r_2 > c_2 \geq 0$.

Consider first the case $n = 1$: We have $c_1 - r_2 + c_2 \leq 0$ (at the end of the day the stack must be empty; the r_1 unmatched returns of w_1 happen when the stack is empty, and thus they will not affect the stack height). Since c_2 is positive, it must be that $c_1 - r_2 \leq 0$, or $c_1 \leq r_2$. From the stack's point of view, this means that c_1 symbols are pushed, then c_1 symbols are popped, then the remaining $r_2 - c_1$ unmatched returns are processed with an empty stack. At this point the stack is empty. M then pushes c_2 symbols; however, at the end of the day the stack needs to be empty, so it is necessary that $c_2 = 0$, that is, w_2 is call-balanced, as desired.

In addition, since $c_2 = 0$, it is also immediate that $c_1 \leq r_2$ (otherwise the stack is not empty at the end), or $c_1 - r_2 \leq 0$.

We now go to larger values of n. Let $d_1 = c_1 - r_1$ and $d_2 = r_2 - c_2 = r_2$. Suppose now that $d_1 > d_2$; then after n iterations of w_1 and further n iterations of w_2 there are symbols left on the stack. It is therefore impossible to stop at this point (since an empty stack is the only possible stopping signal), so the automaton cannot accept $w_1^n w_2^n$. Whenever $d_1 < d_2$, the automaton cannot accept $w_1^{d_2} w_2^{d_2}$ (and thus cannot accept $w_1^n w_2^n$); indeed, the stack becomes empty after d_2 iterations of w_1 followed by d_1 iterations of w_2. The emptiness of the stack being the only stopping condition, there is no way to continue to accept precisely $w_2^{d_1-d_2}$. In all, the only possible variant is that $d_1 = d_2$, that is, $c_1 - r_1 = r_2$.

Now, after the first w_1 is accepted we have c_1 symbols on the stack. After the second w_1 we have $c_1 + c_1 - r_1$ symbols on the stack (the unmatched returns in w_1 will now match r_1 symbols on the stack). After n iterations, the stack height will be $c_1 + (n-1)(c_1 - r_1) = c_1 - (n-1)r_2$ (since $c_1 - r_1 = r_2$). Now comes w_2. Since we already established that $c_2 = 0$, the n copies of w_2 will pop $nr_2 = (n-1)r_2 + r_2$ symbols off the stack. Given the content pushed onto the stack by the iterations of w_1, it follows that the first c_1 symbols pushed must be matched by the last r_2 symbols popped. Unless $r_1 = 0$, some of the last r_2 symbols will be processed as unmatched in the last iteration and as matched earlier. This however looses control over counting the number of occurrences of w_2 (so it is immediate that the automaton cannot accept exactly n occurrences of w_2). Therefore it must be the case that $r_1 = 0$.

In all, we found that w_1 and w_2 cannot be balanced, that w_1 must be return-balanced, and that w_2 must be call-balanced, as desired.

The relation $c_1 - r_1 = r_2$ found earlier now becomes $c_1 = r_2$ (since $r_1 = 0$). Therefore, the string $w_1^n w_2^n$ is balanced for all $n > 0$, again as desired.

That the unmatched calls in w_1 and the unmatched returns in w_2 are not introduced recursively is immediate. Indeed, any recursive construct used to introduce these symbols eliminates the possibility of an automaton to compare c_1 with r_2 on the stack (they are both introduced recursively and so vary arbitrarily), thus eliminating the possibility of recognizing n copies of w_1 followed by exactly n copies of w_2. ■

Once we have the structure established by Theorem 3 we can particularize to some degree the pumping theorem for context-free languages (Proposition 1) to visibly pushdown languages:

Theorem 4 *For any visibly pushdown language L generated by a grammar with no regular recursion there exists a constant n such that any word $w \in L$, $|w| > n$ can be written as $w = uvtxy$, with $vx \neq \varepsilon$, t balanced, v return-balanced, x call-balanced, neither v nor x balanced, and $uv^k tx^k y \in L$ for every $k \geq 0$.*

Proof. Ignoring the partition into call, return, and local symbols every word w as in the theorem can be pumped as established by Proposition 1 (for indeed any VPG is also a context-free grammar). Such a pumping happens because of self-embedding recursion (since no regular recursion is present), so Theorem 3 applies, establishing the structure of pumped strings as specified. ■

4 Conclusions

We presented new insights in the structure of words in visibly pushdown languages via a couple of pumping results.

