A Fixed-Point Approach towards Efficient Models Conversion

Ş. Andrei, H. Koh

Key Words: *Pushdown automaton; context-free grammar; conversion; fixed-point approach.*

Abstract. *Formal languages have different models of language representations, such as acceptors and generators. This paper refers to pushdown automata as acceptors and to context-free grammars as generators. Some applications are better specified using a model, while other applications work better using another model. Our paper deals with a conversion of an arbitrary pushdown automaton to a context-free grammar based upon a fixed-based approach which results in an equivalent context-free grammar of significantly reduced size. In addition, this conversion turns out to be very conducive if the input pushdown automaton is in a special form, which we introduce as a new class of popping static pushdown automata (PSPDA). We prove that for a PSPDA automaton, our conversion algorithm has a linear time complexity, and the resulting context-free grammar has a linear size of the entire representation of the PSPDA. Moreover, we show that the PSPDA class has the same expressive power as the traditional pushdown automata.*

1. Introduction

A *model* is in general a human construction or a representation of some real world systems. A typical model has an information input, an information processor, and an output of expected results. This paper refers to formal languages [11], an area that uses both acceptors and generators as models to characterize a language. Formal languages [8] have many acceptor models, such as finite automata [15], pushdown automata [1], Turing machines [7], and many generator models, like regular grammars [15], context-free grammars [1], context-sensitive grammars [7], and phrase-structure grammars [7]. In fact, as a different important intermediate model we mention the language equations able to describe an arbitrary formal language [12], although their scope is beyond this paper. Some applications in formal languages work better with acceptor models, while others work better with generator models. Hence, there is a need to translate a result expressed in a model to another target model. However, in order to preserve the complexity results, the conversion between the two models should be done efficiently, e.g., in a polynomial time complexity, preferably in a linear time complexity.

It is very important that these conversions between various models be done efficiently and in a synchronized way. For example, IBM researchers have recently developed a prototype system incorporating avatars (i.e., electronic image representing and interpreting a computer user) that translate speech into the sign language [9]. The SiSi (that means 'Say it, Sign it') system uses first speech-recognition technology to convert a conversation into text. Then the text is passed through a translation software module, which conducts syntactic parsing, lexical analysis, and other processes to convert the content into grammatically correct British Sign Language. The system then sends commands written in the Sign Gesture Markup Language for working with avatars to the computer's user.

One of the most important conversions between two models in formal languages is the one from a pushdown automaton and a context-free grammar [1]. This conversion is important because there exist many algorithms for deciding questions or computing data about context-free languages involving these two models. Unfortunately, the traditional conversion from a pushdown automaton to a context-free grammar has an exponential time and space complexity [5]. One way to improve this drawback is to define an equivalent pushdown automaton that avoids long strings of stack symbols. For instance, by having in the input a pushdown automaton with at most two symbols, the equivalent context-free grammar has a size of $O(n^3)$, where n is the length of the entire representation of the pushdown automaton [5].

The above models, pushdown automata and context-free grammars, have been also considered from the perspective of the bisimulation problem of system specification, that is, the problem of testing whether two or more processes have the same behavior. Caucal [2] showed that the bisimulation problem is decidable for the class of prefix transitions graphs of reduced context-free grammars. In addition, Caucal proved that for the deterministic case, the bisimulation problem is inter-reducible to the equivalence problem of deterministic pushdown automata. Recently, Jančar and Srba [6] proved the undecidability of weak bisimilarity for unrestricted popping pushdown processes, an open problem formulated by Senizergues [13] and Stirling [14].

This paper presents an efficient algorithm for doing the conversion from a pushdown automaton to a context-free grammar based on a fixed-point approach. Fixed-point approaches are also very useful for other related problems in formal languages (e.g., elimination of null and chain productions, finding the set of accessible states [5]), propositional and first-order logic (e.g., Robinson resolution [10]), deductive databases (e.g., Datalog [4]), graph theory (reflexive and transitive closure of a relation, stable matchings [3]), and so on.

There are many formulations of the fixed-point approach, but the main idea is the same. Briefly, an initial set of elements satisfying a property is given, and new elements satisfying another property are added (it could be the same property as for the initial set) until there are no more new elements to be added. To be more formal, let us consider a finite set of elements M. The set of all subsets of M is denoted as $P(M) = \{M' \mid M' \subseteq M\}$. Let us denote by $|X|$ the number of elements of set X. It is known that $|P(M)| = 2^{|M|}$. Since M is a finite set of elements, it follows that $P(M)$ is a finite set, too. Let us now consider an operator$T$: $P(M) \to P(M)$ such that $T(F) = F \cup \{x| \ x$ is a new element obtained as a combination of previous existing elements from $M\}$.

