Right-Universality of Visibly Pushdown Automata

Véronique Bruyère, Marc Ducobu, Olivier Gauwin

Right-Universality of Visibly Pushdown Automata

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

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

Abstract. Visibly pushdown automata (VPAs) express properties on structures with a nesting relation such as program traces with nested method calls. In the context of runtime verification, we are interested in the following problem: given u, the beginning of a program trace, and $\mathcal{A}$, a VPA expressing a property to be checked on this trace, can we ensure that any extension uv of u will be accepted by $\mathcal{A}$? We call this property right-universality w.r.t. u. We propose an online algorithm detecting at the earliest position of the trace, whether this trace is accepted by $\mathcal{A}$. The decision problem associated with right-universality is ExpTime-complete. Our algorithm uses antichains and other optimizations, in order to avoid the exponential blow-up in most cases. This is confirmed by promising experiments conducted on a prototype implementation.

1 Introduction

Program traces describe the control flows of program executions, including subroutine calls and returns. Some properties of program traces are specific to the nesting structure of calls and returns, as for instance reentrant locks [BYBC10]. These properties are not captured by regular languages, but can be expressed by visibly pushdown languages [AM09]. These latter languages rely on a partitioning of the input alphabet into internal letters, call letters, and return letters, which naturally fits to program traces. A language over such a partitioned alphabet is *visibly pushdown* if there exists a *visibly pushdown automaton* (VPA) recognizing it. VPAs are pushdown automata, where operations on the stack are driven by the letter type: call letters can only push, return letters can only pop, while internal letters do not have access to the stack.

Runtime verification amounts to check a property during a program execution. In this paper we are interested in finding the earliest point during the execution where the property can be asserted, for properties defined by VPAs. More formally, given a word u over a partitioned alphabet and a VPA $\mathcal{A}$, we say that $\mathcal{A}$ is *right-universal w.r.t.* u if, for every possible continuation v of u, the word uv is accepted by $\mathcal{A}$. Hence, if u is a prefix of the trace w and $\mathcal{A}$ is known to be right-universal w.r.t. u, then it can be asserted that w verifies the property described by $\mathcal{A}$, without reading w entirely. Our aim is to check right-universality *incrementally* after each incoming event of the trace w, as described

Algorithm 1 Checking right-universality incrementally

```
function INCREMENTAL-RIGHT-UNIVERSALITY(A)
    u ← ϵ
    while trace w is not completely read do
        a ← next letter of w
        u ← ua
        if A is right-universal w.r.t. u then
            return True
        end if
    end while
    return False
end function
```

in Algorithm 1. By incremental, we mean that some information is propagated from one event (reading a letter in w) to the next one (reading the next letter in w), avoiding repeated identical computations. Note that our algorithm will not store u, but enough information for asserting right-universality of $\mathcal{A}$ w.r.t. u.

For *safety properties*, right-universality allows to know at the earliest time point, whether the execution is sure, and thus whether controls can be stopped. This is typically the case for properties looking for a pattern (potentially complicated) that must occur during the execution. For *properties to be avoided*, this permits to stop the program before it enters an unsafe configuration, hence avoiding potential attacks [BJLW08].

These questions started to be addressed in the context of XML through *earliest query answering* of XPath expressions [BYFJ05,GNT09]. Indeed, XML documents also entail a nesting structure, similarly to program traces, and properties (or queries) over these documents can be expressed using VPAs. An algorithm asserting at the earliest time point, whether an XML document is accepted by a deterministic VPA has been proposed in [GNT09]. This algorithm runs in polynomial time at each incoming event. For non-deterministic VPAs, however, the decision problem associated with right-universality (i.e. given $\mathcal{A}$ and u, is $\mathcal{A}$ right-universal w.r.t. u?) is known to be ExpTime-complete [GNT09].

Algorithms that incrementally check right-universality of *non-deterministic* VPAs are challenging in several aspects. First, VPAs are usually an intermediate object, resulting from the translation of a property expressed in some logics, like XPath for XML documents. Such translations rely on non-determinism, as for instance when referring to any call inside a given procedure call (which corresponds to the descendant axis in XPath). Any VPA can be determinized, but the procedure yields VPAs of exponential size [AM09]. Second, right-universality w.r.t. u can be considered as a variant of universality, parameterized by u. Recent techniques have been proposed to check universality of non-deterministic VPAs efficiently [TO12,FKL13,BDG13]. Among them, *antichains* have been successfully applied to decision problems related to non-deterministic automata: universality and inclusion for finite word automata [DDHR06], and for non-deterministic bottom-up tree automata [BHH$^+$08]. Whether these techniques also apply to in-

crementally check right-universality is an interesting issue. Third, the algorithm proposed in the deterministic case [GNT09] does not directly generalize to the non-deterministic case.

Our contributions are the following. We propose an efficient incremental algorithm checking right-universality of a non-deterministic VPA $\mathcal{A}$. This algorithm relies on the progressive computation of *safe sets of configurations*, the configuration of a VPA being a state together with a stack content. It avoids to enterily determinize the given VPA. This algorithm uses antichains in order to get a compact representation of safe sets of configurations. We also propose efficient operators and data structures for saving useless computation. We report on some experiments, performed on randomly generated VPAs, and also on VPAs resulting from a translation from XPath expressions. These results exhibit the benefits of our optimizations.

Verification of traces with a nesting structure has already been addressed, through different aspects. Let us mention some of them. In [AEM04], the logic CaRet over words with a nesting structure is introduced, and a model checking algorithm is proposed. An extension is presented in [RCB08], with a corresponding monitor synthesis algorithm. VPAs are also sometimes used as an intermediate model to express properties on program traces [CA07,FJJ$^+$12].

The paper is structured as follows. In Section 2 we introduce VPAs and right-universality. Safe sets of configurations are defined in Section 3. Our antichain-based algorithm is described in Section 4, and further optimizations in Section 5. Experiments are reported in Section 6. Proofs are provided in Appendix.

2 Visibly Pushdown Automata and Right-Universality

Visibly pushdown automata (VPAs) are pushdown automata working on a partitioned alphabet where only call symbols can push, return symbols can pop, and internal symbols can fire transitions without considering the stack [AM04,AM09].

In this paper, we consider trees, instead of words with a nesting structure (i.e. their linearization). Such a mapping is illustrated in Figure 1. Each call and its matching return are mapped to a node, and the calls and returns directly nested under this call correspond to the children of this node. These trees are unranked, as the number of children of a node is not determined by its label. We use VPAs as *unranked tree acceptors*, operating on their linearization [GNR08]. In particular these VPAs do not use internal symbols.