Our motivation for this paper is one of our active research interests, namely developing a VPL-based theory for specification and verification of application software, such as a VPL-based process algebra. Finite-state algebrae have proven useful for the specification and verification of hardware, communication protocols, and drivers. More complex application software cannot be readily modelled using finite-state mechanisms, as they contain a huge, impractical number of distinct finite states. We therefore believe that an infinite-state process algebra can dramatically open the domain of application software to specification and verification using formal methods (and more specifically algebraic methods such as model-based testing [3]).

In this context, our first pumping result (Theorem 3) is particularly worth noting, as it illustrates the nature of words accepted by self-embedding recursive visibly pushdown automata. Our pumping result shows the necessary conditions for pumping pairs of strings in a VPL. Such pairs can be pumped only if the first string has unmatched calls which are subsequently matched by the unmatched returns in the second string; these unmatched symbols cannot be introduced recursively. Overall, we can only pump calls and returns via self-embedding recursion, and the pair of pumped words must be overall balanced.

That only balanced words can be pumped (and that they split into two parts with specific properties, as detailed in Theorem 3) essentially means that interesting verifiable properties have always an end (a return symbol), which is consistent with the undecidability of the halting problem [7]. Indeed, halting is undecidable, so a non-halting program may not be verifiable; however, a halting program can be readily verified. In addition, calls and returns introduced via regular recursion cannot appear in constructs based on self-embedding recursion, which is consistent with the intended use of calls and returns (as models for function calls and returns).

Practical consequences of such a result however cannot be fully determined until a visibly pushdown automata-based process algebra is developed and deployed in practice; as we said before, this is currently our main research interest.

We are also able to give a pumping theorem for VPL (Theorem 4) that is almost identical to the pumping theorem for context-free languages, but is more restricted in that it excludes regular recursion. Normally regular and self-embedding recursion coexist in a VPG, so the applicability of this result is somehow limited. This result however illustrates once more the structure of self-embedding recursion and also the fact that regular recursion should practically be responsible only for introducing local symbols (which would model the loops is a program). We note that regular recursion for local symbols can probably be introduced in Theorem 4 without affecting the result.

Knowing the word structure of VPL (Theorem 2) has proven very useful for us in determining the possibilities of such languages. We have actively used this property in our previous research, and we believe that it will be useful to the research community at large.

We note that the regular words u_i from Theorem 2 can contain technically any combination of calls and returns. Practically (and in conjunction with the pumping result from Theorem 4) they will only contain local symbols. Finding a tighter formalism that restricts the constructions to only practically meaningful situations and is still closed under prefix and the such (the real reason unmatched calls and returns are needed) is an interesting open problem (which might however be unsolvable; we do not have any idea how such a construct, which is also closed under prefix, is realizable).

Overall, we note that such purely theoretical results such as the ones described in this paper have surprisingly practical consequences.

True, the whole VPL-based algebraic approach to program specification and verification has been shown to be impossible due to some missing closure properties for these languages [5]. The proposed approach based on multi-stack VPL [5] is however similar in structure with VPL, so the results from this paper can be easily extended in such a context, and are especially pertinent to the "single-stack processes" (that is, "single-thread processes") that are modelled using single stacks and then put together via disjoint operations [5]. In particular, we believe that our pumping result can be easily extended to multi-stack VPL.

References:

[1] R. ALUR, K. ETESSAMI, AND P. MADHUSUDAN, *A temporal logic of nested calls and returns*, in Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 04), vol. 2988 of Lecture Notes in Computer Science, Springer, 2004, pp. 467–481.

[2] R. ALUR AND P. MADHUSUDAN, *Visibly pushdown languages*, in Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 04), ACM Press, 2004, pp. 202–211.

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

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

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

[6] S. LA TORRE, M. NAPOLI, AND M. PARENTE, *The word problem for visibly pushdown languages described by grammars*, Formal Methods in System Design, 31 (2007), pp. 265–279.

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