On Model Checking for Visibly Pushdown Automata

Tang Van Nguyen and Hitoshi Ohsaki

Research Team for Verification and Specification
National Institute of Advanced Industrial Science and Technology, Japan
{t.nguyen,ohsaki}@ni.aist.go.jp

Abstract. In this paper we improve our previous work by introducing *optimized on-the-fly* algorithms to test universality and inclusion problems of visibly pushdown automata. We implement the proposed algorithms in a prototype tool. We conduct experiments on randomly generated VPA. The experimental results show that the proposed method outperforms the standard one by several orders of magnitude.

1 Introduction

Visibly pushdown automata [1] are pushdown automata whose stack behavior (*i.e.,* whether to execute push, pop, or no stack operation) is completely determined by the input symbol according to a fixed partition of the input alphabet. As shown in [1], this class of visibly pushdown automata enjoys many good properties similar to those of the class of finite automata. The main reason for this being is that, each nondeterministic VPA can be transformed into an equivalent deterministic one. Therefore, checking context-free properties of pushdown models is decidable as long as the calls and returns are made visible. As a result, visibly pushdown automata have turned out to be useful in some context, *e.g.,* as automaton model for processing XML streams [6,3], and as AOP protocols for component-based systems [4]. To check universality for a nondeterministic VPA M over its alphabet Σ (that is, to check if $L(M) = \Sigma^*$), the standard method is first to make it complete, determinize it, complement it, and then check it for emptiness. To check the inclusion problem $L(M) \subseteq L(N)$, the standard method computes the complement of N, takes its intersection with M and then, checks for emptiness. This is costly as computing the complement necessitates a full determinization. This explosion is in some senses inevitable, because determinization for VPA requires exponential time blowup [1]. This raises a natural question: Are there methods to efficiently implement decision procedures like universality (or inclusion) checking for VPA.

A *pushdown system* (*e.g.,* see [2]) is a pushdown automaton that is regardless of input symbols. Bouajjani *et al.* [2] have introduced a method to compute reachable configurations of a pushdown system. The key of their technique is to use a finite automaton so-called $\mathcal{P}$-*automaton* to encode a set of infinite configurations of a pushdown system. In our previous paper [5], we proposed on-the-fly algorithms to check universality and inclusion of VPA. The key idea is based on doing determinization and generating $\mathcal{P}$-automata simultaneously.

In this paper we improve the algorithms presented in [5] by optimizing the determinized VPA and generated $\mathcal{P}$-automaton as follows. First, we propose an on-the-fly method to test universality of VPA M. In particular, in order to check universality of a nondeterministic VPA, we simultaneously determinize this VPA and apply the $\mathcal{P}$-automata technique to compute a set of reachable configurations of the target determinized VPA. When a rejecting configuration is found, the checking process stops and reports that the original VPA is not universal. Otherwise, if all configurations are accepting, the original VPA is universal. Furthermore, to strengthen the algorithm, we define a partial ordering over transitions of $\mathcal{P}$-automaton, and only *minimal* transitions are used to incrementally generate the $\mathcal{P}$-automaton. The purpose of this process is to keep the determinization step implicitly for generating reachable configurations as small as possible. This improvement helps to reduce not only the size of the $\mathcal{P}$-automaton but also the complexity of the determinization phase. The intuitive idea behind this process is to find whether there exists a word w such that $w \notin L(M)$ as early as possible. Second, an algorithmic solution to inclusion checking for VPA using on-the-fly manner will be presented. Again, no explicit determinization is performed. To solve the language-inclusion problem for nondeterministic VPA, $L(M) \subseteq L(N)$, the main idea is to find at least one word w accepted by M but not accepted by N, *i.e.,* $w \in L(M) \setminus L(N)$. Finally, we implement all algorithms in a prototype tool, named VPAChecker, and tested them in a series of experiments. Our preliminary experiments on randomly generated VPA show a significant improvement of on-the-fly methods compared to the standard ones.

The remainder of this paper is organized as follows. In Section 2 we recall basic notions and properties of VPA, and then we give an improvement on determinization of VPA. Section 3 propose optimized on-the-fly algorithms for checking universality and inclusion of VPA. Also, the correctness proof of the proposed algorithm is presented in this section. Implementation as well as experimental results are presented and analyzed in Section 4. Finally, we conclude the paper in Section 5.