2.1 Unranked Trees

We here recall the standard definition of unranked trees, as provided for instance in [CDG$^+$07]. 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. Given two words $v, w \in \Sigma^*$ over Σ, v is a *prefix* (resp. *proper prefix*) of w if there exists a word $v' \in \Sigma^*$ (resp. $v' \in \Sigma^+$) such that $vv' = w$.

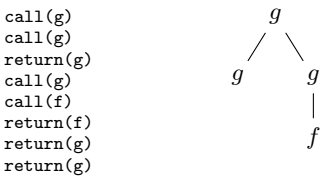


Fig. 1: Representation of a program trace as a tree t.

An *unranked tree* t over Σ is a tree such that its nodes are labeled by a letter of Σ and have an arbitrary number of children (the children are ordered from left to right). We call *a-node* a node with label $a \in \Sigma$. 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.

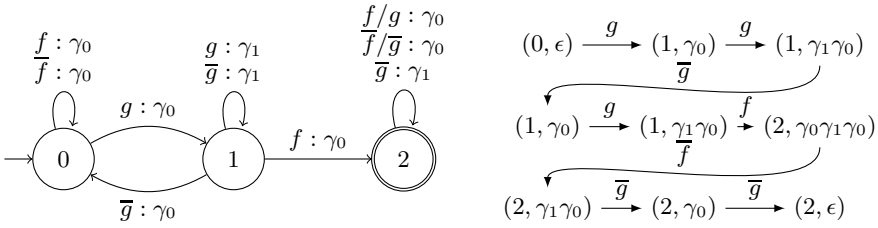
Trees can be described by well-balanced words which correspond to a depth-first traversal of the tree. An opening tag is used to notice the arrival on a node and a closing tag to notice the departure from a node. For each $a \in \Sigma$, let a itself represent the opening tag and $\overline{a}$ the related 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 is the label of the root and $t_1, \ldots, t_n$ are its n subtrees (from left to right). The linearization is extended to hedges as follows. Let $h = t_1 \cdots t_n$ be the sequence of trees t_i, $1 \leq i \leq n$. Then $[h] = [t_1] \cdots [t_n]$. We denote by $[T_\Sigma]$ (resp. $[H_\Sigma]$) the set of linearizations of all trees in T_Σ (resp. hedges in H_Σ). Consider for instance the tree t in Figure 1: its linearization is the word $[t] = gg\overline{g}gf\overline{f}\overline{g}\overline{g}$. Let $\mathit{Pref}(T_\Sigma)$ denote the set of all prefixes of $[T_\Sigma]$: $\mathit{Pref}(T_\Sigma) = \{u \in (\Sigma \cup \overline{\Sigma})^* \mid \exists v \in (\Sigma \cup \overline{\Sigma})^*,\ uv \in [T_\Sigma]\}$.

2.2 Visibly Pushdown Automata

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* $\gamma \in \Gamma$*.*

The left-hand side of a rule $q \xrightarrow{a:\gamma} p \in \Delta$ is (q, a) if $a \in \Sigma$, and (q, a, γ) if $a \in \overline{\Sigma}$. A VPA is *deterministic* if it has at most one initial state, and it does not have two distinct rules with the same left-hand side. We provide an example of deterministic VPA in Figure 2a.

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 $a \in \overline{\Sigma}$. We extend this notation to words $u = a_1 \cdots a_n$, by writing $(q_0, \sigma_0) \xrightarrow{u} (q_n, \sigma_n)$



(a) VPA $\mathcal{A}$ over alphabet $\Sigma = \{f, g\}$. (b) Run of $\mathcal{A}$ on $[t]$

Fig. 2: An automaton and one of its runs.

whenever there exist configurations (q_i, σ_i) such that $(q_{i-1}, \sigma_{i-1}) \xrightarrow{a_i} (q_i, \sigma_i)$ for all $1 \leq i \leq n$.

A *run* of a VPA $\mathcal{A}$ on a linearization $[t] = a_1 \cdots a_n$ of $t \in T_\Sigma$ is a sequence $(q_0, \sigma_0) \cdots (q_n, \sigma_n)$ of configurations such that (q_0, σ_0) is initial, and for every $1 \leq i \leq n$, $(q_{i-1}, \sigma_{i-1}) \xrightarrow{a_i} (q_i, \sigma_i)$. Such a run is *accepting* if (q_n, σ_n) is final. A tree t 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 $L(\mathcal{A})$.[3] For instance, given a tree t, the VPA of Figure 2a checks whether t has a g-node with an f-child. An accepting run on $[t]$ for the tree t of Figure 1, is depicted in Figure 2b.

2.3 Universality and Right-Universality

We conclude the section of preliminaries with the notions of universality[4] and right-universality, that we will study in the remainder of the paper.

Definition 2. *A VPA $\mathcal{A}$ over Σ is said* universal *if $\mathcal{A}$ accepts all trees $t \in T_\Sigma$, i.e. $L(\mathcal{A}) = T_\Sigma$. Let $u \in Pref(T_\Sigma) \setminus \{\epsilon\}$ be a non empty prefix of $[t_0]$ for some tree $t_0 \in T_\Sigma$. The VPA $\mathcal{A}$ is said* right-universal w.r.t. *u if for all trees $t \in T_\Sigma$, if u is a prefix of $[t]$, then t is accepted by $\mathcal{A}$.*

In other words, right-universality w.r.t. u allows to assert that any tree linearization beginning with u is accepted by the automaton. In this definition, notice that when $u = [t_0]$, then $\mathcal{A}$ is right-universal w.r.t. u iff t_0 is accepted by $\mathcal{A}$. Moreover, as a universal VPA is right-universal w.r.t. all non empty words $u \in Pref(T_\Sigma)$, we assume in the sequel that VPAs are not universal.

In this article, our objective is to propose an *incremental* algorithm for right-universality of a VPA $\mathcal{A}$, in the sense described in the introduction (see Algorithm 1). More precisely, the linearization $[t_0]$ of a given tree t_0 is read letter

[3] VPAs considered in this article are tree acceptors, with acceptance on empty stack.

[4] This notion of universality is different from the one proposed for usual VPAs, where a VPA is universal if it accepts all words of $(\Sigma \cup \overline{\Sigma})^*$ (and not only linearizations of trees).

by letter[5], and while $\mathcal{A}$ is not right-universal w.r.t. the current read prefix u of $[t_0]$, the next letter of $[t_0]$ is read. When processing a new letter, we try to reuse prior computations as much as possible. In the sequel t_0 always refers to this particular tree.

We recall that the right-universality problem is ExpTime-complete for VPAs, but in PTime for deterministic VPAs [GNT09]. A dual problem to right-universality w.r.t. u is to ask whether every word starting with u is *rejected* by a given VPA. This problem is in PTime, even when the VPA is not deterministic. Indeed, this amounts to check whether no configuration reached after reading u can reach a final configuration.