Let us consider an initial set of elements denoted by F_1. We apply the operator T to $F_1$and obtain $F_2 = T(F_1)$. This process is then repeated for any $k \geq 2$ such that $F_{k+1} = T(F_k)$. Given V_1 and V_2 two arbitrary subsets of M such that $V_1 \subseteq V_2$, then we get $T(V_1) \subseteq T(V_2)$. It follows that $F_1 \subseteq F_2 \subseteq F_3 \subseteq ... \subseteq P(M)$, so the operator T is monotonic. Since $P(M)$ is a finite set, it results that there exists a $k \in$ N (in this paper, N denotes the set of positive integers) such that $F_k = F_{k+1}$. This means $T(F_k) = F_k$. It is easy to check by mathematical induction that $T(F_{k+i}) = F_k$, $\forall i \geq 1$. That is why F_k is also called the *least fixed point* of T.

The contribution of this paper is two-fold:

• We define a new conversion technique from a pushdown automaton to a context- free grammar based on a fixed-point approach. Compared to the traditional approach, the obtained context-free grammar is small in the sense that all its variables are accessible.

• We define a new normal form of pushdown automata, called popping static pushdown automata (PSPDA), for which the conversion to a context-free grammar has a linear time complexity. The resulted context-free grammar has a linear size of the entire representation of the PSPDA. Moreover, we show that this new normal form has in fact the same expressive power as the traditional pushdown automata.

Section 2 describes the concepts of pushdown automata, context-free grammars, and their conversion. Moreover, a general fixed-point algorithm is defined. A correctness result and complexity issues are also presented. Section 3 defines a new normal form for pushdown automata, for which the previous algorithm leads to a linear size of the equivalent context-free grammar. Conclusions and References end this paper.

2. The Fixed-point Based Conversion Algorithm

We assume the reader is already familiar with pushdown automata and context-free grammars related concepts, but for the sake of the presentation, we shall give some useful definitions. A *pushdown automaton* (PDA), denoted by A, is a 7-tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where Q is the set of states, Σ is the finite set of input symbols, Γ is the stack alphabet, δ is the transition function, q_0 is the start state, z_0 is the start stack symbol, and F is the set of accepting states. If $(p, \alpha) \in \delta(q, a, x)$, then for all strings $w \in \Sigma*$ and $\beta \in \Gamma*$ we get the following *one-step transition* between pushdown configurations $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$. The language accepted by A by final state is $L(A) = \{w \mid (q_0, w, z_0) \overset{*}{\underset{A}{\vdash}} (q, \varepsilon, \alpha)\}$, where $q \in F$, ε is the empty word, α is any stack string, and $\overset{*}{\vdash}$ is the reflexive and transitive closure of the transition relation $(\vdash)$. The language accepted by A by empty stack is $N(\mathrm{A}) = \{w| \ (q_0, w, z_0) \ \overset{*}{\underset{A}{\vdash}} \ (q, \varepsilon, \varepsilon)\}$, for any state q. It is known that the class of languages $L(A_1)$ for some PDA A_1 is the same as the class of languages $N(A_2)$ for some PDA A_2.

A *context-free grammar* (CFG), denoted by G, is a 4-tuple $G = (V, \Sigma, S, P)$, where V is the set of variables, Σ the set of terminals, S the start symbol, and P the set of productions. Let $\alpha \ A \ \beta$ be a string with $\alpha, \ \beta \in (V \cup T)*$, $A \in V$, $\gamma \in (V \cup T)*$, and $A \to \gamma$ a production of G. Then $\alpha \ A \ \beta \ \underset{G}{\Rightarrow} \ \alpha \ \gamma \ \beta$ is called a *derivation step* of G. This derivation relation can be extended to its reflexive and transitive closure, denoted $\overset{*}{\Rightarrow}$. The language of G, denoted $L(G)$, is $L(G) = \{w \mid w \in T*, S \overset{*}{\underset{G}{\Rightarrow}} \ w\}$. A variable X is accessible if there exists a derivation from the start symbol to a string that contains X , that is, $S \overset{*}{\underset{G}{\Rightarrow}} \ \alpha \ X \ \beta$, where $\alpha, \ \beta \in (V \cup T)*$.

The traditional conversion from a PDA to a CFG is done in many formal languages textbooks, including [5,1].

Theorem 2.1. *(Theorem 6.14 from [5]) If A is a pushdown automata, then N(A) is a context-free language.*

Proof In Appendix.

