Visibly Pushdown Automata: Universality and Inclusion via Antichains

Véronique Bruyère[1], Marc Ducobu[1], and Olivier Gauwin[2]

[1] University of Mons, Belgium
[2] LaBRI, University of Bordeaux, France

Abstract. Visibly pushdown automata (VPAs), introduced by Alur and Madhusudan in 2004, are a useful formalism in various contexts, such as expressing and checking properties on control flows of programs, or on XML documents. In this context, we propose efficient antichain-based algorithms to check universality and inclusion of VPAs. Whereas the computation complexity is known to be ExpTime-complete for both problems, we show how antichains can avoid explicit determinization and save computations. The approach is extended to hedge automata. We implement the proposed algorithms in a prototype tool and conduct experiments on randomly generated VPAs. We show that, on numerous instances, our algorithms outperform other VPA tools.

1 Introduction

The model-checking framework provided many successful tools for decades, starting from the seminal work of Büchi. A lot of them rely on the links between logics used to express properties on words, and automata allowing to check them. Some of these results have been adapted to trees, and more recently to words with a nesting structure.

Visibly pushdown automata (VPAs) have been introduced to process such words with nesting [1]. VPAs are similar to pushdown automata, but operate on a partitioned alphabet: a given letter is associated with one action (push or pop), and thus cannot push when firing a transition, and pop when firing another. Such automata were introduced to express and check properties on control flows of programs, where procedure calls push on the stack, and returns pop [2]. They are also suitable to express properties on XML documents [3]. These documents are usually represented as trees, but are serialized as a sequence of opening and closing tags, also called the *linearization* of this document, or its corresponding *XML stream*.

The model-checking framework with VPAs is confronted to the computational hardness of testing universality and inclusion. These two problems are ExpTime-complete on non-deterministic VPAs, due to the expensive determinization step [1]. Non-determinism naturally arises when automata are obtained from logic formulas, as for instance XPath expressions with descendant axis [4].

In this paper we propose antichain-based algorithms for deciding universality and inclusion of VPAs. We use *antichains* to get smaller objects to manipulate

and to avoid an explicit determinization step. Recently, antichains have been successfully applied to decision problems related to non-deterministic automata: universality and inclusion for finite word automata [5], and for non-deterministic bottom-up tree automata [6]. Some simulation relations are also known on unranked trees [7] but it is unclear whether they can help for our problems, as they do in other contexts [8, 9].

Nguyen [10] proposed an algorithm for testing the universality of VPAs. This algorithm simultaneously performs an on-the-fly determinization and reachability checking by $\mathcal{P}$-automaton. The notion of $\mathcal{P}$-automaton proposed in [11] provides a symbolic technique to compute the sets of all reachable configurations of a VPA. This algorithm has been later improved by Nguyen and Ohsaki [12] by introducing antichains over transitions of $\mathcal{P}$-automata, in a way to generate the smallest amount of reachable configurations. Our algorithms for universality are alternative to this one since we do not use the regularity property of the set of reachable configurations. When observed on trees, their approach follows forward steps on the linearization of trees, while our approach is bottom-up on the structure of trees.

In [13], the authors provide solutions for checking universality and inclusion of VPAs over finite and infinite words. They avoid the determinization and complementation steps, and use instead Ramsey-based universality- and inclusion-checking algorithms. Their algorithms do not seem to use antichains.

The paper is structured as follows. In Section 2 we define unranked trees and visibly pushdown automata seen as trees acceptors. In Section 3, we detail our antichain-based algorithm for checking universality of VPAs, together with several optimizations. Section 4 contains extensions of this algorithm to general VPAs and hedge automata. It also proposes an antichain-based algorithm for testing the inclusion of VPAs. Section 5 is devoted to the experiments and comparisons with other prototype tools.

2 Preliminaries

2.1 Unranked Trees

We here recall the standard definition of unranked trees, as provided for instance in [14]. Let Σ be a finite *alphabet*, and Σ^* (resp. Σ^+) be the set of all words (resp. non empty words) over Σ. The empty word is denoted by ϵ.

An *unranked tree* t over Σ is a partial function $t : \mathbb{N}^* \to \Sigma$ such that the domain is non-empty, finite and prefix-closed. The domain is denoted by $nodes(t)$ and contains the *nodes* of the tree t, with the root being the empty word ϵ. The function t labels each node p with a letter $t(p)$ of Σ. The set of all unranked trees over Σ is denoted by T_Σ. A *hedge* h over Σ is a finite sequence (empty or not) of unranked trees over Σ. The empty hedge is denoted by ϵ, and the set of all hedges over Σ is denoted by H_Σ.

Given a tree t, the *subtree* of t rooted at node p of t is the tree denoted by $t_{|p}$, which domain is the set of nodes p' such that $pp' \in nodes(t)$ and verifying

$t_{|p}(p') = t(pp')$. For a given node $p \in nodes(t)$, we call *children* of p the nodes $pi \in nodes(t)$ for $i \in \mathbb{N}$, and use the usual definitions for parents, ancestors and descendants. The *height* of a tree, and more generally of a hedge, is the length of its longest branch (with the length being the number of nodes).

Trees can be described by well-balanced words which correspond to a depth-first traversal of the tree. An opening tag is used to notice the arrival on a node and a closing tag to notice the departure from a node. For each $a \in \Sigma$, let a itself represent the opening tag and $\overline{a}$ the related closing tag. The *linearization* $[t]$ of $t \in T_\Sigma$ is the *well-balanced* word over $\Sigma \cup \overline{\Sigma}$, with $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$, inductively defined by: $[t] = a\ [t_{|1}] \cdots [t_{|n}]\ \overline{a}$, with $a = t(\epsilon)$ and the root has n children.

2.2 Visibly pushdown automata

Visibly pushdown automata (VPAs, [1, 15]) are pushdown automata operating on a partitioned alphabet where only call symbols can push, return symbols can pop, and internal symbols can do transitions without considering the stack. For clarity we only consider languages of unranked trees, so we use VPAs as *unranked trees acceptors*, operating on their linearization [16].[3]

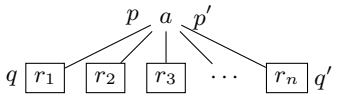
Definition 1. *A* visibly pushdown automaton $\mathcal{A}$ *over a finite alphabet* Σ *is a tuple* $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ *where* Q *is a finite set of states containing initial states* $Q_i \subseteq Q$ *and final states* $Q_f \subseteq Q$*, a finite set* Γ *of stack symbols, and a finite set* Δ *of rules. Each rule in* Δ *is of the form* $q \xrightarrow{a:\gamma} q'$ *with* $a \in \Sigma \cup \overline{\Sigma}$*,* $q, q' \in Q$*, and* $\gamma \in \Gamma$*.*

A *configuration* of a VPA $\mathcal{A}$ is a pair (q, σ) where $q \in Q$ is a state and $\sigma \in \Gamma^*$ a stack content. A configuration is *initial* (resp. *final*) if $q \in Q_i$ (resp. $q \in Q_f$) and $\sigma = \epsilon$. For $a \in \Sigma \cup \overline{\Sigma}$, we write $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if there is a transition $q \xrightarrow{a:\gamma} q'$ in Δ verifying $\sigma' = \gamma \cdot \sigma$ if $a \in \Sigma$, and $\sigma = \gamma \cdot \sigma'$ if $a \in \overline{\Sigma}$.

A *run* of a VPA $\mathcal{A}$ on a word $a_1 \cdots a_n \in (\Sigma \cup \overline{\Sigma})^*$ is a sequence of configurations (q_i, σ_i), $0 \leq i \leq n$, such that $(q_{i-1}, \sigma_{i-1}) \xrightarrow{a_i} (q_i, \sigma_i)$ for all i. It is denoted by $(q_0, \sigma_0) \xrightarrow{a_1 \cdots a_n} (q_n, \sigma_n)$. It is *accepting* if $a_1 \cdots a_n$ is the linearization of a tree $t \in T_\Sigma$, (q_0, σ_0) is initial, and (q_n, σ_n) is final. A tree $t \in T_\Sigma$ is *accepted* by $\mathcal{A}$ if there is an accepting run on its linearization $[t]$. The set of accepted trees is called the *language* of $\mathcal{A}$ and is written $L(\mathcal{A})$.

A VPA $\mathcal{A}$ is said *universal* if it accepts all trees $t \in T_\Sigma$. Our objective in the next section, is to propose efficient algorithms for testing universality of VPAs. A standard method to check universality of a VPA is to determinize it, complement it, and check for emptiness. As determinization is in exponential time for VPAs, the universality problem is ExpTime-complete. Our algorithms aim at avoiding this exponential blow-up on numerous instances.