2.4 Towards an Algorithm

In this section we consider several approaches for addressing right-universality of VPAs, and justify our choice of considering safe sets of configurations. Readers not familiar with the related litterature can skip this section at first reading. As explained in the introduction, we discard the naive approach consisting in determinizing the VPA and then applying the algorithm in [GNT09], in order to avoid state-space explosion.

The algorithm given in [GNT09] for deterministic VPAs is a progressive computation of *safe states*, with the property that the VPA is right-universal w.r.t. u iff the state reached after reading u is safe. Note that safe states cannot be determined statically: a state can be safe at some position of the word, but not at another one. A first idea for the non-deterministic case is a subset construction, using sets of safe states instead of safe states, but this direct adaptation is not correct. Hence, *safe sets of configurations* will be the basis of our algorithm, instead of safe states. This point raises new questions, as the configuration space is infinite, unlike the state space. In fact we will see that at each event, safe sets of configurations have the same stack height as the depth of the word u leading to this event, and are thus of finite (but potentially huge) cardinality at each time point.

Recently, several authors have proposed efficient algorithms to check universality of VPAs [TO12,FKL13,BDG13]. The method that we give in [BDG13] computes in an incremental way the set $\mathcal{R}$ of accessibility relations for all hedges (i.e. $\{\mathsf{rel}_h \mid h \in H_\Sigma\}$ in the present paper, also called summaries in [AM04]). It uses *antichains* to get smaller objects to manipulate (in particular to limit $\mathcal{R}$ to a strict smaller subset) and to avoid an explicit determinization step. It can be easily adapted for checking right-universality of VPAs [BDG12, Section 3.4]. However we face the problem of computing the whole set $\mathcal{R}$ (instead of a strict subset): experiments indicate that this step is too much time consuming [BDG13]. For this reason we do not follow this approach, but use antichains over safe sets of configurations rather than accessibility relations. This approach is developed in the next section.

[5] $[t_0]$ is the trace w of Algorithm 1.

3 Safe Sets of Configurations

In this section, $[t_0]$ is a fixed tree whose linearization is read letter by letter, and u denotes the current read prefix. We introduce the new notion of safe set of configurations related to u. We show how safe sets of configurations can be defined from such sets associated to smaller prefixes u' of $[t_0]$. This property will be useful to derive an incremental algorithm for checking right-universality of a VPA.

3.1 A Notion of Safety for Right-Universality

Definition 3. *Let $\mathcal{A}$ be a VPA and $\mathscr{C} \subseteq Q \times \Gamma^*$ be a non empty set of configurations. Let $u \in Pref(T_\Sigma) \setminus \{\epsilon\}$ be a non empty prefix of t_0.*

- *$\mathscr{C}$* is safe for *u if for every v such that $uv \in [T_\Sigma]$, there exist $(q, \sigma) \in \mathscr{C}$ and $p \in Q_f$ such that $(q, \sigma) \xrightarrow{v} (p, \epsilon)$ in $\mathcal{A}$.*
- *$\mathscr{C}$* is leaf-safe for *u if for every $v = \overline{a}v'$ with $\overline{a} \in \overline{\Sigma}$ such that $uv \in [T_\Sigma]$, there exist $(q, \sigma) \in \mathscr{C}$ and $p \in Q_f$ such that $(q, \sigma) \xrightarrow{v} (p, \epsilon)$ in $\mathcal{A}$.*[6]

Let $Safe(u) = \{\mathscr{C} \mid \mathscr{C}$ is safe for $u\}$ and $LSafe(u) = \{\mathscr{C} \mid \mathscr{C}$ is leaf-safe for $u\}$.

Intuitively, as stated in Theorem 1 below, if $\mathscr{C}$ is the set of configurations reached in $\mathcal{A}$ after reading u, then $\mathcal{A}$ is right-universal w.r.t. u iff $\mathscr{C}$ is safe for u. Indeed, for every possible v, one can find in $\mathscr{C}$ at least one configuration leading to an accepting configuration after reading v. Before stating this theorem, let us give the next definition. Let $Reach(u)$ denote the set of configurations (q, σ) such that $(q_0, \sigma_0) \xrightarrow{u} (q, \sigma)$ for some initial configuration (q_0, σ_0) of $\mathcal{A}$.

Theorem 1. *$\mathcal{A}$ is right-universal w.r.t. u iff $Reach(u) \in Safe(u)$.*

Let us illustrate this on the VPA $\mathcal{A}$ depicted in Figure 2a. Recall that this VPA checks that the tree has a g-node with an f-child. After opening a g-root, the set of safe configurations is $Safe(g) = \{\{(q_2, \gamma_1)\}, \{(q_2, \gamma_0)\}\}$, which does not contain the set of reached configurations, as $Reach(g) = \{(q_1, \gamma_0)\}$. Indeed, $\mathcal{A}$ is not right-universal w.r.t. g, as no f-node has been encountered yet under a g-node. For the word gf, as expected, the situation differs: $Safe(gf) = \{\{(q_2, \gamma_1\gamma_0)\}, \{(q_2, \gamma_0\gamma_0)\}\}$, and $Reach(gf) = \{(q_2, \gamma_0\gamma_0)\}$ is a safe set of configurations. By Theorem 1, $\mathcal{A}$ is right-universal w.r.t. gf.

Hence, an algorithm computing both $Reach(u)$ and $Safe(u)$ can decide right-universality w.r.t. u. The set $Reach(u)$ is easy to compute, just by firing transitions of the VPA. In Sections 3.2-3.4, we detail how the set $Safe(u)$ can be defined from the set $Safe(u')$ with u' a proper prefix of u. In this way, while reading the linearization $[t_0]$ of a given tree t_0, the set $Safe(u)$ with $u \neq \epsilon$ prefix of $[t_0]$, can be incrementally defined. In Section 4, we turn this approach into an algorithm.

[6] The name "leaf-safe" comes from the fact that we only consider suffixes starting with a $\overline{\Sigma}$ symbol, and thus continuations of the tree that do not add children to the current node.

3.2 Starting Point

We begin with $Safe(a)$ with $a \in \Sigma$, for which we recall the definition.

$$Safe(a) = \{\mathscr{C} \subseteq Q \times \Gamma^* \mid \forall h \in H_\Sigma, \exists q_f \in Q_f, \exists (q,\sigma) \in \mathscr{C} : (q,\sigma) \xrightarrow{h\overline{a}} (q_f, \epsilon)\} \quad (1)$$

3.3 Reading a Letter $\overline{a} \in \overline{\Sigma}$

Suppose that we are reading an $\overline{a} \in \overline{\Sigma}$ and we have to compute $Safe(u\overline{a})$. Two cases occur : either $u\overline{a} \neq [t_0]$, or $u\overline{a} = [t_0]$. Let us study these two cases and show how to manage them from an algorithmic point of view.