Remark 2.1. *In the above proof, the equivalent context-free grammar has* $|\Sigma| + |Q|^2 . \ |\Gamma|$ *variables and* $|Q|^k$ *productions for a pushdown transition of type i). Also, we must add the productions obtained at point ii) and* $\{S \to (q_0, z_0, q) \ / \ q \in Q\}$*. Let us denote by n the size of the entire automaton representation. Since k can be quite close to n, it follows that the total number of productions of G can be* n^n*. There already exist efforts to reduce this exponential complexity. For instance, one way to reduce this high complexity is to split all the transitions containing long strings of stack symbols into sequences of at most n pushdown transitions that each pushes one symbol. This will lead to an equivalent CFG of* $O(n^3)$ *size (details in Section 7.4.1 of [5]).*

The next example illustrates the complexity of the equivalent CFG to a PDA.

Example 2.1. *For the context-free language $L = \{0^n1^{3n} / n \geq 1\}$, we consider the pushdown automaton $A = (\{q_0, q_1, q_2\}, \{0, 1\}, \{x, z\}, \delta, q_0, z, \emptyset)$, where δ is given by:*

1. $\delta(q_0, 0, z) = \{(q_1, x\ x\ x\ z)\}$;
2. $\delta(q_1, 0, x) = \{(q_1, x\ x\ x\ x)\}$;
3. $\delta(q_1, 1, x) = \{(q_2, \varepsilon)\}$;
4. $\delta(q_2, 1, x) = \{(q_2, \varepsilon)\}$;
5. $\delta(q_2, \varepsilon, z) = \{(q_0, \varepsilon)\}$.

According to the above proof, we obtain an equivalent grammar having 20 variables and 168 productions.

As anticipated in Remark 2.1, the equivalent CFG generated by the traditional algorithm can be very large. Example 2.1 shows the conversion from a PDA having 3 states, 2 input symbols, 2 stack symbols and 5 transitions to a quite large CFG having 20 variables and 168 productions. The issues in the equivalent CFG are that many of the variables are not accessible by derivations from the start symbol S or some of the variables do not generate the terminal words.

In order to solve these drawbacks, we describe below Algorithm **I** to illustrate our fixed-point approach. This provides as an output a much smaller CFG equivalent to the given PDA. The CFG provided as output has only accessible variables.

Algorithm I

```
The input: $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, \emptyset)$ a pushdown automaton;
The output: $G = (V, \Sigma, S, P)$ an equivalent context-free grammar, with only accessible variables;
The method:
1.  $V_1 := \emptyset$; $P_1 := \emptyset$;
2.  for (all $(q_1, \varepsilon) \in \delta(q, a, X)$, where $a \in \Sigma \cup \{e\}$) do begin
3.      $V_1 := V_1 \cup \{[q, X, q_1]\}$;
4.      $P_1 := P_1 \cup \{[q, X, q_1] \to a\}$
    end
5.  $k := 1$;
    repeat
6.      $V_{k+1} := V_k$; $P_{k+1} := P_k$;
7.      for (all $(q_1, z_1 z_2 \ldots z_m) \in \delta(q, a, X)$, where $a \in \Sigma \cup \{\varepsilon\}$) do begin
8.          $ok$ := true; $tempq := q_1$;
9.          for ($i := 1$; $i \leq m$ and $ok$; $i := i + 1$) do
10.             if ($\neg\exists q'$ such that $[tempq, z_i, q'] \in V_{k+1}$) then $ok$ := false
                else begin
11.                 $temp[i] := q'$; $tempq := q'$
                end
12.         if $ok$ then begin
13.             $V_{k+1} := V_{k+1} \cup \{[q, X, temp[m]]\}$;
14.             $P_{k+1} := P_{k+1} \cup \{[q, X, temp[m]] \to [q_1, z_1, temp[1]][temp[1], z_2, temp[2]] \ldots [temp[m-1], z_m, temp[m]]\}$
            end
        end ;
15.     $k := k + 1$
16. until $V_k = V_{k-1}$ and $P_k = P_{k-1}$;
17. if ($\exists\ [q_0, z_0, q] \in V_k$) then begin
18.     $V_k := V_k \cup \{S\}$;
19.     for (all $[q_0, z_0, q] \in V_k$) do
20.         $P_k := P_k \cup \{S \to [q_0, z_0, q]\}$
    end;
21. $V := V_k$; $P := P_k$;
```

We need some internal definitions, useful to combine the variables of the equivalent CFG and to prove Algorithm **I**'s correctness.