[3] Our algorithms can easily be adapted to usual VPAs, as explained in Section 4.2.



$$p \xrightarrow{a:\gamma} q \in \Delta,$$
$$q' \xrightarrow{\overline{a}:\gamma} p' \in \Delta,$$
$$(q, q') \in r_1 \circ r_2 \circ \cdots \circ r_n$$

Fig. 1: $(p, p') \in Post_a(r_1 \circ \cdots \circ r_n)$.

3 Checking universality

In our approach for checking universality of VPAs, the main idea is to find as fast as possible a tree which linearization is rejected by the automaton (if it exists), without computing an explicit determinization of the VPA. Antichains will limit the computations. Proofs of results in this section are given in Appendix A.

3.1 Accessibility relation

Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA, and t be a tree over Σ. We define the *accessibility relation* $Acc(t) \subseteq Q \times Q$ as the set

$$Acc(t) = \{(q, q') \in Q \times Q \mid (q, \epsilon) \xrightarrow{[t]} (q', \epsilon)\}.$$

Note that in this paper, accessibility means *accessibility through the linearization of a tree*, as opposed to the accessibility between configurations of the VPA. With this notation, a tree t is accepted by $\mathcal{A}$ iff $Acc(t) \cap Q_i \times Q_f \neq \emptyset$.

Let us show how the accessibility relation $Acc(t)$ can be computed for all $t \in T_\Sigma$. In this aim, we need to introduce the *Post* operator. For a set $\mathscr{R}$ of relations $r \subseteq Q \times Q$, let $\mathscr{R}^*$ denote the set of all relations obtained by composing elements of $\mathscr{R}$: $\mathscr{R}^* = \{r_1 \circ r_2 \circ \cdots \circ r_n \mid n \geq 0 \text{ and } r_i \in \mathscr{R} \text{ for all } 1 \leq i \leq n\}$. In particular $\mathscr{R}^*$ contains the identity relation id_Q over Q, obtained when $n = 0$.

Definition 2. *Given $r \in \mathscr{R}$ and $a \in \Sigma$, let*

$$Post_a(r) = \{(p, p') \in Q \times Q \mid \exists (q, q') \in r, p \xrightarrow{a:\gamma} q \in \Delta, q' \xrightarrow{\overline{a}:\gamma} p' \in \Delta\}.$$

For $\mathscr{R}$ a set of relations over Q, let

$$Post(\mathscr{R}) = \{Post_a(r) \mid a \in \Sigma, r \in \mathscr{R}^*\} \cup \mathscr{R}$$

and $Post^*(\mathscr{R}) = \cup_{i \geq 0} Post^i(\mathscr{R})$ such that $Post^0(\mathscr{R}) = \mathscr{R}$, and for all $i > 0$, $Post^i(\mathscr{R}) = Post(Post^{i-1}(\mathscr{R}))$.

We illustrate the definition of *Post* in Figure 1. The following lemmas relate these operators with the accessibility relation.

Lemma 1. *Let $t \in T_\Sigma$ be such that its root is a node with n children that is labeled by a. Let $r_i = Acc(t_{|i})$ for $1 \leq i \leq n$. Then $Acc(t) = Post_a(r_1 \circ \cdots \circ r_n)$.*

Lemma 2. *$Post^i(\emptyset) = \{Acc(t) \mid t \text{ is a tree with height} \leq i\}$.*

The next proposition is an immediate consequence of Lemmas 1 and 2.

Proposition 1. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA. Then $\mathcal{A}$ is universal iff $\forall r \in Post^*(\emptyset)$, $r \cap Q_i \times Q_f \neq \emptyset$.*

3.2 Algorithm

We are now able to propose an algorithm to check the universality of VPAs. With Algorithm 1, the set $Post^*(\emptyset)$ is computed incrementally and the universality test is performed thanks to Proposition 1. More precisely, at step i, the variable $\mathscr{R}$ is used for $Post^i(\emptyset)$, and the variable $\mathscr{R}^*$ contains its closure by composition. We compute $\mathscr{R}^*$ with Function CompositionClosure, and then potential new relations with $\{Post_a(r) \mid a \in \Sigma, r \in \mathscr{R}^*\}$. The algorithm stops when a relation proving non-universality is found, or no new relation can be produced.

Algorithm 1 Checking universality

```
function Universality(A)
    R ← ∅
    R* ← {id_Q}
    repeat
        R_new ← {Post_a(r) | a ∈ Σ, r ∈ R*} \ R
        if ∃r ∈ R_new : r ∩ Q_i×Q_f = ∅ then
            return False    // Not universal
        end if
        R ← R ∪ R_new
        R' ← R_new \ R*
        if R' ≠ ∅ then
            R* ← CompositionClosure(R*, R')
        end if
    until R' = ∅
    return True    // Universal
end function
```

Let us detail Function CompositionClosure$(\mathscr{R}^*, \mathscr{R}')$ which computes the set $(\mathscr{R}^* \cup \mathscr{R}')^*$. In Algorithm 2, we show how to compute this set without recomputing $\mathscr{R}^*$ from $\mathscr{R}$. Initially, *Relations* is equal to $\mathscr{R}^* \cup \mathscr{R}'$ and will be equal to $(\mathscr{R}^* \cup \mathscr{R}')^*$ at the end of the computation. *ToProcess* contains the relations that can produce new relations by composition with an element of *Relations*.

Proposition 2. *Given $\mathscr{R}^*$ and $\mathscr{R}'$, Algorithm 2 computes $(\mathscr{R}^* \cup \mathscr{R}')^*$.*

3.3 Antichain-based optimization

In this section we explain how to use the concept of antichain for saving computations. We show that it is sufficient to only compute the $\subseteq$-minimal elements of $Post^*(\emptyset)$ for checking universality.

Consider the set $2^{Q \times Q}$ of all binary relations over Q. This set is equipped with the $\subseteq$ operator such that $r \subseteq r'$ iff $(q, q') \in r \Rightarrow (q, q') \in r'$. An *antichain* $\mathscr{R}$ of relations over Q is a set of pairwise incomparable relations with respect to $\subseteq$. Given a set $\mathscr{R}$ of relations, we denote by $\lfloor \mathscr{R} \rfloor$ the $\subseteq$-*minimal* elements of $\mathscr{R}$, similarly we denote by $\lceil \mathscr{R} \rceil$ the $\subseteq$-*maximal* elements of $\mathscr{R}$.

Algorithm 2 Computing $(\mathscr{R}^* \cup \mathscr{R}')^*$

```
function CompositionClosure(R*, R')
    Relations ← R*
    ToProcess ← R'
    while ToProcess ≠ ∅ do
        rel ← Pop(ToProcess)
        R* ← R* ∪ {rel}
        NewRelations ← ∅
        for r ∈ Relations do
            NewRelations ← NewRelations ∪ {r ∘ rel, rel ∘ r}
        end for
        ToProcess ← ToProcess ∪ (NewRelations \ Relations)
        Relations ← Relations ∪ NewRelations
    end while
    return Relations
end function
```

Definition 3. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA, and $\mathscr{R}$ be a set of relations over Q. Let $Post^*_{min}(\mathscr{R}) = \cup_{i \geq 0} Post^i_{min}(\mathscr{R})$ such that $Post^0_{min}(\mathscr{R}) = \lfloor \mathscr{R} \rfloor$, and for all $i > 0$, $Post^i_{min}(\mathscr{R}) = \lfloor Post(Post^{i-1}_{min}(\mathscr{R})) \rfloor$.*

Lemma 3. *Given $\mathscr{R}$ a set of relations over Q, for all $r \in Post^*(\mathscr{R})$, there exists $r' \in Post^*_{min}(\mathscr{R})$ such that $r' \subseteq r$.*

We have the next counterpart of Proposition 1.

Proposition 3. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA. Then $\mathcal{A}$ is universal iff $\forall r \in Post^*_{min}(\emptyset), r \cap Q_i \times Q_f \neq \emptyset$.*

Algorithm 3 checks whether a VPA is universal by computing incrementally $Post^*_{min}(\emptyset)$. It is an adaptation of Algorithm 1. Notice that we can compute $\lfloor Post(\mathscr{R}) \rfloor$ as $\lfloor \{Post_a(r) \mid a \in \Sigma, r \in \lfloor \mathscr{R}^* \rfloor\} \cup \mathscr{R} \rfloor$ (we limit the computation to $r \in \lfloor \mathscr{R}^* \rfloor$). The set $\lfloor \mathscr{R}^* \rfloor$ is denoted by $\mathscr{R}^*_{min}$ in the algorithm.