If $u\overline{a} \neq [t_0]$, we can retrieve safe sets configurations from prior computed such sets: $Safe(u\overline{a}) = Safe(u')$ where $u' \neq \epsilon$ is the unique prefix of u such that $u = u'a[h]$, with $h \in H_\Sigma$. Indeed as shown by Lemma 1 below, we have $Safe(u'a[h]\overline{a}) = Safe(u')$. Hence, from an algorithmic point of view, we just have to use a stack $\mathcal{S}$ to store these safe sets of configurations. When opening a, we put $Safe(u')$ on the stack, and when closing $\overline{a}$, we pop it. As h is a hedge, the stack before reading $\overline{a}$ is exactly the stack after reading a.

Lemma 1. *If $h \in H_\Sigma$, then $Safe(u[h]) = Safe(u)$ and $LSafe(u[h]) = LSafe(u)$.*

If $u\overline{a} = [t_0]$, the previous argument is no longer correct since $u' = \epsilon$ and $Safe(u')$ is not defined for the empty word. Nevertheless, by definition $Safe(u\overline{a}) = \{\mathscr{C} \subseteq Q \times \Gamma^* \mid \exists (q,\epsilon) \in \mathscr{C} \text{ with } q \in Q_f\}$. Therefore we can again use stack $\mathcal{S}$ as described before if it is initialized with the set

$$Init = \{\mathscr{C} \subseteq Q \times \Gamma^* \mid \exists (q,\epsilon) \in \mathscr{C} \text{ with } q \in Q_f\}. \quad (2)$$

3.4 Reading a Letter $a \in \Sigma$

Let us now consider the much more involved case of sets $Safe(ua)$ with $a \in \Sigma$. When reading an $a \in \Sigma$, two successive steps are performed, with leaf-safe sets of configurations as intermediate object:

$$Safe(u) \xrightarrow{\text{Step 1}} LSafe(ua) \xrightarrow{\text{Step 2}} Safe(ua)$$

For this purpose, we introduce the notion of *predecessor*. Let $\mathscr{C}, \mathscr{C}'$ be two sets of configurations, let $\overline{a} \in \overline{\Sigma}$ and $h \in H_\Sigma$.

- *$\mathscr{C}$ is an $\overline{a}$-predecessor of $\mathscr{C}'$* if $\forall (q',\sigma') \in \mathscr{C}',\ \exists (q,\sigma) \in \mathscr{C},\ (q,\sigma) \xrightarrow{\overline{a}} (q',\sigma')$.
- *$\mathscr{C}$ is a h-predecessor of $\mathscr{C}'$* if $\forall (q',\sigma') \in \mathscr{C}',\ \exists (q,\sigma) \in \mathscr{C},\ (q,\sigma) \xrightarrow{[h]} (q',\sigma')$.

Let $Pred_{\overline{a}}(\mathscr{C}') = \{\mathscr{C} \mid \mathscr{C} \text{ is an } \overline{a}\text{-predecessor of } \mathscr{C}'\}$ and $Pred_h(\mathscr{C}') = \{\mathscr{C} \mid \mathscr{C} \text{ is a } h\text{-predecessor of } \mathscr{C}'\}$.

The next proposition can be used to perform Step 1 and Step 2. It states that safe sets of configurations are only among predecessors of prior safe sets of configurations.

Proposition 1. *Let $ua \in Pref(T_\Sigma)$.*

$$\mathscr{C} \in LSafe(ua) \iff \exists \mathscr{C}' \in Safe(u),\ \mathscr{C} \in Pred_{\overline{a}}(\mathscr{C}') \tag{3}$$

$$\mathscr{C} \in Safe(ua) \iff \forall h \in H_\Sigma,\ \exists \mathscr{C}' \in LSafe(ua),\ \mathscr{C} \in Pred_{\overline{h}}(\mathscr{C}') \tag{4}$$

However, to get an algorithm, we face the problem that the number of hedges to consider in Equivalence (4) is infinite. We use relations to overcome this. Also the size of $Safe(u)$ may be huge and not all configurations of $Safe(u)$ are crucial for checking right-universality w.r.t. u. We use antichains to get a compact representation of $Safe(u)$. These two concepts are explained in the following section.

4 An Antichain-Based Algorithm for Right-Universality

In this section, we give an antichain-based algorithm for incrementally checking right-universality of a VPA, that uses the approach given in Sections 3.2-3.4.

Let us summarize this approach. Let $[t_0]$ be the linearization of a given tree t_0. We initialize a stack $\mathcal{S}$ with set *Init* defined in (2) and we start by computing $Safe(u)$ with u being the first letter of $[t_0]$ (see (1)). Suppose that $Safe(u)$ has been computed from the current read prefix u of $[t_0]$. Then, if the next letter read in $[t_0]$ is $a \in \Sigma$, we compute $Safe(ua)$ from $Safe(u)$ by Steps 1 and 2, and we put $Safe(u)$ on the stack $\mathcal{S}$. If the next letter read in $[t_0]$ is $\overline{a} \in \overline{\Sigma}$, then we pop the stack $\mathcal{S}$. The element that has been popped is the set $Safe(u') = Safe(u\overline{a})$ where u' is the unique prefix of u such that $u = u'a[h]$ (except if $u\overline{a} = t_0$ in which case the popped set is equal to $Init = Safe(u\overline{a})$), see Section 3.3. At each computation of $Safe(u)$, we check whether $Reach(u) \in Safe(u)$ (see Theorem 1).

4.1 Finite Number of Hedges

We begin by showing that only a finite number of hedges has to be considered in Equivalence (4). The reason is that a hedge h does not change the stack of a configuration during a run of a VPA, that is $(q, \sigma) \xrightarrow{[h]} (q', \sigma)$. So h can be considered as a function mapping each state q to the set of states obtained when traversing h from q. Formally, for every $h \in H_\Sigma$, rel_h is the function from Q to 2^Q such that $q' \in \mathsf{rel}_h(q)$ iff $(q, \sigma) \xrightarrow{[h]} (q', \sigma)$ for some $\sigma \in \Gamma^*$. The number of such functions is finite, and bounded by $|Q| \cdot 2^{|Q|}$. These functions naturally define an equivalence relation of finite index over H_Σ:

$$h \sim h' \iff \mathsf{rel}_h = \mathsf{rel}_{h'}.$$

Let us note H for a subset containing one hedge per $\sim$-class. We have $|H| \leq |Q| \cdot 2^{|Q|}$. The next lemma indicates that the computation of h-predecessors can be limited to $h \in H$.