2 Visibly Pushdown Automata

2.1 Definitions

Let Σ be the finite input alphabet, and let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ be a partition of Σ. The intuition behind the partition is: Σ_c is the finite set of *call* (push) symbols, Σ_r is the set of *return* (pop) symbols, and Σ_i is the finite set of *internal* symbols. Visibly pushdown automata are formally defined as follows:

Definition 1 (Visibly Pushdown Automata [1]). *A* visibly pushdown automaton *(VPA) M over Σ is a tuple $M = (Q, \Gamma, Q_0, \Delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Γ is a* *finite stack alphabet with a special symbol $\perp$ (representing the* bottom-of-stack*), and $\Delta = \Delta_c \cup \Delta_r \cup \Delta_i$ is the transition relation, where $\Delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\perp\})$, $\Delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$, and $\Delta_i \subseteq Q \times \Sigma_i \times Q$.*

- If $(q, c, q', \gamma) \in \Delta_c$, where $c \in \Sigma_c$ and $\gamma \neq \perp$, there is a *push-transition* from q on input c where when reading c, γ is pushed onto the stack and the control changes from state q to q'; we denote such a transition by $q \xrightarrow{c/+\gamma} q'$.
- Similarly, if $(q, r, \gamma, q') \in \Delta_r$, there is a *pop-transition* from q on input r where γ is read from the top of the stack and popped (if the top of the stack is $\perp$, then it is read but not popped), and the control state changes from q to q'; we denote such a transition $q \xrightarrow{r/-\gamma} q'$.
- If $(q, i, q') \in \Delta_i$, there is an *internal-transition* from q on input i where when reading i, the state changes from q to q'; we denote such a transition by $q \xrightarrow{i} q'$. Note that there are no stack operations on internal transitions.

Let $St = \{w\perp \mid w \in (\Gamma \setminus \{\perp\})^*\}$ be the set of *stack contents*. A *configuration* is a pair (q, σ) of $q \in Q$ and $\sigma \in St$. The transition relation of a VPA can be used to define how the configuration of the machine changes in a single step: we say $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if one of the following conditions holds:

- If $a \in \Sigma_c$ then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/+\gamma} q'$ and $\sigma' = \gamma \cdot \sigma$
- If $a \in \Sigma_r$, then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/-\gamma} q'$ and either $\sigma = \gamma \cdot \sigma'$, or $\gamma = \perp$ and $\sigma = \sigma' = \perp$
- If $a \in \Sigma_i$, then $q \xrightarrow{a} q'$ and $\sigma = \sigma'$.

A (q_0, w_0)-*run* on a word $u = a_1 \cdots a_n$ is a sequence of configurations $(q_0, w_0) \xrightarrow{a_1} (q_1, w_1) \cdots \xrightarrow{a_n} (q_n, w_n)$, and is denoted by $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$. A word u is accepted by M if there is a run $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$ with $q_0 \in Q_0$, $w_0 = \perp$, and $q_n \in F$. The language $L(M)$ is the set of words accepted by M. The language $L \subseteq \Sigma^*$ is a *visibly pushdown language* (VPL) if there exists a VPA M with $L = L(M)$.

Definition 2 (Deterministic VPA [1]). *A VPA M is* deterministic *if $|Q_0| = 1$ and for every configuration (q, σ) and $a \in \Sigma$, there are at most one transition from (q, σ) by a. For deterministic VPA (DVPAs) we denote the transition relation by δ instead of Δ, and write: $\delta(q, a) = (q', \gamma)$ instead of $(q, a, q', \gamma) \in \Delta_c$ if $a \in \Sigma_c$, $\delta(q, a, \gamma) = q'$ instead of $(q, a, \gamma, q') \in \Delta_r$ if $a \in \Sigma_r$, and $\delta(q, a) = q'$ instead of $(q, a, q') \in \Delta_i$ if $a \in \Sigma_i$.*

As shown in [1], any nondeterministic VPA can be transformed into an equivalent deterministic one. The construction has two components: a set of *summary edges* S, that keeps track of what state transitions are possible from a push transition to the corresponding pop transition, and a set of *path edges* R, that keeps track of all possible states reached from initial states. Reader(s) are referred to [1] for more details. During implementation of VPA's operations, we found that the set of summaries S may contain redundant pairs in the sense that these pairs