Definition 2.1. *We consider the operator,* $\circ$ *, useful for triplets' composition, thus:*

$\circ : (Q \times \Gamma^* \times Q)^2 \to Q \times \Gamma^* \times Q$

$$[q_1, u, q_2] \circ [q_3, v, q_4] = \begin{cases} [q_1, uv, q_4] & \text{if } q_2 = q_3 \\ \varnothing & \text{otherwise} \end{cases}$$

Using the left associativity, we can extend the operator $\circ$ to three or more triplets, that is, $(Q \times \Gamma^* \times Q)^m \to Q \times \Gamma^* \times Q$, $m \geq 3$, such as:

$[q_1, u_1, q_2] \circ [q_2', u_2, q_3] \circ \ \dots \ \circ [q_m', u_m, q_{m+1}] = ([q_1, u_1, q_2] \circ [q_2', u_2, q_3] \circ \dots \circ [q'_{m-1}, u_{m-1}, q_m]) \circ [q'_m, u_m, q_{m+1}]$.

Theorem 2.2. (*correctness and termination of Algorithm* **I**) *Let* $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, \varnothing)$ *be a pushdown automaton which is input for Algorithm* **I**. *After the execution of Algorithm* **I**, *then the following facts hold:*

(i) G is equivalent with A (*i.e.*, $L(G) = N(A)$);

(ii) The number of iterations of the Algorithm I is finite ($k \leq |Q|^2 \cdot |\Gamma|$).

Proof In Appendix.

Example 2.2. *We consider the PDA from Example 2.1. for finding the corresponding equivalent grammar, using Algorithm* **I** *instead of the traditional conversion algorithm.*

$V_1 = \{[q_1, x, q_2], [q_2, x, q_2], [q_2, z, q_0]\}$;

$P_1 = \{[q_1, x, q_2] \to 1, [q_2, x, q_2] \to 1, [q_2, z, q_0] \to \varepsilon\}$;

$V_2 = V1 \ \cup \ \{[q_0, z, q_0], [q_1, x, q_2]\}$;

$P_2 = P_1 \cup \{[q_0, z, q_0] \to 0[q_1, x, q_2][q_2, x, q_2][q_2, x, q_2][q_2, z, q_0],$
$\quad [q_1, x, q_2] \to 0[q_1, x, q_2][q_2, x, q_2][q_2, x, q_2][q_2, x, q_2]\}$;

$V_3 = V_2$;

$V_3 = V_3 \cup \{S\}$;

$P_3 = P_2 \cup \{S \to [q_0, z, q_0]\}$.

Therefore, following Algorithm I, we obtain an equivalent grammar with 6 rules and 5 variables (instead of 168 rules and 20 variables obtained using the usual algorithm).

As a remark regarding Example 2.2, the number of productions of the equivalent CFG is less than the equivalent context-free grammar provided by the traditional conversion algorithm for the same PDA. Section 3 presents a special type of pushdown automata for which Algorithm **I** provides a linear-size equivalent CFG in a linear time.

A different way to reduce the size of the equivalent CFG is to consider a restricted PDA in the input. Exercise 6.2.8 from [5] calls a PDA *restricted* if its transitions can increase the height of the pushdown stack by at most one symbol. That exercise asks to show a way to convert any PDA to a restrictive PDA. The idea of this conversion is to break the pushing of a long string of k stack symbols into a sequence of at most $k - 1$ transitions that each pushes one symbol. These $k - 1$ new transitions will introduce $k - 2$ new states, so the size of the equivalent restricted PDA is definitely larger than the original PDA. Given as input of their conversion algorithm a restricted PDA of length n, the equivalent CFG will has a $O(n^3)$ size. The reason for that is the n PDA transitions will generate $O(n^3)$ productions is because there are only two states that need to be chosen in the productions that come from each transition (Theorem 7.31 from [5]).

Next section defines a different approach than the one from [5] for obtaining even smaller CFGs equivalent to another special kind of PDA.

3. Popping Static Pushdown Automata

This section defines a special form of pushdown automata that lead to linear size for the equivalent CFGs. It refers to the popping transitions of a pushdown automata. These are transitions that remove the top of the pushdown stack. More formally, a transition $(q', \varepsilon) \in \delta(q, a, x)$ is called a *popping transition*. We say that a popping transition is *static* if and only if for all $q \in Q$ and $x \in \Gamma$, there exists at most one state $q' \in Q$ such that $(q', \varepsilon) \in \delta(q, a, x)$, $\forall a \in \Sigma$. We say that a pushdown automaton is *popping static* if and only if its popping transitions are static (we denote by PSPDA a popping static PDA). In other words, for all $a, b \in \Sigma$ the transitions $(q', \varepsilon) \in \delta(q, a, x)$ and $(q'', \varepsilon) \in \delta(q, b, x)$ lead to a popping static pushdown automaton if and only if $q' = q''$.