Lemma 2. *For every $h \in H_\Sigma$, $\mathscr{C}$ is a h-predecessor of $\mathscr{C}'$ iff there exists $h' \in H$ with $h \sim h'$, such that $\mathscr{C}$ is a h'-predecessor of $\mathscr{C}'$.*

The set H can be computed by a saturation method based on the VPA rules, as described by Algorithm 3 in Appendix B.2.

4.2 Antichains

Let us now introduce antichains. Consider the set $2^{Q\times\Gamma^*}$ of all sets of configurations, with the $\subseteq$ operator. An *antichain* is a set of pairwise incomparable sets of configurations with respect to $\subseteq$. Given a set α of sets of configurations, we denote by $\lfloor\alpha\rfloor$ the $\subseteq$-*minimal* elements of α. The set α is $\subseteq$-*upward closed* if for all $\mathscr{C}\in\alpha$ and $\mathscr{C}\subseteq\mathscr{C}'$, we have $\mathscr{C}'\in\alpha$. From their definition, it is immediate that $Safe(u)$, $LSafe(u)$, $Pred_{\overline{a}}(\mathscr{C})$ and $Pred_h(\mathscr{C})$ are $\subseteq$-upward closed sets.

The idea is to compute the antichain $\lfloor Safe(u)\rfloor$ (resp. $\lfloor LSafe(u)\rfloor$) instead of the whole $\subseteq$-upward closed set $Safe(u)$ (resp. $LSafe(u)$). The next corollary of Theorem 1 indicates that such a limited computation is enough to check whether a VPA $\mathcal{A}$ is right-universal w.r.t. u.

Corollary 1. *$\mathcal{A}$ is right-universal w.r.t. u iff there exists $\mathscr{C}\in\lfloor Safe(u)\rfloor$ such that $\mathscr{C}\subseteq Reach(u)$.*

Moreover, we show in Appendix B.3 that the antichains $\lfloor Safe(u)\rfloor$ and $\lfloor LSafe(u)\rfloor$ are finite and only contain finite sets $\mathscr{C}$ of configurations such that $\mathscr{C}\subseteq Q\times\Gamma^k$ for some k. We now try to use these antichains at the starting point, and in Steps 1 and 2 of our approach.

4.3 Starting Point with Antichains

Let us explain how to compute $Safe(a)$. Clearly, by definition of H and using (1) (see Section 3.2), we can compute $\lfloor Safe(a)\rfloor$ as follows:

$$\lfloor Safe(a)\rfloor = \left\lfloor \left\{ \mathscr{C} \mid \forall h\in H, \exists q_f\in Q_f, \exists (q,\sigma)\in\mathscr{C} : (q,\sigma)\xrightarrow{h\overline{a}}(q_f,\epsilon)\right\}\right\rfloor . \quad (5)$$

4.4 Step 1 with Antichains: from $\lfloor Safe(u)\rfloor$ to $\lfloor LSafe(ua)\rfloor$

For the two steps, the goal is to adapt Proposition 1 so that it uses $\lfloor Safe(.)\rfloor$ instead of $Safe(.)$, and $\lfloor LSafe(.)\rfloor$ instead of $LSafe(.)$. We begin with Step 1. Implication ($\Rightarrow$) of Equivalence (3) can be directly adapted.

$$\mathscr{C}\in\lfloor LSafe(ua)\rfloor \implies \exists\mathscr{C}'\in\lfloor Safe(u)\rfloor,\ \mathscr{C}\in Pred_{\overline{a}}(\mathscr{C}'). \quad (6)$$

Implication (6) gives us a way to compute $\lfloor LSafe(ua)\rfloor$ from $\lfloor Safe(u)\rfloor$: it suffices to take all $\overline{a}$-predecessors of elements of $\lfloor Safe(u)\rfloor$ and then limit to those predecessors that are $\subseteq$-minimal. We can even only consider minimal $\overline{a}$-predecessors of $\lfloor Safe(u)\rfloor$ in the following sense: $\mathscr{C}$ is a *minimal $\overline{a}$-predecessor* of $\mathscr{C}'$ if for all $\mathscr{C}''$ $\overline{a}$-predecessor of $\mathscr{C}'$, $\mathscr{C}''\subseteq\mathscr{C}\implies\mathscr{C}''=\mathscr{C}$. We finally obtain:

$$\lfloor LSafe(ua)\rfloor = \lfloor\{\mathscr{C} \mid \mathscr{C} \text{ is a minimal } \overline{a}\text{-predecessor of } \mathscr{C}'\in\lfloor Safe(u)\rfloor\}\rfloor . \quad (7)$$

4.5 Step 2 with Antichains: from $\lfloor LSafe(ua) \rfloor$ to $\lfloor Safe(ua) \rfloor$

The second step for computing $\lfloor Safe(ua) \rfloor$ from $\lfloor Safe(u) \rfloor$ relies on the introduction of antichains in Equivalence (4). Implication ($\Rightarrow$) holds with antichains:

$$\mathscr{C} \in \lfloor Safe(ua) \rfloor \implies \forall h \in H_\Sigma, \exists \mathscr{C}' \in \lfloor LSafe(ua) \rfloor, \ \mathscr{C} \in Pred_{\overline{h}}(\mathscr{C}'). \tag{8}$$

Similarly to Implication (6), we can restrict h-predecessors to only consider minimal ones: $\mathscr{C}$ is a *minimal h-predecessor* of $\mathscr{C}'$ if for all $\mathscr{C}''$ h-predecessor of $\mathscr{C}'$, $\mathscr{C}'' \subseteq \mathscr{C} \implies \mathscr{C}'' = \mathscr{C}$. Moreover, by Lemma 2, we can limit the computations to hedges $h \in H$ where H is the finite set introduced in Section 4.1. Therefore, we obtain the next equality.

$$\lfloor Safe(ua) \rfloor = \left\lfloor \left\{ \mathscr{C} \mid \mathscr{C} = \bigcup_{h \in H} \mathscr{C}_h \text{ with } \mathscr{C}_h \text{ a minimal } h\text{-predecessor of } \mathscr{C}' \in \lfloor LSafe(ua) \rfloor \right\} \right\rfloor \tag{9}$$

4.6 Algorithm

We are now able to give our antichain-based algorithm (see Algorithm 2). It uses a stack $\mathcal{S}$ (initially empty) as recalled at the beginning of this section. The computation of $\lfloor H \rfloor$ is considered as a *preprocessing*, as its value only depends on $\mathcal{A}$ and not on $[t_0]$. The used results are mentioned inside the algorithm.