Algorithm 1. Optimized determinization for VPA

```
    Data: A nondeterministic VPA $M = (Q, \Gamma, Q_0, \Delta, F)$
    Result: A determinized VPA $M^{od} = (Q', \Gamma', Q'_0, \Delta', F')$
1   begin
2       $Q' = 2^{Q \times Q}$, $\Gamma' = Q' \times \Sigma_c$,
3       $Q'_0 = \{Id_{Q_0}\}$, $F' = \{S \in Q' \mid \Pi_2(S) \cap F \neq \varnothing\}$,
4       and the transition relation $\Delta' = \Delta'_i \cup \Delta'_c \cup \Delta'_r$ is given by:
          – Internal: For every $a \in \Sigma_i$, $S \xrightarrow{a} S' \in \Delta'_i$ where
            $S' = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S, q'' \xrightarrow{a} q' \in \Delta_i\}$.
          – Push: For every $a \in \Sigma_c$, $S \xrightarrow{a/+(S,a)} Id_{R'} \in \Delta'_c$ where
            $R' = \{q' \mid \exists q \in \Pi_2(S) : q \xrightarrow{a/+\gamma} q' \in \Delta_c\}$.
          – Pop: For every $a \in \Sigma_r$,
            - if the stack is empty : $S \xrightarrow{a/-\bot} S' \in \Delta'_r$ where
            $S' = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S, q'' \xrightarrow{a/-\bot} q' \in \Delta_r\}$.
            - otherwise:
            $S \xrightarrow{a/-(S',a')} S'' \in \Delta'_r$, where
              $\begin{cases} S'' = \{(q,q') \mid \exists q_3 \in Q : (q,q_3) \in S', (q_3,q') \in Update\} \\ Update = \left\{ (q,q') \,\middle|\, \begin{array}{l} \exists q_1, q_2 \in Q : (q_1,q_2) \in S, \\ q \xrightarrow{a'/+\gamma} q_1 \in \Delta_c, q_2 \xrightarrow{a/-\gamma} q' \in \Delta_r \end{array} \right\} \end{cases}$
```

do not keep information of reachable states. In other words, Alur-Madhusudan's algorithm defines each state of the output deterministic VPA as a pair (S, R). However, by a little modification of the algorithm, we can make every pair (S, R) satisfy $\Pi_2(S) = R$ without disturbing the correctness of the algorithm (where, $\Pi_2(S) = \{s \mid (s, s') \in S\}$ is the projection on the second component.) After this modification, the component R is no longer needed. In the following, we formally present an optimization for determinization by keeping the set of summaries as small as possible. For a finite set X, let denote $Id_X = \{\ (q, q) \mid q \in X\}$. Let $M = (Q, \Gamma, Q_0, \Delta, F)$ be a nondeterministic VPA. We construct an equivalent deterministic VPA $M^{od} = (Q', \Gamma', Q'_0, \Delta', F')$ (*od* stands for *optimized determinization*) as presented in Algorithm 1.

Theorem 3 (Optimized Determinization [5]). *For a given nondeterministic VPA M of n states, one can construct a deterministic VPA M^{od} such that $L(M^{od}) = L(M)$. Moreover, the number of states and stack symbols of M^{od} in the worst case are 2^{n^2} and $|\Sigma_c| \cdot 2^{n^2}$, respectively.*

3 Universality and Inclusion Checking

3.1 Emptiness Checking

A *pushdown system* (*e.g.,* see [2]) is a pushdown automaton that is regardless of input symbols. Bouajjani *et al.* [2] have introduced a method to compute reachable configurations of a pushdown system. The key of their technique is to use a finite automaton "so-called $\mathcal{P}$*-automaton*" to encode a set of infinite configurations of a pushdown system. In the following, we apply $\mathcal{P}$-automata technique to check emptiness of visibly pushdown automata. Our definition, though in essence does not differ from the one in [2], has been tailored so that concepts discussed in this paper are easily related to the definition. Given a VPA $\mathcal{P} = (Q, \Gamma, Q_0, \Delta, F)$, a $\mathcal{P}$-automaton is used in order to represent sets of configurations C of $\mathcal{P}$. A $\mathcal{P}$-automaton uses Γ as the input alphabet, and Q as set of initial states. Formally,

Definition 4 ($\mathcal{P}$-automata [2]).