Like many other models in formal languages, pushdown automata have two views: a syntactical view and a semantical view. Since we introduce a new kind of nontrivial class of automata, it makes sense to compare this class with the traditional classes of automata. Given two automata $A_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, q_{0,1}, Z_{0,1}, F_1)$ and $A_2 = (Q_2, \Sigma_2, \Gamma_2, \delta_2, q_{0,2}, Z_{0,2}, F_2)$, we say that A_1 is syntactically equal with A_2 (denoted as $A_1 = A_2$) if and only if $Q_1 = Q_2$, $\Sigma_1 = \Sigma_2$, $\Gamma_1 = \Gamma_2$, $\delta_1 = \delta_2$, $q_{0,1} = q_{0,2}$, $Z_{0,1} = Z_{0,2}$, and $F_1 = F_2$. In other words, two automata are syntactically equal if and only if they coincide element by element.

The semantical view refers to the automata's acceptance or expressive power. We say that A_1 and A_2 are semantically equal (denoted as $A_1 \equiv A_2$, also known as equivalent), if and only if $L(A_1) = L(A_2)$. Obviously, the syntactical equality is 'stronger' than the semantical equality (because two syntactically equal automata are semantically equal, too).

One can ask about the relationship between PSPDA and the traditional deterministic PDA. A PDA $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is *deterministic* (denoted by DPDA) if and only if the following conditions hold:

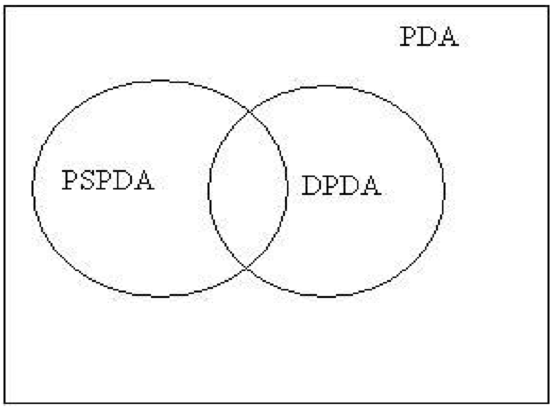

Figure 1. The syntactical view for PSPDA, DPDA, and PDA

• there exists at most one transition $\delta(q, a, x)$, for all $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $x \in \Gamma$;

• If $\delta(q, a, x) \neq \emptyset$ for some $a \in \Sigma$, then $\delta(q, \varepsilon, x) = \emptyset$.

Both PSPDA and DPDA contain determinism in their transitions. But the traditional DPDA refers to determinism for all transitions for a given $\delta(q, a, x)$, whereas the PSPDA is a relaxed version of determinism for $\delta(q, a, x)$ that covers only the popping transitions.

For instance, the PDA defined in Example 2.1 is both a DPDA and PSPDA automaton. However, if we simply do some minor changes, it changes its membership. Let us denote by A_1 the automaton A that has one more transition, that is, $\delta(q_2, 0, x) = \{(q_1, \varepsilon)\}$. Then automaton A_1 is still a DPDA, but not a PSPDA because the pair (q_2, x) corresponds to two states, these are, q_1 and q_2.

On the other hand, let us denote by A_2 the automaton A that has one more different transition, that is, $\delta(q_1, 0, x) = \{(q_2, \varepsilon)\}$. Then automaton A_2 is still a PSPDA, but not a DPDA because $\delta(q_1, 0, x)$ is not unique.

The above three automata examples motivate *figure 1*. Automaton A from Example 2.1 is both PSPDA and DPDA. Automaton A_1 is from PSPDA-DPDA, whereas A_2 is from DPDA-PSPDA.

The above comparison between DPDA and PSPDA was done syntactically, and not semantically. The next result shows that the expressive power of PSPDA equals to the whole class of, context-free languages. It is known that DPDA represents a proper inclusion in the class of context-free languages. Hence, PSPDA have strictly more expressive power than DPDA.

Theorem 3.1. *Let G = (V, T, P, S) be an arbitrary CFG with a representation of length n. Then there exists a PSPDA A of size O(n) such that N(A) = L(G), that can be generated using a linear time complexity algorithm.*

Proof In Appendix.

It is known that an arbitrary PDA can be converted to a CFG. Theorem 3.1. shows that an arbitrary CFG can be (linearly) converted to a PSPDA. Thus, an arbitrary PDA can be converted to a PSPDA. Since any PSPDA is a PDA, it follows that the class of PSPDA equals to the class of PDA. Moreover, the class of DPDA is a proper inclusion of PSPDA. *Figure 2* illustrates these relationships.

The next result proves the complexity of Algorithm **I**.