5 Improvements and Implementation

Section 4 resulted in a first algorithm for incrementally testing whether a VPA is right-universal. In this section, we show how this algorithm can be improved by limiting hedges to consider, and optimizing operators and predecessors to be computed. We also give the improved algorithm in a more detail way than in Algorithm 2, as well as the underlying data structures.

5.1 Minimal Hedges

A first improvement is obtained by further restricting hedges to consider. Indeed it suffices to consider *minimal hedges* wrt their function rel_h. Formally, let us write $h \leq h'$ whenever $\mathsf{rel}_h(q) \subseteq \mathsf{rel}_{h'}(q)$ for every $q \in Q$. We denote by $\lfloor H \rfloor$ the $\leq$-minimal elements of H. From the definition of h-predecessor, we have:

$$\mathscr{C} \ h\text{-predecessor of } \mathscr{C}' \text{ and } h \leq h' \implies \mathscr{C} \ h'\text{-predecessor of } \mathscr{C}' \tag{10}$$

This property can be used to replace $h \in H$ in (9) by $h \in \lfloor H \rfloor$:

$$\lfloor Safe(ua) \rfloor = \left\lfloor \left\{ \mathscr{C} \mid \mathscr{C} = \bigcup_{h \in \lfloor H \rfloor} \mathscr{C}_h \text{ with } \mathscr{C}_h \text{ a minimal } h\text{-predecessor of } \mathscr{C}' \in \lfloor LSafe(ua) \rfloor \right\} \right\rfloor \tag{11}$$

Algorithm 2 Checking right-universality incrementally with antichains

```
function ANTICHAIN-BASED INCREMENTAL-RIGHT-UNIVERSALITY(A, ⌊H⌋)
    PUSH(S, ⌊Init⌋)    % by (2), S being a stack
    u ← first letter of [t_0]
    R ← ⟨Compute Reach(u)⟩
    α ← ⟨Compute ⌊Safe(u)⌋⟩    % by (5)
    if ∃C ∈ α : C ⊆ R then
        return True    % Corollary 1
    end if
    while [t_0] is not completely read do
        a ← next letter of [t_0]
        u ← ua
        R ← ⟨Compute Reach(u) from R⟩
        if a ∈ Σ̄ then
            α ← POP(S)
        else
            PUSH(S, α)
            α ← ⟨Compute ⌊Safe(u)⌋ from α⟩    % by (7) and (9)
        end if
        if ∃C ∈ α : C ⊆ R then
            return True    % by Corollary 1
        end if
    end while
    return False
end function
```

and similarly for the starting point:

$$\lfloor Safe(a) \rfloor = \left\lfloor \left\{ \mathscr{C} \mid \forall h \in \lfloor H \rfloor, \exists q_f \in Q_f, \exists (q,\sigma) \in \mathscr{C} : (q,\sigma) \xrightarrow{h\overline{a}} (q_f, \epsilon) \right\} \right\rfloor \quad (12)$$

5.2 An Appropriate Operator

Equation (11) expresses that every set of configurations $\mathscr{C}$ in $\lfloor Safe(ua) \rfloor$ is the union of $\mathscr{C}_h$ with $h \in \lfloor H \rfloor$. We introduce a new operator to improve the readability and find new properties. Let S be a finite set, and $A, B \in 2^{2^S \setminus \{\emptyset\}}$. The set $A \sqcup B \in 2^{2^S}$ is defined by

$$A \sqcup B = \{a \cup b \mid a \in A \text{ and } b \in B\}.$$

Operator $\sqcup$ builds sets obtained by taking one set of each of its operands, and performing their union. It is obviously associative and commutative. Notice that the elements of A, B are supposed to be non empty sets. This is always the case in the algorithms using this operator.

When combined with operator $\lfloor . \rfloor$, operands of the $\sqcup$ operator can be splitted, so that $\sqcup$ is to be computed on smaller sets:

$$\lfloor A \sqcup B \rfloor = \lfloor (A \cap B) \cup (A \setminus B \ \sqcup \ B \setminus A) \rfloor \quad (13)$$

We experimentally observed that a good strategy to evaluate an expression $A_1 \sqcup \ldots \sqcup A_n$ is to process sets A_i with increasing cardinality.

Equation (11) can now be rewritten in a simpler way as follows.

$$\lfloor \mathit{Safe}(ua) \rfloor = \left\lfloor \bigsqcup_{h \in \lfloor H \rfloor} \{ \mathscr{C}_h \mid \mathscr{C}_h \text{ is a minimal } h\text{-predecessor of } \mathscr{C}' \in \lfloor \mathit{LSafe}(ua) \rfloor \} \right\rfloor \quad (14)$$

We can go further by reconsidering the notions of minimal $\overline{a}$- and h-predecessor with the new operator $\sqcup$. From the definition of minimal predecessor, we immediately get that $\mathscr{C}$ is a minimal $\overline{a}$-predecessor of $\mathscr{C}'$ iff $\mathscr{C}$ belongs to the set $\left\lfloor \bigsqcup_{(q',\sigma') \in \mathscr{C}'} \left\{ \{(q,\sigma)\} \mid (q,\sigma) \xrightarrow{\overline{a}} (q',\sigma') \right\} \right\rfloor$, and similarly for h-minimal predecessors. We provide in Appendix C.3 a similar improvement for $\lfloor \mathit{Safe}(a) \rfloor$.

5.3 Using a SAT Solver to Find Minimal Predecessors

The operator $\sqcup$ allows to compute Step 1 and Step 2. Equation (13) accelerates these computations. In this section, we propose a method to compute Step 1 based on a SAT solver (a similar approach also works for Step 2, see Appendix C.4).

A SAT solver is an algorithm used to efficiently test the satisfiability of a boolean formula φ, that is, to check whether there exists a valuation v of the boolean variables of φ that makes φ true. In this case we say that v is a *model* of φ, denoted by $v \models \varphi$.

In Step 1, the computation of set $\lfloor \mathit{LSafe}(ua) \rfloor$ from $\lfloor \mathit{Safe}(u) \rfloor$ is given in (7):

$$\lfloor \mathit{LSafe}(ua) \rfloor = \lfloor \{ \mathscr{C} \mid \mathscr{C} \text{ is an } \overline{a}\text{-predecessor of } \mathscr{C}' \in \lfloor \mathit{Safe}(u) \rfloor \} \rfloor .$$

We recall that $\mathscr{C}$ is an $\overline{a}$-predecessor of $\mathscr{C}'$ if for all $(q',\sigma') \in \mathscr{C}'$, there exists $(q,\sigma) \in \mathscr{C}$ such that $(q,\sigma) \xrightarrow{\overline{a}} (q',\sigma')$. We also recall that $\mathscr{C} \in \lfloor \mathit{LSafe}(ua) \rfloor$ is a finite object such that $\mathscr{C} \subseteq Q \times \Gamma^k$ for some k. Let us associate a boolean variable x_c to each configuration $c \in Q \times \Gamma^k$.