1. *A $\mathcal{P}$-automaton of a VPA $\mathcal{P}$ is a finite automaton $A = (P, \Gamma, \delta, Q, F_A)$ where P is the finite set of states, $\delta \subseteq P \times \Gamma \times P$ is the set of transitions, $Q \subseteq P$ is the set of* initial states *and $F_A \subseteq P$ is the set of* final states*.*
2. *A $\mathcal{P}$-automaton* accepts *or* recognizes *a configuration (p, w) if $p \xrightarrow{w} q$, for some $p \in Q$, $q \in F_A$. The set of configurations recognized by $\mathcal{P}$-automaton A is denoted by $Conf(\mathcal{P})$.*

For a VPA $\mathcal{P} = (Q, \Gamma, Q_0, \Delta, F)$ and a set of configurations C, let A be a $\mathcal{P}$-automaton representing C. The $\mathcal{P}$-automaton $A_{Post^*(C)}$ representing the set of configurations reachable from C (denoted by $Post^*(C)$) is constructed as follows: We compute $Post^*(C)$ as a language accepted by a $\mathcal{P}$-automaton $A_{post^*(C)}$ with ϵ-moves. We denote the relation $q(\xrightarrow{\epsilon})^* \cdot \xrightarrow{\gamma} \cdot (\xrightarrow{\epsilon})^* p$ by $q \Longrightarrow^{\gamma} p$. Formally, $A_{post^*(C)}$ is obtained from A by the procedure given in Algorithm 2:

3.2 Universality Checking

Standard Method. The standard algorithm for universality of VPA is to first determinize the automaton, and then check for the reachability of a non-accepting state. Reachable configurations of a determinized VPA can be computed by using $\mathcal{P}$-automata technique. A configuration $c = (q, w)$ is *rejecting* if q is not a final state. When a rejecting configuration is found, we stop and report that the original VPA is not universal. Otherwise, if all reachable configurations of determinized VPA are accepting, the original VPA is universal. Let $\mathsf{ReachableConf}(M^{od})$ and $\mathsf{RejectingConf}(M^{od})$ denote the sets of reachable and rejecting configurations of M^{od}, respectively. With the above observation, we obtain the following lemma:

Lemma 5. *Let M be a nondeterministic VPA according to Algorithm 1. The automaton M is not universal iff there exists a rejecting reachable configuration of M^{od},* i.e., $\mathsf{ReachableConf}(M^{od}) \cap \mathsf{RejectingConf}(M^{od}) \neq \varnothing$.

Algorithm 2. The algorithm to construct $\mathcal{P}$-Automaton

```
Data: A VPA $\mathcal{P} = (Q, \Gamma, Q_0, \Delta, F)$ and a finite automaton $A$ representing set
      of configurations $C$.
Result: A finite automaton $A_{post^*(C)}$ representing $post^*(C)$
1 begin
2     for (each pair $(q', \gamma')$ such that $\mathcal{P}$ contains at least one rule of the form
      $q \xrightarrow{a/+\gamma'} q' \in \Delta_c$) do
3         Add a new state $p_{(q',\gamma')}$ to $A$. Here $(q', \gamma')$ is used as an index to
          distinguish $p_{(q',\gamma')}$ with others newly added states, and thus, we can
          minimize the number of necessary added states.
4     Add new transitions to $A$ according to the following saturation rules:
        1. Internal: if ($q \xrightarrow{a} q' \in \Delta_i$ and $q \Longrightarrow^{\gamma} p$ in the current automaton) then
               Add a transition $(q', \gamma, p)$.
        2. Push: if ($q \xrightarrow{a/+\gamma'} q' \in \Delta_c$ and $q \Longrightarrow^{\gamma} p$ in the current automaton) then
               first add $(q', \gamma', p_{(q',\gamma')})$, and then add $(p_{(q',\gamma')}, \gamma, p)$.
        3. Pop: if ($q \xrightarrow{a/-\gamma} q' \in \Delta_r$ and $q \Longrightarrow^{\gamma} p$ in the current automaton) then
               Add a transition $(q', \epsilon, p)$.
```

Optimized On-the-fly Method. To improve efficiency of the checking process, we perform simultaneously on-the-fly determinization and $\mathcal{P}$-automata construction. There are two interleaving phases in this approach. First, we determinize VPA M step by step (iterations). After each step of determinization, we update the $\mathcal{P}$-automaton. Second, using the $\mathcal{P}$-automaton, we perform determinization again, and so on. It is crucial to note that this procedure terminates. This is because the size of the M^{od} is finite, and the $\mathcal{P}$-automaton construction terminates. Lemma 5 means that checking universality of M amounts to finding a rejecting configuration of M^{od}. In the following we present an on-the-fly way to explore such rejecting configurations (if there are any) efficiently. We begin with the following observations that play an important role in establishing theoretical background for correctness of our algorithms. For a given nondeterministic VPA M, let M^{od} be the determinized VPA. Recall that a state S of M^{od} belongs to $2^{Q \times Q}$. We now define an ordering over states and stack symbols of M^{od} as follows:

Definition 6 (Partial ordering over states and stack symbols).