Corollary 3.1. *If the input of Algorithm* **I** *is a PSPDA automaton, then the equivalent CFG provided as output has a linear size of the entire representation of PSPDA. Algorithm* **I** *runs in linear time complexity.*

Proof In Appendix.

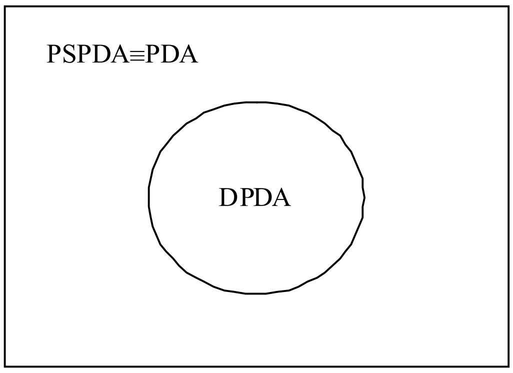

Figure 2. The semantical view for PSPDA, DPDA, and PDA

4. Conclusions

In this paper, we introduced a new algorithm for conversion between two important models from formal languages, pushdown automata and context-free grammars. The algorithm is efficient in the sense that it provides a smaller CFG than the traditional conversion result. Our technique is based on fixed-point approach and can successfully be used for converting PDAs and PSPDAs into CFGs. PSPDA can be considered a new canonical form for defining pushdown automata. If the input automaton is PSPDA, then our conversion algorithm will generate in linear time a linear-size equivalent CFG. Even if PSPDA have clear syntactical restrictions to the classical PDA, the expressive power of PSPDA is the same as the traditional PDA.

As future work, we shall investigate a direct translation (that avoids CFGs) from an arbitrary PDA to a PSPDA. Moreover, it is worth to investigate the fixed-based approach for converting other models, e.g., from arbitrary CFGs to special normal form CFGs, and so on. Authors intend to investigate specification of larger industrial systems using our subclasses of popping static pushdown automata.

References

1. Autebert, Jean-Michel, Jean Berstel, and Luc Boasson. Context-free Languages and Pushdown Automata. Handbook of Formal Languages, vol. 1: Word, Language, Grammar, 1997, 111-174.

2. Caucal, Didier. Bisimulation of Context-free Grammars and of Pushdown Automata. Modal Logic and Process Algebra, 1995, 53:85-106.

3. Fleiner, Tamas. A Fixed-point Approach to Stable Matchings and Some Applications. – *Math. Oper. Res.*, 28 (1), 2003, 103-126.

4. Hafner, Carole D. and Kurt Godden. Portability of Syntax and Semantics in Aatalog. – *ACM Trans. Inf. Syst.*, 3 (2), 1985, 141-164.

5. Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

6. Jančar, Petr and Jivři Srba. Undecidability of Bisimilarity by Defender's Forcing. – *Journal of the ACM*, 55 (1), 2008, 1-26.

7. Mateescu, Alexandru and Arto Salomaa. Aspects of Classical Language Theory. Handbook of Formal Languages, vol. 1: Word, Language, Grammar, 1997, 175-251.

8. Mateescu, Alexandru and Arto Salomaa. Formal Languages: an Introduction and a Synopsis. Handbook of Formal Languages, 1: Word, Language, Grammar, 1997, 1-39.

9. Paulson, Linda Dailey. IBM System is a Virtual Sign-language Interpreter. – *Computer*, 41 (2), 2008, 23-23.

10. Robinson, J. A. A Machine-oriented Logic based on the Resolution Principle. – *J. ACM*, 12 (1), 1965, 23-41.

11. Rozenberg, Grzegorz and Arto Salomaa. Editors. Handbook of Formal Languages, vol. 1: Word, Language, Grammar. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

12. Salomaa, Arto. Theory of Automata. Pergamon Press, 1969.

13. Sénizergues, Géraud. L(A) = L(B)? Decidability Results from Complete Formal Systems. – *Theor. Comput. Sci.*, 251 (1-2), 2001, 1-166.

14. Stirling, Colin. Decidability of Bisimulation Equivalence for Normed Pushdown Processes. – *Theor. Comput. Sci.*, 195 (2), 1998, 113-131.