3.4 Other optimizations

Algorithms 2 and 3 can be optimized in several directions. The optimized algorithm is given in Appendix C.

Witness of non universality - As soon as a new relation r is yielded (via the *Post* operator or the composition of two relations), the emptiness of its intersection with $Q_i \times Q_f$ is *immediately* tested, to check whether it is a witness of non universality.

In Algorithm 2, the relation *rel* is composed with all relations $r \in \mathscr{R}$. We order the set $\mathscr{R}$ in a way to first treat the relations r that are the *most promising* to yield a witness of non universality. This ordering is based on the sizes of the domain and codomain of the relations. Indeed, the compositions $r \circ rel$, $rel \circ r$ with a relation r with small such sizes are usually smaller, and thus more incline to not intersecting $Q_i \times Q_f$.

Algorithm 3 Checking universality

```
function UNIVERSALITY(A)
    R ← ∅
    R*_min ← {id_Q}
    repeat
        R_new ← ⌊{Post_a(r) | a ∈ Σ, r ∈ R*_min}⌋ \ R
        if ∃r ∈ R_new : r ∩ Q_i×Q_f = ∅ then
            return False    // Not universal
        end if
        R ← ⌊R ∪ R_new⌋
        R' ← R_new \ R*_min
        if R' ≠ ∅ then
            R*_min ← ⌊COMPOSITIONCLOSURE(R*_min, R')⌋
        end if
    until R' = ∅
    return True    // Universal
end function
```

Data structures - Efficient data structures are used both for the relations and the antichains. A relation is stored as an array of *bit-vectors*. In this way the composition is computed efficiently using bit-operations, as well as its domain and codomain.

A *hash table* is used to store an antichain, such that relations with different domain or codomain are stored in different lists. In this way, comparing a new relation r with the elements of the antichain is made more efficient, by limiting the comparison of r with elements of the same domain and codomain.

To compute $\lfloor \mathscr{R}^* \rfloor$ in Algorithm 3, we first make a call to Function CompositionClosure, and then keep the $\subseteq$-minimal elements of the result. One optimization is, at *each* step of the CompositionClosure computation, to only consider the minimal elements.

4 Extensions

4.1 Checking inclusion

An antichain-based algorithm for testing the inclusion of two VPAs can be designed, following the same ideas as for testing universality. We here give the main ideas for VPAs accepting linearizations of trees. For two VPAs $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \Gamma_\mathcal{A}, Q_{i,\mathcal{A}}, Q_{f,\mathcal{A}}, \Delta_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \Gamma_\mathcal{B}, Q_{i,\mathcal{B}}, Q_{f,\mathcal{B}}, \Delta_\mathcal{B})$ over the same alphabet Σ, we have $L(\mathcal{A}) \not\subseteq L(\mathcal{B})$ iff there exists in $\mathcal{A}$ an accepting run on some word w such that all runs on w in $\mathcal{B}$ are not accepting. For a tree t over Σ, instead of considering the accessibility relation $Acc(t)$ as defined in Section 3.1, we consider pairs $((q, q'), r) \in (Q_\mathcal{A} \times Q_\mathcal{A}) \times 2^{Q_\mathcal{B} \times Q_\mathcal{B}}$ such that $(q, q') \in Acc_\mathcal{A}(t)$ and $r = Acc_\mathcal{B}(t)$ for the same tree t. In this way, $t \in L(\mathcal{A}) \setminus L(\mathcal{B})$ iff $(q, q') \in Q_{i,\mathcal{A}} \times Q_{f,\mathcal{A}}$ for some $(q, q') \in Acc_\mathcal{A}(t)$, and $Acc_\mathcal{B}(t) \cap Q_{i,\mathcal{B}} \times Q_{f,\mathcal{B}} = \emptyset$.

Given a set $\mathscr{S} \subseteq (Q_\mathcal{A} \times Q_\mathcal{A}) \times 2^{Q_\mathcal{B} \times Q_\mathcal{B}}$, we define $\mathscr{S}^*$ as the set $\{((p,p'),s) \mid$ there exist $n \geq 0, ((q_i,q_i'),r_i) \in \mathscr{S}$ with $q_i' = q_{i+1}$ for all $1 \leq i < n, p = q_1, p' = q_n,$ and $s = r_1 \circ \cdots \circ r_n\}$. In particular $\mathscr{S}^*$ contains the pairs $((q,q), id_{Q_\mathcal{B}})$ for all $q \in Q_\mathcal{A}$, obtained when $n = 0$.

The *Post* operator is adapted to $Post_\subseteq$ as follows. In this definition, we use notation $Post^\mathcal{B}$ to denote the *Post* operator used in automaton $\mathcal{B}$.

Definition 4. *For $\mathscr{S} \subseteq (Q_\mathcal{A} \times Q_\mathcal{A}) \times 2^{Q_\mathcal{B} \times Q_\mathcal{B}}$, we define $Post_\subseteq(\mathscr{S}) = \mathscr{S}' \cup \mathscr{S}$, where $\mathscr{S}'$ is the set of pairs $((p,p'),s)$ such that*

$$p \xrightarrow{a:\gamma} q \in \Delta_\mathcal{A},\ q' \xrightarrow{\overline{a}:\gamma} p' \in \Delta_\mathcal{A} \text{ and } s = Post_a^\mathcal{B}(r)$$

for some $a \in \Sigma$ and $((q,q'),r) \in \mathscr{S}^*$. Let $Post_\subseteq^*(\mathscr{S}) = \cup_{i \geq 0} Post_\subseteq^i(\mathscr{S})$ such that $Post_\subseteq^0(\mathscr{S}) = \mathscr{S}$, and for all $i > 0$, $Post_\subseteq^i(\mathscr{S}) = Post_\subseteq(Post_\subseteq^{i-1}(\mathscr{S}))$.

Proposition 1 is adapted into the next proposition.

Proposition 4. *Let $\mathcal{A}$ and $\mathcal{B}$ be two VPAs. Then $L(\mathcal{A}) \subseteq L(\mathcal{B})$ iff $\forall((q,q'),r) \in Post_\subseteq^*(\emptyset)$, $(q,q') \in Q_{i,\mathcal{A}} \times Q_{f,\mathcal{A}} \ \Rightarrow\ r \cap Q_{i,\mathcal{B}} \times Q_{f,\mathcal{B}} \neq \emptyset$.*

An algorithm for testing the inclusion can be designed as done for universality. Again we can save computations by using antichains. In this context, the set $(Q_\mathcal{A} \times Q_\mathcal{A}) \times 2^{Q_\mathcal{B} \times Q_\mathcal{B}}$ is equipped with the $\subseteq$ operator such that $((q,q'),r) \subseteq ((p,p'),s)$ iff $(q,q') = (p,p'), r \subseteq s$. As done in Section 3.3, we can define $Post_{\subseteq,min}^*(\emptyset)$ and get the counterpart of Proposition 4 using antichains.

4.2 Universality of general VPAs

We considered VPAs as unranked trees acceptors, i.e. VPAs that only operate on linearizations of trees. We show here how our algorithms can be easily adapted to the original VPA model [1], accepting words with internal actions and pending calls or returns.

Three types of symbols - The original VPA model operates on an alphabet partitioned into three disjoint sets: a set Σ_c of call symbols, a set Σ_r of return symbols[4], and a set Σ_i of internal symbols that have no effect on the stack. Algorithm 1 (resp. Algorithm 3) can be adapted to VPAs with these three types of symbols as follows. We initialize $\mathscr{R}^*$ (resp. $\mathscr{R}_{min}^*$) to $\{id_Q\} \cup \bigcup_{a \in \Sigma_i} \{(q,q') \mid q \xrightarrow{a} q' \in \Delta\}$ instead of $\{id_Q\}$. Indeed, internal symbols can appear at any place in a word, so the relation $\{(q,q') \mid q \xrightarrow{a} q' \in \Delta\}$ has to be combined with any other yielded relation. We also adapt the *Post* operator by defining $Post_{a,\overline{b}}(r) = \{(p,p') \in Q \times Q \mid \exists (q,q') \in r,\ p \xrightarrow{a:\gamma} q \in \Delta,\ q' \xrightarrow{\overline{b}:\gamma} p' \in \Delta\}$ for all $a \in \Sigma_c$ and $\overline{b} \in \Sigma_r$.

[4] Σ_c and Σ_r are no longer related by the relation $\Sigma_c = \Sigma$ and $\Sigma_r = \overline{\Sigma}$.