- *Let S_1 and S_2 be states of M^{od}. We say $S_1 \leq S_2$ if, $S_1 \subseteq S_2$.*
- *Let $\gamma'_1 = (S_1, a)$ and $\gamma'_2 = (S_2, a)$ be stack symbols of M^{od}. We say $\gamma'_1 \leq \gamma'_2$ if $S_1 \subseteq S_2$.*

Lemma 7. *Let $S_1 \xrightarrow{a} S'_1$ and $S_2 \xrightarrow{a} S'_2$ be internal transitions of the determinized VPA M^{od}. We have $S'_1 \leq S'_2$ if $S_1 \leq S_2$.*

Proof. By the determinization procedure, we have:

- $S_1 \xrightarrow{a} S'_1 \in \Delta'_i$ where $S'_1 = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_1, q'' \xrightarrow{a} q' \in \Delta_i\}$.
- $S_2 \xrightarrow{a} S'_2 \in \Delta'_i$ where $S'_2 = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_2, q'' \xrightarrow{a} q' \in \Delta_i\}$.

Thus, it is easy to verify that $S'_1 \leq S'_2$ if $S_1 \leq S_2$. □

Similarly, for the cases of push and pop transitions, the next two lemmas hold:

Lemma 8. *Let $S_1 \xrightarrow{a/+(S_1,a)} Id_{R'_1}$ and $S_2 \xrightarrow{a/+(S_2,a)} Id_{R'_2}$ be push transitions of the determinized VPA M^{od}. If $S_1 \leq S_2$, we have $Id_{R'_1} \leq Id_{R'_2}$.*

Lemma 9. *Let $S_1 \xrightarrow{a/-(S'_1,a')} S''_1$ and $S_2 \xrightarrow{a/-(S'_2,a')} S''_2$ be pop transitions of the determinized VPA M^{od}. Assume that $(S'_1, a') \leq (S'_2, a')$ and $S_1 \leq S_2$. Then, we have $S''_1 \leq S''_2$.*

Now, we are in a position to extend the ordering in Definition 6 to an ordering over configurations of the determinized VPA M^{od}.

Definition 10. *Let $c_1 = (S_1, \gamma_n \cdots \gamma_1 \bot)$ and $c_2 = (S_2, \gamma'_n \cdots \gamma'_1 \bot)$ be two configurations of M^{od}. We say $c_1 \leq c_2$ iff the following conditions hold: $S_1 \leq S_2$, and $\gamma_i \leq \gamma'_i$ for all $1 \leq i \leq n$.*

Lemma 11. *Let $c_1 = (S_1, \gamma_n \cdots \gamma_1 \bot)$ and $c_2 = (S_2, \gamma'_n \cdots \gamma'_1 \bot)$ be configuration of M^{od} such that $c_1 \leq c_2$. For any word $w = a_1 \cdots a_k \in \Sigma^*$, if $(S_1, \gamma_n \cdots \gamma_1 \bot) \xrightarrow{w} (\bar{S}_1, \bar{\gamma}_m \cdots \bar{\gamma}_1 \bot)$ and $(S_2, \gamma'_n \cdots \gamma'_1 \bot) \xrightarrow{w} (\bar{S}_2, \bar{\gamma}'_m \cdots \bar{\gamma}'_1 \bot)$, then $(\bar{S}_1, \bar{\gamma}_m \cdots \bar{\gamma}_1 \bot) \leq (\bar{S}_2, \bar{\gamma}'_m \cdots \bar{\gamma}'_1 \bot)$*

Proof. We prove this lemma by induction on the length $|w|$ of w. If $|w| = 1$, it means that $w = a$. The proof immediately follows from Lemma 7, Lemma 8, or 9 wrt. the type of input symbol a. Now, assume that the lemma holds for the case $|w| = i$. Again, using induction hypothesis and Lemmas 7, 8 or 9, it is easy to verify that this lemma also holds for the case $|w| = i + 1$. The lemma is proved. □

Algorithm 3. Extract minimal transitions of P-automata at each incremental step

```
Data: A set of transitions $T(A)$ of $\mathcal{P}$-automaton $A$
Result: A set of "minimal" transitions T(A) of $A$
1 begin
2     for each state $S$ of $A$ do
3         if $(S_1, \gamma_1, S) \in T(A) \wedge (S_2, \gamma_2, S) \in T(A)$ such that: $S_1 \leq S_2$ and
          $\gamma_1 \leq \gamma_2$ then
4             $T(A) \longleftarrow T(A) \backslash \{(S_2, \gamma_2, S)\}$.
5     return T(A);
```