We consider the following boolean formula $\varphi_{\overline{a}}$:

$$\varphi_{\overline{a}} = \bigvee_{\mathscr{C}' \in \lfloor \mathit{Safe}(u) \rfloor} \bigwedge_{c' \in \mathscr{C}'} \bigvee_{c \xrightarrow{\overline{a}} c'} x_c,$$

Let $v_{\mathscr{C}}$ be the valuation such that $v_{\mathscr{C}}(x_c) = 1$ iff $c \in \mathscr{C}$. We immediately obtain that:

$$v_{\mathscr{C}} \models \varphi_{\overline{a}} \qquad \text{iff} \qquad \mathscr{C} \in \mathit{LSafe}(ua) \cap Q \times \Gamma^k.$$

We define an ordering over valuations as follows, in a way to have a notion of minimal models equivalent to $\subseteq$-minimal elements of $\mathit{LSafe}(ua)$.

Let V be a set of boolean variables, let v and v' be two valuations over V. We define $v' \leq v$ iff for all variables $x \in V$, $v'(x) = 1 \implies v(x) = 1$. We denote

$v' < v$ if $v' \leq v$ and $v' \neq v$. Given φ a boolean formula over V, we say that a model v of φ is *minimal* if for all model v' of φ, we have $v' \leq v \implies v' = v$. We get the next characterization.

Lemma 3. *$v_{\mathscr{C}}$ is a minimal model of $\varphi_{\overline{a}}$ iff $\mathscr{C} \in \lfloor LSafe(ua) \rfloor$.*

We can now explain how to compute all the minimal models of formula $\varphi_{\overline{a}}$. Let φ be a boolean formula over V.
First, we explain, knowing a model v of φ, how to compute a model v' of φ such that $v' < v$ (if it exists). Consider the next formula φ':

$$\varphi' = \varphi \wedge (\bigwedge_{x \in V_0} \neg x) \wedge (\bigvee_{x \in V_1} \neg x)$$

where V_0 (respectively V_1) is the set of all variables $x \in V$ such that $v(x) = 0$ (resp. $v(x) = 1$). If φ' has a model v', it follows from the definition of φ' that v' is a model of φ such that $v' < v$. Otherwise, v is a minimal model of φ. So from a model of φ we can compute a minimal model of φ by repeating the above procedure.
Second, let us explain how to compute all the minimal models of φ. Suppose that we already know some minimal model v of φ, and let V_1 be the set of variables $x \in V$ such that $v(x) = 1$. Consider the formula

$$\varphi' = \varphi \wedge (\bigvee_{x \in V_1} \neg x).$$

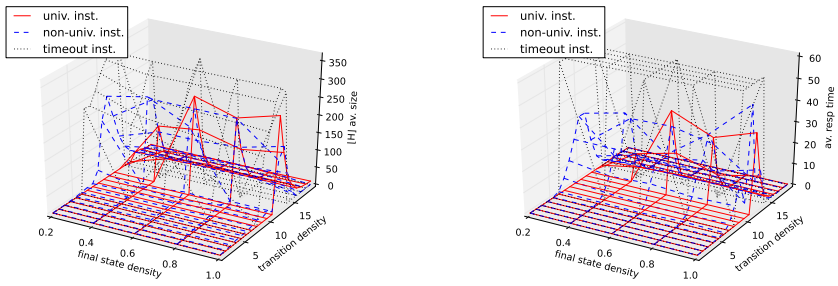
Then a model v' of φ', if it exists, is a model of φ such that neither $v' < v$ (since v is minimal) nor $v < v'$ (by definition of φ'). With the previous procedure, we thus get a minimal model of φ that is distinct from v. In this way we can compute all minimal models of φ.

5.4 Improved Algorithm and Data Structures

Let us come back to Algorithm 2 by indicating the underlying data structures and the improvements resulting from the previous three sections.

As explained in Section 5.1, we restrict the computations to the set $\lfloor H \rfloor$ of minimal hedges. Notice that Algorithm 3 that computes the set $\{\mathsf{rel}_h \mid h \in H\}$ (see Appendix) can be easily adapted to compute the set of its $\leq$-minimal elements. In the main loop of Algorithm 2, Steps 1 and 2 can be computed either with the new operator $\sqcup$ or with a SAT solver (see Sections 5.2 and 5.3 resp.).

Efficient data structures are used both for the relations associated to minimal hedges and for the antichains $\lfloor H \rfloor$, $\lfloor Safe(u) \rfloor$ and $\lfloor LSafe(u) \rfloor$. A relation rel_h, with $h \in \lfloor H \rfloor$, is stored as an array of bit-vectors. In this way the composition is computed efficiently using bit-operations, as well as the number of elements of the relation. A hash table is used to store each antichain, such that elements with different weights are stored in different lists. In the case of $\lfloor H \rfloor$, the weight is the number of elements of $\mathsf{rel}_h \in \lfloor H \rfloor$. In the case of $\lfloor Safe(u) \rfloor$ (resp. $\lfloor LSafe(u) \rfloor$), the weight is the number of elements of $\mathscr{C} \in \lfloor Safe(u) \rfloor$ (resp. $\mathscr{C} \in \lfloor LSafe(u) \rfloor$). In this way, comparing a new element with the elements of the antichain is made more efficient, by limiting the comparison with elements of the same weight.



(a) Size of $\lfloor H \rfloor$.

(b) Time to compute $\lfloor H \rfloor$.

Fig. 3: Space and time consumption for computing $\lfloor H \rfloor$ on 50 VPAs of various densities, $|Q| = 10$, $|\Sigma| = |\Gamma| = 3$, and timeout of 60s.

6 Experiments

We have implemented Algorithm 2 in a prototype tool together with the data structures and improvements proposed in Section 5 (following both approaches, using operator $\sqcup$ and a SAT solver). We mainly use Python, with C binding to the Glucose SAT solver (first ranked in recent SAT competitions) [Glucose].

The experimental tests are performed on two different benchmarks, one composed of randomly generated VPAs, and another one based on the translation of XPath expressions to VPAs. The experiments were run on a PC equipped with an Intel i7 2.8GHz processor, 6 GB of RAM and running Linux Ubuntu 3.2.