Pending calls and returns - So far we considered linearizations of trees, but all definitions and results proposed above also hold for linearizations of hedges, formally defined by: $[t_1 \cdots t_n] = [t_1] \cdots [t_n]$. A word w over $\Sigma \cup \overline{\Sigma}$ is not necessarily the linearization of a hedge. The general shape of such a word w is:

$$w = [h_0]\overline{b}_0[h_1]\overline{b}_1 \cdots [h_m]\overline{b}_m[h]a_1[h'_1]a_2[h'_2] \cdots a_n[h'_n]$$

where all h_i, h'_j, and h are hedges of H_Σ, and $\overline{b_i} \in \overline{\Sigma}$, $a_j \in \Sigma$ for all i, j. In other words, w is a sequence of words $[h_i]\overline{b_i}$, followed by the linearization of a hedge $[h]$, followed by a sequence of words $a_j[h'_j]$. The idea here is to adapt Algorithm 1 so that it computes :

- as before, the set $\mathscr{R} = \{Acc(t) \mid t \in T_\Sigma\}$ of all accessibility relations through linearizations of trees, and its closure by composition $\mathscr{R}^*$. This latter set corresponds to the accessibility relation through linearizations of hedges.
- $\mathscr{C}^*$, the closure by composition of $\mathscr{C}$, which is the set of all accessibility relations through the linearization of a hedge, followed by a symbol in $\overline{\Sigma}$: $\mathscr{C} = \{Acc([h]\overline{b}) \mid h \in H_\Sigma,\ \overline{b} \in \overline{\Sigma}\}$.
- $\mathscr{O}^*$, the closure by composition of $\mathscr{O} = \{Acc(a[h]) \mid a \in \Sigma,\ h \in H_\Sigma\}$.

Each time a new relation r is added to $\mathscr{R}$, we update $\mathscr{R}^*$ as previously, using Function CompositionClosure. We also update $\mathscr{C}^*$ by first adding all relations $r \circ r_{\overline{b}}$ (instead of r) where $r_{\overline{b}} = \{(q, q') \mid q \xrightarrow{\overline{b}:\perp} q' \in \Delta\}$, and then compute the closure by composition of $\mathscr{C}^*$ as done for $\mathscr{R}^*$.The same method is used for $\mathscr{O}^*$, with relations $r_a \circ r$. Each time a new relation is added into one of these three sets ($\mathscr{R}^*$, $\mathscr{C}^*$ and $\mathscr{O}^*$), we check whether it is a witness of non-universality. Once these three sets are fully computed, we check whether all relations $r_c \circ r_h \circ r_o$ with $r_c \in \mathscr{C}^*$, $r_h \in \mathscr{R}^*$ and $r_o \in \mathscr{O}^*$ intersect $Q_i \times Q_f$.

4.3 Universality of hedge automata

The algorithms presented in this paper are easily adapted from VPAs to hedge automata [17, 14]. For clarity, we present hedge automata as unranked tree acceptors, but the result presented here also hold when considered as hedge acceptors. We recall in Appendix B the definition of hedge automata, and provide a translation of hedge automata to VPAs. Hence we can derive universality and inclusion algorithms for hedge automata, from the ones we propose for VPAs.

Let us briefly sketch how our algorithm operate on hedge automata. A hedge automaton $\mathcal{A}$ can be considered as a set of states Q and finite state automata $\mathcal{A}_{a,q}$ over Q: the state q can be assigned to a node labeled by a if states $(q_1, \ldots, q_n)$ can be respectively assigned to its n children, and $q_1 \cdots q_n \in L(\mathcal{A}_{a,q})$. We name *horizontal languages* the languages of automata $\mathcal{A}_{a,q}$, and write $t \underset{\mathcal{A}}{\hookrightarrow} q$ if q can be assigned to the root of t. The VPA resulting from the translation basically simulates all current horizontal languages, using the stack to store the current state of each level. Hence control states of the VPA are exactly control states of automata $\mathcal{A}_{a,q}$. Algorithm 1, when applied to (translations of) hedge automata,

builds binary relations r over states η, η' of automata $\mathcal{A}_{a,q}$: $(\eta, \eta') \in r$ iff there exists $t_1 \cdots t_n \in H_\Sigma$ for which there is a run $t_i \underset{\mathcal{A}}{\hookrightarrow} q_i$ for all $1 \leq i \leq n$ with $\eta \xrightarrow{q_1 \cdots q_n} \eta' \in \mathcal{A}_{a,q}$. From this, Algorithm 1 checks whether for every symbol $a \in \Sigma$, when performing a $Post_a$ operator on these relations, we obtain a relation intersecting $Q_i \times Q_f$.

5 Experiments

We have implemented the algorithms of Sections 3 and 4 with the described optimizations in a prototype tool named ATC4VPA.[5] The code is written mainly in Python, a small part being written in C for the low level operations on arrays of bit-vectors (used to represent relations). The experiments were run on a PC equipped with a Intel i7 2.8GHz processor, 6 GB of RAM and running Linux Ubuntu 3.2.

The experimental tests are performed on randomly generated automata, and compared with the results obtained with known prototype tools:

- VPAchecker, a package for deciding universality and inclusion of VPAs, by using either the classical algorithms based on the determinization of VPAs, or optimized on-the-fly algorithms [12],
- FADecider, a package for deciding universality and inclusion of VPAs (over finite and infinite words) using Ramsey-based methods [13],
- OpenNWA, a nested-word automaton library that provides the standard boolean operations [18]. Nested-word automata are another formalism which can be seen as an alternative encoding of VPAs [15].

During the random generation of VPAs, some parameters are fixed: there is one initial state ($|Q_i| = 1$), all states are final ($Q_f = Q$), the alphabets have size 3 ($|\Sigma_c| = |\Sigma_r| = |\Sigma_i| = 3$)[6]. Other parameters vary, like the size $|Q|$, the transition density, i.e. the number of outgoing transitions per state and per alphabet[7], and the number of generated VPAs for a fixed size (sample size).

In Figures 2, 4-6, we compare ATC4VPA with OpenNWA, VPAchecker and FADecider for increasing automata sizes, with a fixed transition density, and samples of 100 random automata for each size. In each figure, the first graph indicates the average time for false (resp. true) instances (without taking into account the timeouts), whereas the second graph indicates the number of false (resp. true) instances that did not reach a timeout fixed to 60 seconds (the number of timeouts is thus the sample size minus these two numbers). By true instances, we mean universal VPAs, and VPAs for which inclusion holds.

Classical automata techniques (typically complementation) do not scale for universality and inclusion tests. This is the case for instance with OpenNWA,

[5] AnTiChains for Visibly Pushdown Automata.

[6] $|\Sigma_i| = 0$ when the tested tool offers this possibility.

[7] For instance, a density transition of 5 means 5 outgoing transitions labeled by symbols of Σ_c (Σ_r, Σ_i resp.) for each state.

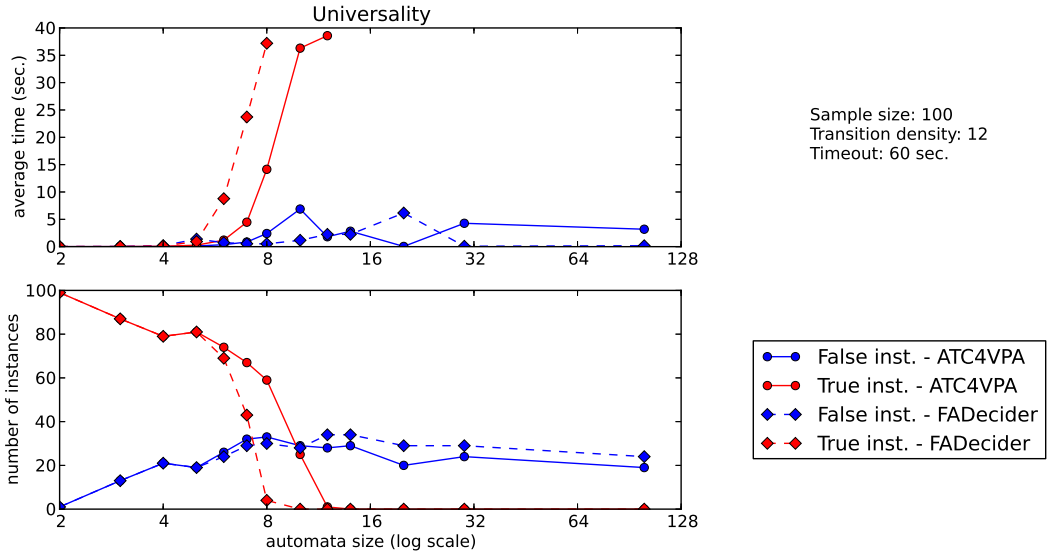


Fig. 2: Universality test for ATC4VPA and FADecider.

which quickly faces timeouts when the automata sizes increase (see Appendix D, Figure 4). Let us now focus on optimized tools.