It is crucial to note that, $L(M) \neq \Sigma^*$, iff there exists a *rejecting* reachable configuration of M^{od}. Recall that a configuration (S, σ) is rejecting if $\Pi_2(S) \cap F = \varnothing$. Note that if $(S, \sigma) \leq (S', \sigma')$ and $\Pi_2(S') \cap F = \varnothing$, then $\Pi_2(S) \cap F = \varnothing$. Based on this observation and Lemma 11, it is sufficient to compute only minimal reachable configurations and check for the existence of a *rejecting* configuration. Formally, we define the set of *minimal reachable* configurations of a determinized VPA N as follows:

Definition 12. $\mathsf{MinimalReachableConf}(N) = \{(s, \sigma) \in ReachableConf(N) \mid \neg\exists(s', \sigma') \in ReachableConf(N) \cdot (s', \sigma') \leq (s, \sigma)\}$.

Let $C_0 = \{(Id_{Q_0}, \bot)\}$ be the set of initial configurations of M^{od}. Let $A_{post^*(C_0)}$ be the $\mathcal{P}$-automaton for presenting the set of $\mathsf{ReachableConf}(M^{od})$. Let A be the $\mathcal{P}$-automaton that is obtained from $A_{post^*(C_0)}$ at each incremental expansion step as follows: for two configurations $(S_1, \gamma_1\sigma)$ and $(S_2, \gamma_2\sigma)$, we only need to compare the states (*i.e.,* S_1 and S_2) and top-of-stack symbols (*i.e.,* γ_1 and γ_2). Assume that $S_1 \leq S_2$ and $\gamma_1 \leq \gamma_2$, then $(S_1, \gamma_1\sigma) \leq (S_2, \gamma_2\sigma)$. So, we only need to keep the "smaller" configuration $(S_1, \gamma_1\sigma)$. We formalize this observation in Algorithm 3.

Algorithm 4. The optimized on-the-fly algorithm for university checking of VPA

```
Data: A nondeterministic VPA $M = (Q, \Gamma, Q_0, \Delta, F)$
Result: Universality of $M$
1  begin
2      Create the initial state of the minimal determinized VPA $Md$ using
       Algorithm 1;
3      Initiate $\mathcal{P}$-automaton $A$ to represent the initial configuration of $Md$;
4      Create transitions of $Md$ departing from the initial state using Algorithm
       1;
5      while (the set of new transitions of $Md$ is not empty) do
6          Apply Algorithm 2 on the current $Md$ to create new transitions and
           states of $A$;
7          Then apply Algorithm 3 on the current $A$ to obtain the newly
           optimized $\mathcal{P}$-automaton $A$;
8          if a rejecting state is added to $A$ then
9              return False;
10         Apply Algorithm 1 on new states of $A$ to create new transitions of $Md$;
11     return True;
```

Theorem 13 (Correctness of Algorithm 4). *Let M be a nondeterministic VPA. Let M_i and A_i be deteminized VPA and the P-automaton obtained in the i^{th} repetition of the while-loop in Algorithm 4. The VPA M is not universal if and only if $L(A_i) \cap \mathsf{RejectingConf}(M^i) \neq \emptyset$ for some i.*

Proof. If $L(A_i) \cap \mathsf{RejectingConf}(M_i) \neq \varnothing$ for some i then of course M is not universal. Conversely, if M is not universal, there exists at least one rejecting reachable configuration $q = (S, \sigma)$ (i.e., $\Pi_2(S) \cap F = \emptyset$). If $q \in L(A_i)$ for some i, then the theorem holds. If $q \notin L(A_i)$ for any i (within a bound), it means that q was out of $L(A_i)$ by a removing action at a certain step of the incremental expansion using Algorithm 3. Thus, there is at least one reachable configuration $q' = (S', \sigma')$, such that $q' \leq q$ and $q' \in L(A_i)$. Because $q' \leq q$, $S' \leq S$ and $\Pi_2(S') \cap F = \emptyset$, and thus q' is a rejecting configuration. As a result, there exists an i such that $q' \in L(A_i) \cap \mathsf{RejectingConf}(M_i)$, the theorem holds. □

Complexity: In the worst case (i.e., the automaton is universal), the complexity of our proposed algorithm is the same as of the standard one, $O(2^{3n^2})$ where n are numbers of states of M (this is because checking emptiness of VPA is cubic time [2]). However, in the case of not universal, our methods outperforms the standard one as the experiments will show in the next section.