15. Yu, Sheng. Regular Languages. Handbook of Formal Languages, vol. 1: Word, Language, Grammar, 1997, 41-110.

Manuscript received on 7.05.2009

Stefan Andrei *received the BSc and MSc degrees in computer science from Cuza University of Iasi, Romania in 1994 and 1995, respectively, and the PhD degree in computer science from Hamburg University, Germany in 2000. He is currently an Associate Professor with the Department of Computer Science, Lamar University, Texas, U.S.A. His research interests are in real-time embedded systems and software engineering. He has served as a program committee member in more than 30 international reputable conferences and has published more than 70 peer-reviewed refereed scientific papers. More details about Stefan are at his webpage: http://galaxy.lamar.edu/~sandrei/*

Contacts:
Lamar University, Department of Computer Science
211 Red Bird Ln, Texas, 77710 USA,
tel: 409-880-8748
fax: 409-880-2364
email: sandrei@my.lamar.edu

Hikyoo Koh *receives the B.A. in Law from the Yung-Nam University, Taegu, Korea, in 1963, and M.S. in Computer and Information Sciences from University of Hawaii, Honolulu, Hawaii, USA in 1971. He received Ph.D. in Computer Science, University of Pittsburgh, Pennsylvania, USA in 1978. He is currently a Professor of Computer Science with Lamar University, Beaumont, Texas, USA. His research areas include Computational Complexity Analysis, Algorithm Design, Language Design, and Ethics Education. In addition, he is a Life Member of IEEE.*

Contacts:
Lamar University, Department of Computer Science
211 Red Bird Ln, Texas, 77710, USA
tel: 409-880-8779
fax: 409-880-2364
email: hkoh@my.lamar.edu

APPENDIX

Proof of Theorem 2.1. Let $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$. We define $G = (V, \Sigma, S, P)$, where $V = \Sigma \cup (Q \times \Gamma \times Q)$, $S = \{(q_0, z_0, q) \mid q \in Q\}$, and P is given by:

i) For all $k \geq 1$, each $a \in \Sigma \cup \{\varepsilon\}$, each $z, z_1, \ldots, z_k \in \Gamma$, each $q, p, q_1, \ldots, q_k \in Q$, $(q, z_1, q_k) \rightarrow a(p, z_1, q_1)(q_1, z_2, q_2) \ldots (q_{k-2}, z_{k-1}, q_{k-1})(q_{k-1}, z_k, q_k)$ if $(p, z_1 \ldots z_k) \in \delta(q, a, z)$;

ii) $(q, z, p) \rightarrow a$ if $(p, \varepsilon) \in \delta(q, a, z)$.

Proof of Theorem 2.2. The proof is based on showing that the pair $F = (V, P)$ is actually a fixed-point value obtained as an output of the previous algorithm. From Algorithm **I**, (step 8), it is easy to remark that $V_1 \subseteq V_2 \subseteq \ldots \subseteq V_j \subseteq V_{j+1} \subseteq \ldots \subseteq Q \times \Gamma \times Q$. But $Q \times \Gamma \times Q$ is a finite set, and the inclusion array from above is infinite. Therefore, $\exists k \geq 1$ such that $V_k = V_{k+1}$ (we denote by k the minimal integer which satisfies this property).

By induction on j, we prove that $V_k = V_{k+j}$, $\forall j \geq 1$. This means that V_k is the maximal set which contain accessible variables. Let us recall that a variable X is accessible if there exists a derivation from the start symbol to a string that contains X.

The Induction Basis: For $j = 1$ it is obvious that the equality $V_k = V_{k+j}$ holds.

The Induction Step: We suppose $V_k = V_{k+l}$, $\forall l \leq j$, and we must prove that $V_k = V_{k+l+1}$. But $V_{k+l+1} = V_{k+l} \cup SV$, where SV is the set of variables $[q, X, temp[m]]$ which is added at step 13 of Algorithm **I**. So, at step 14 of Algorithm **I**, the production $[q, A, temp[\mathrm{m}]] \rightarrow [q_1, z_1, temp[1]]\,[temp[1], z_2, temp[2]] \ldots [temp[m-1], z_m, temp[m]]$ is added to P_{k+1}, where $[q_1, z_1, temp[1]], [temp[1], z_2, temp[2]], \ldots, [temp[m-1], z_m, temp[m]] \in V_{k+l}$. But $V_{k+l} = V_k$, (from the inductive hypothesis), so $[q_1, z_1, temp[1]], [temp[1], z_2, temp[2]], \ldots, [temp[m-1], z_m, temp[m]] \in V_k$. Therefore $V_{k+l+1} = V_{k+1}$. Also, from the induction step for $l = 1$, it results $V_{k+1} = V_k$. Hence $V_{k+l+1} = V_k$. Thus, from the induction basis and the induction step, it results that $V_k = V_{k+j}$, $\forall j \geq 1$.

Therefore, $V_k \subseteq Q \times \Gamma \times Q$. This implies $|V_k| \leq |Q|^2 \cdot |\Gamma|$. But the number of iterations of Algorithm **I** is less or equal than $|V_k|$. Hence, part (ii) of this theorem is proved.