On all instances, we observe that the *memory footprint* of our tool is much lower than for FADecider (as reported in Table 1 of Appendix D). Indeed, antichains reduce the number of relations that have to be computed (and thus stored and processed). This is also highlighted in Table 2 of Appendix D that compares the size of $\mathscr{R}^*$ and $\mathscr{R}^*_{min}$ in Algorithms 1 and 3 respectively.

On *true* instances, antichains show their full power, as the state space to be explored is usually huge. For these instances, ATC4VPA is indeed faster than both FADecider and VPAchecker. It also answers to more instances than the other tools (less timeouts), the only exception being the universality test in VPAchecker, where VPAchecker encounters slightly less timeouts (Appendix D, Figure 6). Note however that it is difficult to make a fair comparison since VPAchecker outputs a wrong answer for a few true instances.

On *false* instances, our prototype ATC4VPA is a bit slower and faces a bit more timeouts, but with the same asymptotical behavior. This is due to the fact that the data structure for antichains is more involved and our prototype did not optimize all its details. For instance, ATC4VPA sometimes proposes a witness of large size, whereas much smaller witnesses exist. The way we construct witnesses is related to the ordering in which the relations are treated in the algorithm (see Section 3.4). The efficiency of ATC4VPA could be improved by studying different such orderings and their effect on the size of witnesses.

Acknowledgements. The second author is supported by a grant from FRIA. We want to thank Thomas Brihaye for numerous useful discussions. We also thank the authors of VPAchecker, FADecider and OpenNWA for helping us performing experiments with their tools.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC. (2004) 202–211
2. L C.: Propositional dynamic logic with recursive programs. The Journal of Logic and Algebraic Programming **73**(1-2) (2007) 51–69
3. Kumar, V., Madhusudan, P., Viswanathan, M.: Visibly pushdown automata for streaming XML. In: WWW. (2007) 1053–1062
4. Francis, N., David, C., Libkin, L.: A Direct Translation from XPath to Nondeterministic Automata. In: 5th Alberto Mendelzon International Workshop. (2011)
5. De Wulf, M., Doyen, L., Henzinger, T., Raskin, J.: Antichains: A new algorithm for checking universality of finite automata. In: CAV. Volume 4144 of LNCS. (2006) 17–30
6. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: CIAA. Volume 5148 of LNCS. (2008) 57–67
7. Srba, J.: Visibly pushdown automata: From language equivalence to simulation and bisimulation. In: CSL. Volume 4207 of LNCS. (2006) 89–103
8. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: TACAS. Volume 6015 of LNCS. (2010) 158–174
9. Doyen, L., Raskin, J.F.: Antichain algorithms for finite automata. In: TACAS. Volume 6015 of LNCS. (2010) 2–22
10. Nguyen, T.V.: A tighter bound for the determinization of visibly pushdown automata. In: INFINITY. Volume 10 of EPTCS. (2009) 62–76
11. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: CAV. Volume 1855 of LNCS. (2000) 232–247
12. Nguyen, T.V., Ohsaki, H.: On model checking for visibly pushdown automata. In: LATA. Volume 7183 of LNCS. (2012) 408–419
13. Friedmann, O., Klaedtke, F., Lange, M.: Ramsey goes visibly pushdown. Technical report (2012)
14. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. (2007)
15. Alur, R., Madhusudan, P.: Adding nesting structure to words. Journal of the ACM **56**(3) (2009) 1–43
16. Gauwin, O., Niehren, J., Roos, Y.: Streaming tree automata. Information Processing Letters **109**(1) (2008) 13–17
17. Brüggemann-Klein, A., Murata, M., Wood, D.: Regular tree and regular hedge languages over unranked alphabets: Version 1. Technical Report HKTUST-TCSC-2001-05, HKUST Theoretical Computer Science Center Research (2001)
18. Driscoll, E., Thakur, A.V., Reps, T.W.: OpenNWA: A nested-word automaton library. In: CAV. Volume 7358 of LNCS. (2012) 665–671
19. LETHAL, a tree and hedge automata library http://lethal.sf.net/.
20. Segoufin, L., Vianu, V.: Validating streaming XML documents. In: PODS. (2002) 53–64

A Additional proofs

A.1 Proof of Lemma 1

Let $(q, q') \in Acc(t)$. Then there exists a run $(q, \epsilon) \xrightarrow{a:\gamma} (p_0, \gamma) \xrightarrow{[t_{|1}]} (p_1, \gamma) \xrightarrow{[t_{|2}]} \cdots \xrightarrow{[t_{|n-1}]} (p_{n-1}, \gamma) \xrightarrow{[t_{|n}]} (p_n, \gamma) \xrightarrow{\overline{a}:\gamma} (q', \epsilon)$. It follows that $(p_{i-1}, p_i) \in Acc(t_{|i})$ for $1 \leq i \leq n$, and thus $(p_0, p_n) \in r_1 \circ \cdots \circ r_n$. Therefore $(q, q') \in Post_a(r_1 \circ \cdots \circ r_n)$. The converse is proved similarly.

A.2 Proof of Lemma 2

We proceed by induction on i.

The basic case, $i = 1$, directly follows from $Post_a(id_Q) = Acc(t)$ with t being a leaf labelled by a.

Let $i > 1$ and suppose that the property holds for all $j, 1 \leq j < i$.
($\Rightarrow$) Let $r \in Post^i(\emptyset)$. If $r \in Post^{i-1}(\emptyset)$, then the property holds by induction hypothesis. Otherwise there exist $n \geq 0$, $r_1, \ldots, r_n \in Post^{i-1}(\emptyset)$, and $a \in \Sigma$, such that $r = Post_a(r_1 \circ \cdots \circ r_n)$. By induction hypothesis, $\forall k, 1 \leq k \leq n$, $\exists t_k \in T_\Sigma$ such that $height(t_k) < i$ and $r_k = Acc(t_k)$. Let t be the tree with a root labelled by a and the n subtrees $t_1, \ldots, t_n$. Then $height(t) \leq i$ and $r = Acc(t)$ by Lemma 1.
($\Leftarrow$) Let $t \in T_\Sigma$ with $height(t) \leq i$ and $r = Acc(t)$. If $height(t) < i$, then by induction hypothesis $r \in Post^{i-1}(\emptyset) \subseteq Post^i(\emptyset)$. Otherwise let a be the label of the root of t and $t_{|1}, \ldots, t_{|n}$ its n subtrees. Let $r_k = Acc(t_{|k})$, $1 \leq k \leq n$. As $height(t_{|k}) < i$, we have by induction hypothesis that $r_k \in Post^{i-1}(\emptyset)$. By Lemma 1, $r = Post_a(r_1 \circ \cdots \circ r_n)$, and thus $r \in Post^i(\emptyset)$.

A.3 Proof of Proposition 2

Let *Relations* be the set computed by Algorithm 2. Clearly, $Relations \subseteq (\mathscr{R}^* \cup \mathscr{R}')^*$. Assume by contradiction there exists r that belongs to $(\mathscr{R}^* \cup \mathscr{R}')^* \setminus Relations$. Then $r \notin \mathscr{R}^* \cup \mathscr{R}'$ and we can suppose wlog that $r = r'_2 \circ r'_1$ with $r'_1, r'_2 \in Relations$. Notice that at least one element among r'_1, r'_2 has been added to *ToProcess* during the execution of Algorithm 2, since otherwise $r'_1, r'_2 \in \mathscr{R}^*$ and then $r \in \mathscr{R}^*$. If r'_1 is the last one (among r'_1, r'_2) to be popped from *ToProcess*, then the relation $r'_2 \circ r'_1$ is added to *NewRelations*, which leads to a contradiction. The conclusion is similar if r'_2 is is the last one to be popped.

A.4 Proof of Lemma 3

The proof is done by induction on i such that $Post^*(\mathscr{R}) = \cup_{i \geq 0} Post^i(\mathscr{R})$, and on the next two observations:

- Given $a \in \Sigma$, and r, r' two relations over Q, if $r \subseteq r'$ then $Post_a(r) \subseteq Post_a(r')$.
- Let $r_1, \cdots, r_n, r'_1, \cdots, r'_n$ be relations over Q, if $r_i \subseteq r'_i, \forall 1 \leq i \leq n$, then $r_1 \circ \cdots \circ r_n \subseteq r'_1 \circ \cdots \circ r'_n$.