3.3 Inclusion Checking

To check whether $L(A) \subseteq L(B)$, the standard method is to check whether $L(A \times \overline{B}) = \emptyset$, where $\overline{B}$ is the complement of B. For inclusion checking of VPA, we first determinize B, take its complement $\overline{B}$) and make product with A incrementally step by step (denoted by $A \times \overline{B}$). Concurrently, we check for reachability of this product automaton using on-the-fly manner. If there is a reachable state (q, s) such that $q \in F_A$ and $s \cap F_B = \emptyset$ (s is an accepting state of $\overline{B}$). In this case, there exists a word w such that $w \in L(A)$ and $w \notin L(B)$. In this case we stop and report that $L(A) \nsubseteq L(B)$. Although our formalization is different from the antichain of finite automata [7], the intuition behind our method is the same as theirs. In other words, the on-the-fly approach tries to find if there exists at least a word $w \in L(A) \setminus L(B)$. If such a word w was found, we can conclude that $L(A) \nsubseteq L(B)$. Otherwise, $L(A)$ is a subset of $L(B)$. In the worst case, the complexity is $O(m^3 \cdot 2^{3n^2})$ for VPA inclusion checking, where m and n are numbers of states of A and B.

4 Implementation and Experiments

We have implemented the above approaches, on the top of VPAlib [1], for testing universality and inclusion of VPA in a prototype tool named VPAchecker. The package is implemented in Java 1.5.0 on Windows XP. To compare the optimized algorithm with the standard algorithm, we run our implementations on randomly generated VPA. All tests are performed on a PC equipped with 1.50 GHz Intel® Core™ Duo Processor L2300 and 1.5 GB of memory. During experiments, we fix the size of the input alphabet to $|\Sigma_c| = |\Sigma_r| = |\Sigma_i| = 2$, and the size of the stack alphabet to $|\Gamma| = 3$. The *density of final states* $f = \frac{|F|}{|Q|}$ is the ratio of number of

[1] http://www.emn.fr/x-info/hnguyen/vpa/

Table 1. Universality checking for VPA generated by r-random (50 automata for each sample)

	number of states							
ON-THE-FLY-OPT	5	10	20	40	60	80	100	150
no. of success	50	50	50	50	50	50	50	47
total time (s)	19	35	43	87	147	222	496	336
no. of timeout (*60* s)	0	0	0	0	0	0	0	3
	number of states							
ON-THE-FLY	5	10	20	40	60	80	100	150
no. of success	50	50	50	50	50	50	46	47
total time	23	46	52	110	210	247	686	372
no. of timeout (*60* s)	0	0	0	0	0	0	4	3
	number of states							
STANDARD	5	10	20	40	60	80	100	150
no. of success	21	1	0	0	0	0	0	0
total time (s)	456	31	0	0	0	0	0	0
no. of timeout (*60* s)	29	49	50	50	50	50	50	50

Table 2. Universality checking for VPA generated by r-regular random (50 automata for each sample)

	number of states						
ON-THE-FLY-OPT	5	10	15	20	30	40	50
no. of success	50	45	19	2	0	0	0
total time (s)	52	833	892	86	0	0	0
no. of timeout (*180* s)	0	5	31	48	50	50	50
	number of states						
ON-THE-FLY	5	10	15	20	30	40	50
no. of success	50	43	13	0	0	0	0
total time	68	1425	754	0	0	0	0
no. of timeout (*180* s)	0	7	37	50	50	50	50
	number of states						
STANDARD	5	10	15	20	30	40	50
no. of success	0	0	0	0	0	0	0
total time (s)	0	0	0	0	0	0	0
no. of timeout (*180* s)	50	50	50	50	50	50	50

Table 3. Checking inclusion with $r(q, a) = 2$, $f = 1$

	number states of A and B				
ON-THE-FLY-OPT	(10,5)	(100,5)	(500,5)	(1000,5)	(3000,5)
success	20	20	5	5	2
time(s)	27	910	336	1257	357
timeout (*300* s)	0	0	15	15	18

final states over the number of states of a VPA. We first set parameters of the tests as follows:

Definition 14 (r-random). *The* density of final states $f = \frac{|F|}{|Q|} = 1$ *and the* density of transitions $r = \frac{k_a}{|Q|} = 2$*, where k_a is the number of transitions for each input symbol a, F and Q are set of final states and states of VPA, respectively.*