To prove part (i), we show that Algorithm **I** provides by V_k the set of accessible variables and by P_k the corresponding productions. Then, according to Theorem 6.14 from [5], it results $L(G') = N(A)$, where G' is the CFG from Theorem 2.1. Grammar G is actually similar to grammar G', but it contains only the productions of G' that has accessible symbols. The productions of G' that contain inaccessible symbols (that is, symbols that cannot be reached by any derivation from the starting symbol S) are in fact useless, hence they are not to be found in the grammar G. In conclusion, $L(G) = L(G')$. Thus, it follows that $L(G) = N(A)$.

We still have to prove that Algorithm **I** provides as an output the correct grammar G. Step 1 initializes V_1 to Ø. At steps 2, 3, and 4 we construct the accessible variables using an one-step derivation:

$[q, A, q_1] \rightarrow a$ if and only if $[q, A, q_1] \underset{G}{\overset{+}{\Rightarrow}} a$.

So, we get that V_1 is the set of accessible variables using an one-step derivation.

$(V_{k+1} = V_k \cup \{x \mid x \rightarrow x_1 x_2 \ldots x_k,\ x_1, x_2, \ldots, x_k \in V_k\})$.

Suppose by induction that we construct V_k the set of accessible variables and P_k the corresponding rules. From step 6, we have $V_{k+1} = V_k$. Step 9 corresponds to a PDA transition from state q, input symbol a and top of stack X. The set V_k has m variables such that:

$[q_1, z_1, temp[1]] \circ [temp'[1], z_2, temp[2]] \circ \ldots \circ [temp'[m\text{-}1], z_m, temp[m]] = [q_1'\ z_1 z_2 \ldots z_m, q]$.

According to Definition 1.1, we obtain:

$temp[2] = temp'[2]$, $temp[3] = temp'[3]$, ..., $temp[m\text{-}1] = temp'[m\text{-}1]$, $q_1 = q_1'$, and $q = temp[m]$.

If $\exists\, k \in \{2, 3, \ldots, m\}$ such that $temp[k] \neq temp'[k]$, it results that we have to find another variable for the correct composite. This process is achieved by steps 8 to 16 of Algorithm **I**. The Boolean variable ok from steps 8 to 11 is evaluated to $true$ if and only if all the variables $[q_1, z_1, temp[1]], [temp[1], z_2, temp[2]], \ldots, [temp[m-1], z_m, temp[m]]$ lead to terminal words. Steps 17 to 21 introduce the start symbol in the set of variables and add the corresponding productions to P_k. Hence, all the generated variables are accessible. Thus, the proof of Theorem 2.2 is completed.

Proof of Theorem 3.1. The proof follows first the traditional method of converting a CFG into an equivalent PDA (Theorem 6.13 from [5]). Given $G = (V, T, P, S)$ an arbitrary CFG, we consider A a PDA defined as $A = (\{q\}, T, V \cup T, \delta, q, S)$, where the transition function δ is defined by:

- for each $x \in V$, $\delta(q, \varepsilon, x) = \{(q, x) \mid x \rightarrow \alpha \in P\}$;
- for each $a \in T$, $\delta(q, a, a) = \{(q, \varepsilon)\}$.

The first part of the proof is identical with the traditional result. In fact, the number of transitions of the equivalent

PDA equals to $|V| \cup |T|$, so A has size $O(n)$ and can be generated using a linear time complexity algorithm.

We need now to prove that A is a PSPDA. In order to show that there exists at most one popping transition for all $q \in Q$ and $y \in V \cup T$, we distinguish two cases:

- $y \in T$. Then there is exactly one popping transition, namely $\delta(q, y, y) = \{(q, \varepsilon)\}$;
- $y \in V$. If $\alpha \neq \varepsilon$ then there is no popping transition for q and y. Otherwise, if $\alpha = \varepsilon$ then there is exactly one state $q \in Q$ such that $(q, \varepsilon) \in \delta(q, \varepsilon, y)$.

In conclusion, A is a PSPDA automaton, hence Theorem 3.1 is completely proved.

Proof of Corollary 3.1. The proof is similar to proof of Theorem 2.2. Since the input automaton A is PSPDA, then according to Algorithm **I**, for any pushdown transition of A, there exists only one context-free production and a variable corresponding to it. In fact, the number of variables of the equivalent CFG equals to the number of transitions of A. Moreover, the number of productions equals to the number of transitions of A plus one more production for the start symbol. In the end, Algorithm **I** adds a constant number of productions corresponding to the initial state and stack symbol. We conclude that the size of the equivalent CFG is $O(n)$ and Algorithm **I** runs in a linear time complexity, where n is the length of the entire representation of A.