A.5 Proof of Proposition 3

The proof is based on Proposition 1.
($\Rightarrow$) As $Post^*_{min}(\emptyset) \subseteq Post^*(\emptyset)$, the proof is immediate.
($\Leftarrow$) Suppose that $\forall r \in Post^*_{min}(\emptyset), r \cap Q_i \times Q_f \neq \emptyset$. Let $r' \in Post^*(\emptyset)$. By Lemma 3, $\exists r \in Post^*_{min}(\emptyset) : r \subseteq r'$. It follows that $r' \cap Q_i \times Q_f \neq \emptyset$.

A.6 Proof of Proposition 4

The proof follows the same schema as for Proposition 1. Similarly to Lemma 2, we have $Post^i_{\subseteq}(\emptyset) = \{((q,q'),r) \mid (q,q') \in Acc_{\mathcal{A}}(t), r = Acc_{\mathcal{B}}(t)$ for some tree t with height $\leq i\}$.
($\Rightarrow$) Suppose that $L(\mathcal{A}) \subseteq L(\mathcal{B})$. Let $((q,q'),r) \in Post^*_{\subseteq}(\emptyset)$. Then there exists t such that $(q,q') \in Acc_{\mathcal{A}}(t)$ and $r = Acc_{\mathcal{B}}(t)$. If $(q,q') \in Q_{i,\mathcal{A}} \times Q_{f,\mathcal{A}}$, then $t \in L(\mathcal{A})$, and thus $t \in L(\mathcal{B})$. Therefore $r \cap Q_{i,\mathcal{B}} \times Q_{f,\mathcal{B}} \neq \emptyset$.
($\Leftarrow$) Suppose now that $\forall((q,q'),r) \in Post^*_{\subseteq}(\emptyset)$, $(q,q') \in Q_{i,\mathcal{A}} \times Q_{f,\mathcal{A}} \Rightarrow r \cap Q_{i,\mathcal{B}} \times Q_{f,\mathcal{B}} \neq \emptyset$. Let $t \in L(\mathcal{A})$. Then there exists $(q,q') \in Acc_{\mathcal{A}}(t)$ such that $(q,q') \in Q_{i,\mathcal{A}} \times Q_{f,\mathcal{A}}$. With $r = Acc_{\mathcal{B}}(t)$, it follows that $r \cap Q_{i,\mathcal{B}} \times Q_{f,\mathcal{B}} \neq \emptyset$, and thus $t \in L(\mathcal{B})$.

B From hedge automata to VPAs

We present here a translation from hedge automata to VPAs recognizing linearizations of trees. A translation from specialized DTDs is presented in [20]. It is quite similar to our translation, except that our translation is in polynomial time (we avoid determinization) and our presentation avoids specialization, which is already included in hedge automata.

A *hedge automaton* over Σ is a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where Q is a finite set of states, $Q_f \subseteq Q$ is the set of final states, and Δ is a finite set of transition rules of the following type: (a, L, q) where $a \in \Sigma$, $q \in Q$, and $L \subseteq Q^*$ is a regular language over Q, called a *horizontal language*. We assume wlog that, given a and q, there is a unique L such that $(a, L, q) \in \Delta$, and name $\mathcal{A}_{a,q}$ a finite state automaton recognizing L.

A *run* of $\mathcal{A}$ on a tree $t \in T_\Sigma$ is a tree $r \in T_Q$ with the same domain as t such that for each node $p \in nodes(r)$ and its n children $p1, p2, \ldots, pn$, if $a = t(p)$ and $q = r(p)$, then there is a rule $(a, L, q) \in \Delta$ with $r(p1)r(p2)\ldots r(pn) \in L$. In particular, applying the rule (a, L, q) at a leaf requires that the empty word ϵ belongs to L. Intuitively, a hedge automaton $\mathcal{A}$ operates in a bottom-up manner on a tree t: with a run r, it assigns a state to each leaf, and then to each internal node, according to the states assigned to its children. We use notation $t \underset{\mathcal{A}}{\hookrightarrow} q$ to indicate the existence of a run r on t that labels the root of t by the state q. The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of trees t on which $t \underset{\mathcal{A}}{\hookrightarrow} q$ for some $q \in Q_f$.

Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a hedge automaton. We assume, wlog, that all automata $\mathcal{A}_{a,q}$ have disjoint sets of states, and only one initial state per automaton,

that we write $i_{a,q}$. Given a state η, we write $m(\eta)$ for the pair $(a,q) \in \Sigma \times Q$ such that $\eta \in Q_{a,q}$.

Let $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \Gamma_\mathcal{B}, \{i\}, \{f\}, \Delta_\mathcal{B})$ be the VPA where:

- $Q_\mathcal{B} = \{i, f\} \cup \bigcup_{a\in\Sigma, q\in Q} Q_{a,q}$ with i, f being new states,
- $\Gamma_\mathcal{B} = \{f\} \cup \bigcup_{a\in\Sigma, q\in Q} Q_{a,q}$,
- $\Delta_\mathcal{B} = \quad \{i \xrightarrow{a:f} i_{a,q} \mid a \in \Sigma,\ q \in Q_f\}$
  $\cup\ \{\eta \xrightarrow{a:\eta'} i_{a,q} \mid a \in \Sigma,\ \eta \xrightarrow{q} \eta' \in \Delta_{m(\eta)}\}$
  $\cup\ \{\eta \xrightarrow{\overline{a}:\eta'} \eta' \mid a \in \Sigma,\ \eta' \in \Gamma_\mathcal{B},\ \text{and } \eta \in F_{m(\eta)}\}$

The idea here is that the VPA runs one automaton $\mathcal{A}_{a,q}$ at each depth between siblings. The choice of the transition $\eta \xrightarrow{q} \eta'$ of $\mathcal{A}_{a,q}$ to apply is done when opening a node (only η is known). The target state η' is put on the stack. When closing a node, we just pop the state η' from the stack and take it as new control state.

Proposition 5. *$L(\mathcal{A}) = L(\mathcal{B})$.*

Proof. Let us prove the following property by induction on the structure of trees:

$$\left(\eta \xrightarrow{q} \eta' \in \Delta_{m(\eta)} \text{ and } t \underset{\mathcal{A}}{\hookrightarrow} q\right) \text{ iff } (\eta, \epsilon) \xrightarrow{[t]} (\eta', \epsilon) \text{ is a run of } \mathcal{B}$$

(Basic case) If t is a leaf, then $t = a$ for some $a \in \Sigma$, and we have:

$\eta \xrightarrow{q} \eta' \in \Delta_{m(\eta)}$ and $t \underset{\mathcal{A}}{\hookrightarrow} q$ iff $\eta \xrightarrow{q} \eta' \in \Delta_{m(\eta)}$ and $\epsilon \in L(\mathcal{A}_{a,q})$

iff $\eta \xrightarrow{q} \eta' \in \Delta_{m(\eta)}$ and $i_{a,q} \in F_{a,q}$

iff $\eta \xrightarrow{a:\eta'} i_{a,q} \in \Delta_\mathcal{B}$ and $i_{a,q} \xrightarrow{\overline{a}:\eta'} \eta' \in \Delta_\mathcal{B}$

iff $(\eta, \epsilon) \xrightarrow{[t]} (\eta', \epsilon)$ is a run of $\mathcal{B}$

(Induction step) Assume t is not a leaf, has a root labeled by a, has n children, and thus n subtrees $t_{|1}, \ldots, t_{|n}$, for which the property holds. We have:

$t \underset{\mathcal{A}}{\hookrightarrow} q$ iff $\exists q_1, \ldots, q_n \in Q,\ t_{|i} \underset{\mathcal{A}}{\hookrightarrow} q_i$ for all $1 \le i \le n$, and $q_1 \cdots q_n \in L(\mathcal{A}_{a,q})$

iff $\exists q_1, \ldots, q_n \in Q,\ \exists \eta_0, \ldots, \eta_n \in Q_{a,q}$ with $\eta_0 = i_{a,q},\ \eta_n \in F_{a,q}$,
and $\eta_{i-1} \xrightarrow{q_i} \eta_i \in \Delta_{a,q}$ and $t_{|i} \underset{\mathcal{A}}{\hookrightarrow} q_i$ for all $1 \le i \le n$

iff $\exists \eta_0, \ldots, \eta_n \in Q_{a,q}$ with $\eta_0 = i_{a,q},\ \eta_n \in F_{a,q}$,
and $(\eta_{i-1}, \epsilon) \xrightarrow{[t_{|i}]} (\eta_i, \epsilon)$ is a run of $\mathcal{B}$ for all $1 \le i \le n$
(by induction hypothesis)

iff $\exists \eta_f \in F_{a,q}$, and $(i_{a,q}, \epsilon) \xrightarrow{[t_{|1}]\cdots[t_{|n}]} (\eta_f, \epsilon)$ is a run of $\mathcal{B}$

so we get:

$\eta \xrightarrow{q} \eta' \in \Delta_{m(\eta)}$ and $t \underset{\mathcal{A}}{\hookrightarrow} q$

iff $\eta \xrightarrow{q} \eta' \in \Delta_{m(\eta)}$ and $\exists \eta_f \in F_{a,q}$, and $(i_{a,q}, \epsilon) \xrightarrow{[t_{|1}]\cdots[t_{|n}]} (\eta_f, \epsilon)$ is a run of $\mathcal{B}$

iff $\eta \xrightarrow{a:\eta'} i_{a,q} \in \Delta_\mathcal{B}$ and $\exists \eta_f \in F_{a,q},\ \eta_f \xrightarrow{\overline{a}:\eta'} \eta' \in \Delta_\mathcal{B}$
and $(i_{a,q}, \eta') \xrightarrow{[t_{|1}]\cdots[t_{|n}]} (\eta_f, \eta')$ is a run of $\mathcal{B}$

iff $(\eta, \epsilon) \xrightarrow{[t]} (\eta', \epsilon)$ is a run of $\mathcal{B}$

Now that we have proved the property, let us show that $L(\mathcal{A}) = L(\mathcal{B})$. Consider $t \in T_\Sigma$ with a root labeled by a, and n children (with $n \geq 0$). We have: $t \in L(\mathcal{A})$ iff $\exists q \in Q_f,\ t \underset{\mathcal{A}}{\hookrightarrow} q$. Using the same sequence of equivalences as above, replacing the induction hypothesis by the property we proved, we get: $t \in L(\mathcal{A})$ iff $\exists q \in Q_f,\ \exists \eta_f \in F_{a,q}, (i_{a,q}, \epsilon) \xrightarrow{[t_{|1}]\cdots[t_{|n}]} (\eta_f, \epsilon)$ is a run of $\mathcal{B}$. This is equivalent to $t \in L(\mathcal{B})$ because rules $i \xrightarrow{a:f} i_{a,q}$ and $\eta_f \xrightarrow{\overline{a}:f} f$ are both in $\Delta_\mathcal{B}$, and $(i_{a,q}, f) \xrightarrow{[t_{|1}]\cdots[t_{|n}]} (\eta_f, f)$ is a run of $\mathcal{B}$.

C Optimized algorithm

Algorithm 4 uses antichains and includes the optimizations described in Section 3.4. *ToProcess* collects the newly constructed relations that still have to be processed. It is ordered such that the most promising relation is first popped. For each popped relation r, we apply the *Post* operator, and we compute the composition with all the relations of $\mathscr{R}^*$. During these computations, we directly test whether any new relation is a witness of non universality (see Algorithms 6 and 7).

Variables $\mathscr{R}^*$, *ToProcess* and $\mathscr{R}_{new}$, are antichains. The antichain $\mathscr{R}^*$ is the largest one. It is implemented with a hash table such that all relations with the same domain and codomain are in the same list. Function UNION computes the $\subseteq$-minimal elements of the union of two antichains.

Algorithm 4 Checking universality

```
function UNIVERSALITY(A)
    on exception NotUniversal return False    // Not Universal
    R* ← {id_Q}
    ToProcess ← {id_Q}
    while ToProcess ≠ ∅ do
        r ← POP(ToProcess)
        R_new ← POST(r, R*)
        ToProcess ← UNION(ToProcess, R_new)
        R_new ← COMPOSITION(r, R*)
        ToProcess ← UNION(ToProcess, R_new)
    end while
    return True    // Universal
end function
```

D Further experiments

D.1 Instances

Parameters - The random generation of VPAs and the tests depend on some parameters (some of which being fixed):

Algorithm 5 Checking for a witness of non universality

```
function TestWitness(r, R, R_new)
    if r ∉ R then
        if r ∩ Q_i×Q_f = ∅ then
            raise exception NotUniversal
        else
            R_new ← Union(R_new, {r})
        end if
    end if
    return R_new
end function
```

Algorithm 6 Computing *Post*

```
function Post(r, R)
    R_new ← ∅
    for a ∈ Σ do
        r_new ← Post_a(r)
        TestWitness(r_new, R, R_new)
    end for
    return R_new
end function
```

Algorithm 7 Computing the composition of two relations

```
function Composition(r, R)
    R ← R ∪ {r}
    R_new ← ∅
    for r' ∈ R do
        r_new ← r ∘ r'
        TestWitness(r_new, R, R_new)
        r_new ← r' ∘ r
        TestWitness(r_new, R, R_new)
    end for
    return R_new
end function
```

- size $|Q|$
- size $|Q_i|$ (always fixed to 1)
- density of final states $\frac{|Q_f|}{|Q|}$
- size of the alphabets Σ_c, Σ_r, Σ_i (all always fixed to 3)[8]
- transition density, i.e. the number of outgoing transitions per state and per alphabet
- number of generated VPAs for a fixed size (sample size)
- timeout

In Figure 3, we consider the universality test with our prototype ATC4VPA, for VPAs used as tree acceptors ($\Sigma_i = \emptyset$). The transition density and final state density are variable, whereas the size of Q is fixed to 10, the sample size equals 50, and the timeout is fixed to 60 seconds. The transition density varies from 1 to 20 in the following sense: a density transition of d means d outgoing transitions labeled by symbols of Σ_c (Σ_r resp.) for each state. The figure indicates the number of true and false instances that did not reach the timeout, it also indicates the number of instances that reach the timeout before being completely checked. When the two densities are high (resp. low), true (resp. false) instances are more probable.

Automata families - We have compared ATC4VPA with OpenNWA, VPAchecker and FADecider. The kind of tested automata changes a little depending on the compared tool. For VPAchecker, the automata are general VPAs and they are tested with our algorithm extended to pending calls and returns, as discussed in Section 4.2. As OpenNWA works with weak nested-word automata, the tests are performed with this family of automata for OpenNWA and with their translation to VPAs for ATC4VPA.[9] As FADecider allows to use VPAs as hedge acceptors, we limit the tests to this family and perform them with the optimized Algorithm 4.

D.2 Comparison with other tools

The experimental tests are performed with ATC4VPA for the universality and inclusion tests, and compared with the results obtained with OpenNWA, VPAchecker and FADecider. In Figures 2, 4-6, the automata size varies whereas the following parameters are fixed : $|Q_f| = |Q|$, transition density, sample size (100), and timeout (60 sec.). In each figure, the first graph indicates the average time for false (resp. true) instances (without taking into account the timeouts), whereas the second graph indicates, inside each sample, the number of false (resp. true) instances that did not reach the timeout (the number of timeouts is thus the sample size minus these two numbers).

[8] $|\Sigma_i| = 0$ when the tested tool offers this possibility.
[9] This translation needs to store the state used when opening the node in the stack.

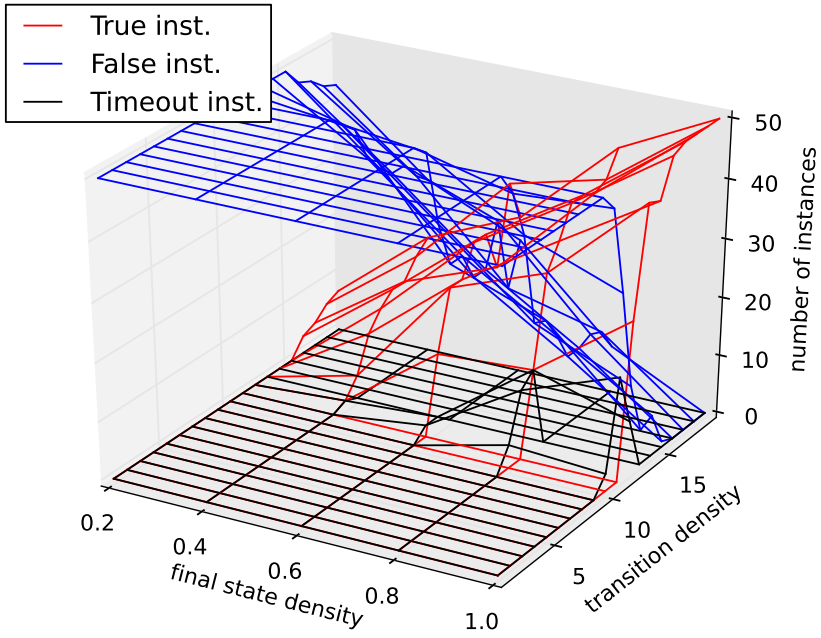


Fig. 3: Universality test for ATC4VPA with variable density of final states and transitions.