We ran our tests on randomly VPA generated by the parameter r-random. We have tried VPA sizes from 10 to 100. We generated 50 VPA for each sample point, and setting timeout to 60 seconds. The experimental results are given in Table 1. We found that all successfully checked VPA are not universal, and thus we omit the row for universal results in the table. The experiments shows that STANDARD can solve for generated VPA instances with 5 states only. It becomes stuck when the number of states greater than or equal to 10. Meanwhile, ON-THE-FLY is significantly more efficient than STANDARD, it can check for almost all VPA. We also applied optimization of $\mathcal{P}$-automaton and implemented it as ON-THE-FLY-OPT. Based on experimental results, ON-THE-FLY-OPT is a little bit better than ON-THE-FLY. The parameter r-random does not guarantee the completeness of VPA. Therefore, the probability of being universal is very low. In order to increase the probability of being universal, we define a new parameter as below:

Definition 15 (r-regular random). *The* density of final states$f = \frac{|F|}{|Q|} = 1$ *and the* density of transitions $r : Q \times \Sigma \to N$*; $r(q, a)$ depends on not only the input symbol a but also on the state q. In particular, we select: $r(q, a) = 2$ for all $q \in Q$ and $a \in \Sigma_c$, $r(q, b) = 6$ for all $q \in Q$ and $b \in \Sigma_r$, and $r(q, c) = 2$ for all $q \in Q$ and $c \in \Sigma_i$.*

With r-regular random, a VPA with 10 states has 200 transitions. We again test for various sizes of VPA from 5 to 50. We ran with 50 samples for each point, setting timeout to 180 seconds. The results are reported in Table 2. For this parameter, results of STANDARD are all timeout even with only 5 states. ON-THE-FLY behaves in significantly better ways than those of STANDARD. Since VPA generated by r-regular random parameter are universal, all three algorithms work more slowly. Especially, if the number of states is greater than 30, results of ON-THE-FLY are all timeout. Similarly, ON-THE-FLY-OPT is a little bit better than ON-THE-FLY for this parameter.

We also performed experiments for inclusion checking $L(A) \subseteq L(B)$. For this, we selected parameter r-regular random for generating the most difficult instances of inclusion checking. This is because using r-regular random both A and B are universal, and thus $L(A)$ is included in $L(B)$. We generated various sizes of A (10, 100, 200, 500, 1000, and 3000 states) and B (5 and 10 states). We ran with 20 samples for each point, setting timeout to 300 seconds. The experimental results are summarized in Table 3. There we only list the number of tests that finish within timeout. The detailed results are reported in Table 3. We see that STANDARD does not work well, it get all timeout for the smallest size $(10, 5)$. The results show that ON-THE-FLY-OPT outperforms STANDARD.

5 Conclusion

In this paper we presented the optimized on-the-fly algorithms for testing universality and inclusion of nondeterministic VPA. In summary, to check universality of a nondeterministic VPA M, the intuition idea behind on-the-fly manner is try to find whether there exists a word w such that $w \notin L(M)$. Similarly, to check inclusion $L(M) \subseteq L(N)$, the intuition idea is to find whether there exists at least a word w such that $w \in L(M) \setminus L(N)$. All algorithms have been implemented in a prototype tool. Although the ideas of our methods are simple, the experimental results showed that the proposed algorithms are considerably faster than the standard ones.

We should emphasize that much work needs to be done in the future, which includes, *e.g.,* (1) consider how to apply the tool to case studies in practice, for example, checking correctness requirements for XML streams. At the moment, the data structures for VPA are rather naive. That is reason why the running time of our tool is not fast. It would be interesting to explore a more compact data structure; for this, (2) we plan to manipulate VPA using BDD-based representation. Currently, we have been working on aspects (1) and (2). The preliminary results related to these mentioned future work will be reported in another opportunity.

Acknowledgements. We would like to thank Prof. Mizuhito Ogawa, Dr. Nao Hirokawa of JAIST and anonymous reviewers for helpful comments to improve this paper.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Babai, L. (ed.) STOC, pp. 202–211. ACM (2004)
2. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model Checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
3. Kumar, V., Madhusudan, P., Viswanathan, M.: Visibly pushdown automata for streaming xml. In: WWW, pp. 1053–1062 (2007)
4. Nguyen, D.H., Südholt, M.: Vpa-based aspects: better support for aop over protocols. In: SEFM, pp. 167–176. IEEE Computer Society (2006)
5. Nguyen, T.V.: A tighter bound for determinization of visibly pushdown automata. In: INFINITY, vol. 10, pp. 62–76 (2009)
6. Pitcher, C.: Visibly pushdown expression effects for xml stream processing. In: PLAN-X, pp. 5–19 (2005)
7. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)