D.3 Antichains and memory

In our approach, antichains are used to reduce as much as possible the state space to be explored (the set of accessibility relations). On true instances, they show their full power, as this state space is usually huge (see Figures 2 and 5). The benefit of antichains is further highlighted by data of Tables 1 and 2.

The first table shows that the memory footprint of ATC4VPA is much lower than for FADecider: it stays arround 15 MB for ATC4VPA, but varies between 67 to 122 MB for FADecider.

The second table presents a comparison between the size of $\mathscr{R}^*$ and $\mathscr{R}^*_{min}$ in Algorithms 1 and 3 respectively. While $|\mathscr{R}^*|$ raises up to almost 6200 (the bound reached at timeout), $|\mathscr{R}^*_{min}|$ is at most 210. The small size of $\mathscr{R}^*_{min}$ follows from the use of antichains.

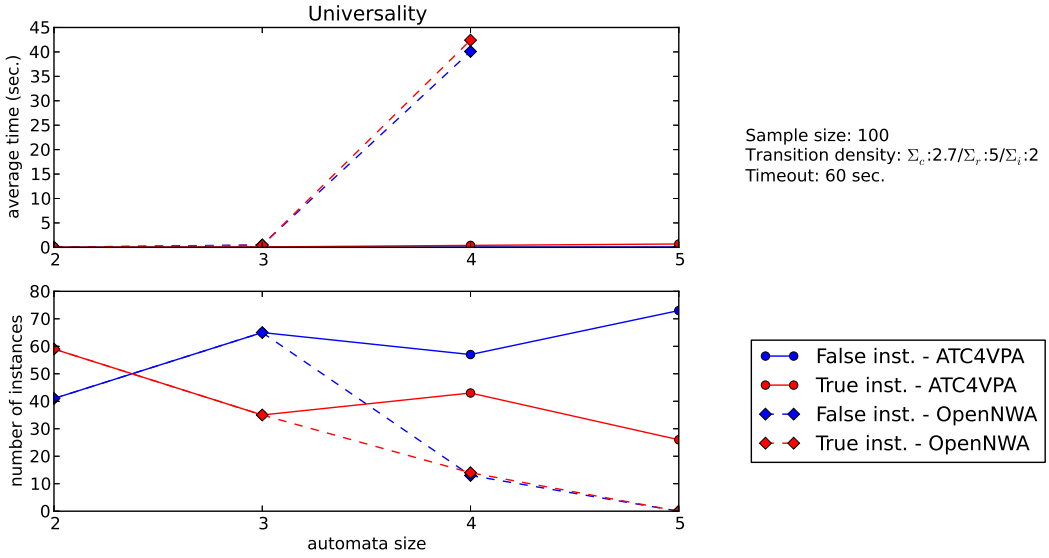


Fig. 4: Universality test for ATC4VPA and OpenNWA.

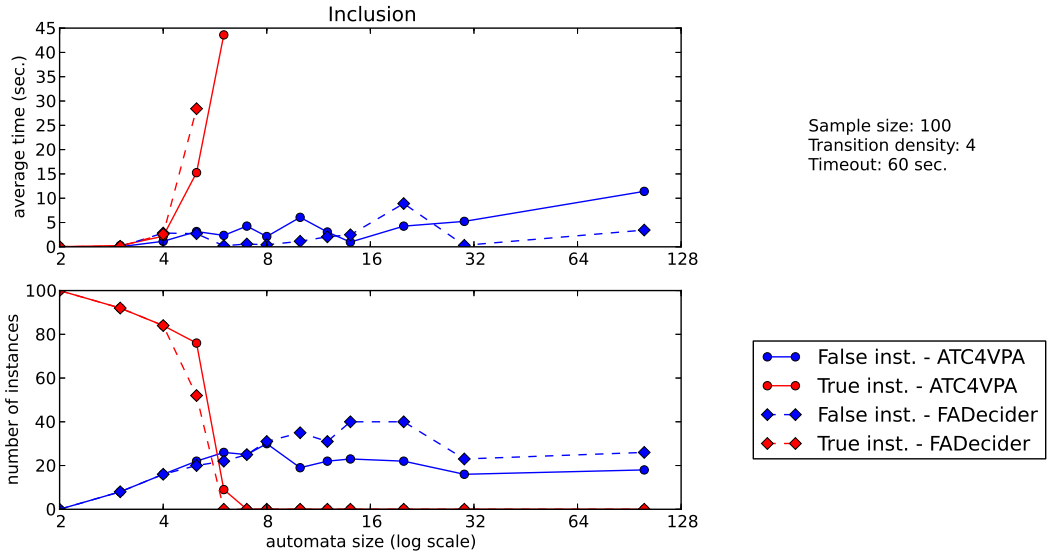


Fig. 5: Inclusion test for ATC4VPA and FADecider.

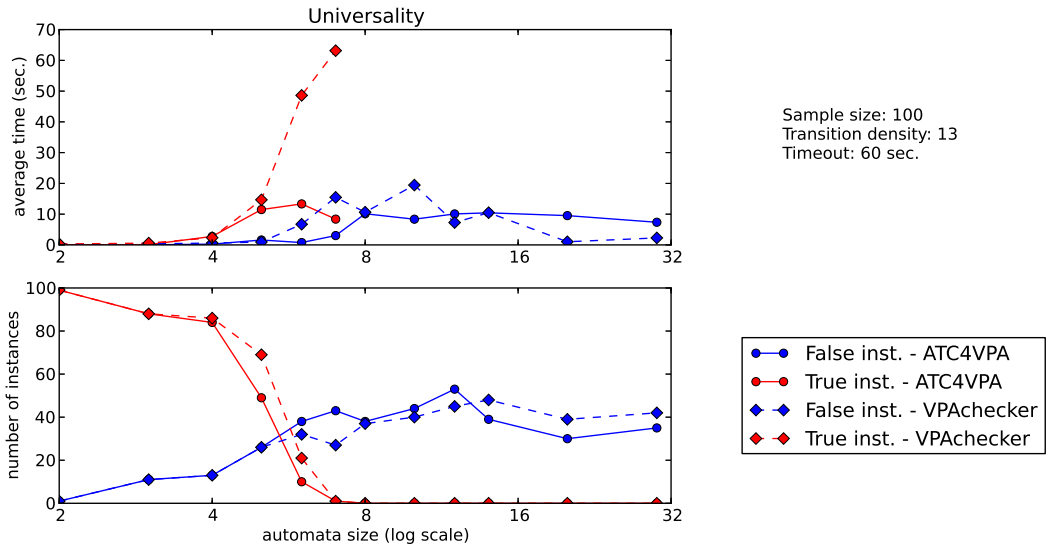


Fig. 6: Universality test for ATC4VPA and VPAchecker.

Table 1: Maximal memory consumption comparison between FADecider and ATC4VPA for 10 universal VPAs with 10 states and a transition density of 12. The timeout is fixed at 180 seconds.

♯	ATC4VPA: time (s)	ATC4VPA: space (kB)	FADecider: time (s)	FADecider: space (kB)
1	70.16	14744	Timeout	67112
2	59.99	14480	Timeout	94456
3	67.47	14744	Timeout	119384
4	69.26	15272	Timeout	72128
5	55.74	14216	Timeout	122440
6	56.56	15008	Timeout	86592
7	51.31	14216	Timeout	92460
8	61.47	14748	Timeout	76092
9	54.08	13952	Timeout	82424
10	63.52	14684	Timeout	75036

Table 2: Adding antichains: comparison between $|\mathscr{R}^*|$ and $|\mathscr{R}^*_{min}|$, on VPAs with 10 states and a transition density of 12. The timeout is fixed at 6 hours.

♯	Instance	time with antichains (s)	$\lvert\mathscr{R}^*_{min}\rvert$	time without antichains (s)	$\lvert\mathscr{R}^*\rvert$
1	False	5.66	92	Timeout	5932
2	False	8.85	104	20.27	145
3	False	11.38	41	139.35	496
4	True	11.37	122	Timeout	5593
5	True	12.90	131	Timeout	5708
6	True	29.88	210	Timeout	6149