6.1 Randomly Generated VPAs

During the random generation of VPAs $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$, $|Q_i|$ is fixed to 1, and several parameters vary: the sizes $|Q|$, $|\Sigma|$ and $|\Gamma|$, the density of final states $\frac{|Q_f|}{|Q|}$, and the transition density[7]. Our online algorithm for checking right-universality needs to compute the set $\lfloor H \rfloor$ of minimal hedges as a preprocessing. This set can be huge and thus lead to a timeout. In Figure 3a (resp. Figure 3b), we indicate the average size of $\lfloor H \rfloor$ (resp. the average execution time to compute it) for randomly generated VPAs with variable transition density (from 1 to 19) and variable final state density. In this test, we distinguish universal automata from non-universal ones since a universal VPA is right-universal w.r.t all $u \in \mathit{Pref}(T_\Sigma) \setminus \{\epsilon\}$. The two figures show that timeout happens for instances with transition density around 12, and that the density of final states has few influence.

[7] equal to the number of outgoing transitions per state and per symbol.

$\|Q\|$	universality		right-universality		null rel_h		timeout (preproc.)		timeout (online)	
	tr. density: 8	tr. density: 16	8	16	8	16	8	16	8	16
10	12 (0.59/0.01)	40 (1.19/0.01)	22 (16.62/9.36)	44 (11.00/1.10)	52 (0.67/0.01)	0	2	2	2	4
20	13 (9.35/0.01)	29 (0.86/0.01)	14 (20.44/30.85)	41 (5.18/1.07)	46 (9.62/0.01)	0	14	17	3	3
30	9 (20.22/0.01)	34 (13.39/0.01)	11 (21.23/5.85)	34 (27.31/3.43)	40 (1.23/0.01)	0	30	21	0	1

Table 1: Number for each type (universal, right-universal, null rel_h, timeout) among 90 instances, with a timeout of 300s, and various number of states and transition densities. Average (preprocessing/online) time is given in parentheses (in s).

VPA id	Response	Time (s.)	$\lfloor H \rfloor$	$\lvert Reach(u) \rvert$	$\lvert \lfloor Safe(u) \rfloor \rvert$
q20-a02-x02-o16-c16-f0.5-00	right-universal w.r.t. a_1	0.03	3	4	33
q20-a02-x04-o16-c16-f0.5-02	right-universal w.r.t. a_1	0.07	3	8	53
q20-a02-x04-o16-c16-f0.5-06	right-universal w.r.t. a_0	0.06	3	5	54
q20-a03-x02-o16-c16-f0.5-06	right-universal w.r.t. a_2	0.13	12	4	32
q20-a03-x03-o16-c16-f0.5-06	right-universal w.r.t. a_0	0.37	23	4	59
q20-a03-x03-o16-c16-f0.5-09	right-universal w.r.t. a_2a_0	1.74	13	14	179
q20-a03-x04-o16-c16-f0.5-07	right-universal w.r.t. a_2	1.02	44	5	116
q20-a04-x02-o16-c16-f0.5-03	right-universal w.r.t. a_3	0.71	71	5	75
q20-a04-x02-o16-c16-f0.5-07	right-universal w.r.t. a_0	1.07	81	5	74
q20-a04-x03-o16-c16-f0.5-03	right-universal w.r.t. a_2	15.88	359	5	447

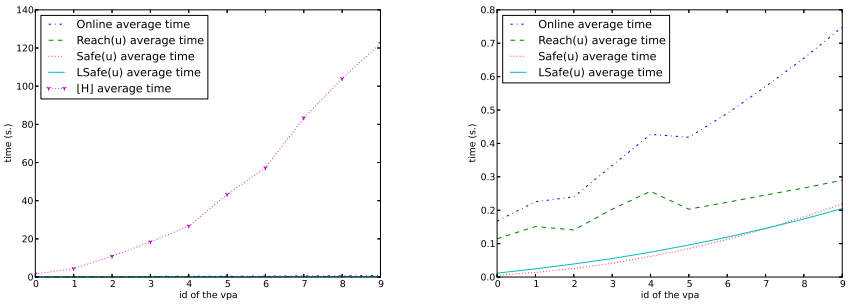
Table 2: Time and data structures size on random VPAs, with transition density of 16, $|Q| = 20$, $|\Sigma| = |\Gamma| \in \{2, 3, 4\}$, and timeout of 300s.

In the next experiment, we study the behavior of our algorithm on 90 random instances of size 10 (resp. 20, 30), with fixed transition density 8 (resp. 16) and fixed density 0.5 of final states[8]. The considered tree t_0 is a complete binary tree up to height 3 filled with randomly generated letters of Σ. We only comment the experiment using a SAT solver since it outperforms the approach with operator $\sqcup$. We observe several behaviors (see Table 1): many automata are either universal, or right-universal w.r.t. u with $|u| = 1$ (except 7 cases where $|u| = 2$), or exhibits a hedge h with rel_h being the null relation[9]; the number of timeout increases with $|Q|$. Table 2 indicates, for some representative VPAs, the execution time and the sizes of $\lfloor H \rfloor$, $Reach(u)$ and $\lfloor Safe(u) \rfloor$.

When it declares that a given VPA is right-universal w.r.t. u, our algorithm has the nice property that it reproduces the *same execution* (thus with the same time) for each tree t_0 such that u is prefix of $[t_0]$. This is clearly not the case for the *membership algorithm* that computes $Reach([t_0])$ and checks if it contains a final configuration. For instance, on a random VPA with size $|Q| = 20$ and transition density 8, our algorithm consumes less than 5s to declare that the

[8] to deal with managable and not too small sets $\lfloor H \rfloor$.

[9] This means that the automaton is never right-universal w.r.t. u, for any proper prefix u of $[t_0]$. Therefore, in such cases, our algorithm is slower that the membership algorithm.



(a) With the $\lfloor H \rfloor$ average time curve. (b) Without the $\lfloor H \rfloor$ average time curve.

Fig. 4: Average time for computing $Safe(u)$, $LSafe(u)$ and $Reach(u)$ and overall computation for 15 random trees.

automaton is right-universal w.r.t. a, whereas the membership algorithm takes more than 300s as soon as the height of a binary tree t_0 with an a-root is equal to 8 (see also Appendix D.1).

6.2 VPAs resulting from XPath translation

Our second benchmark is based on VPAs obtained from queries over XML documents expressed in the XPath language, and then translated into VPAs. This translation was performed by the QuiXProc tool, as described in [GN11]. This family of XPath expressions yields VPAs of linear size increase (VPA with id i has $16+11i$ states), and looks for some complex patterns in the tree. In Figure 4 we report the time used by our algorithm on randomly generated trees, for this family of VPAs. This shows that for real-world VPAs, the size of $\lfloor H \rfloor$ is outside the hardness threshold exhibited in Figure 3. For instance, for the VPA with id 9 and size $|Q| = 115$, $\lfloor H \rfloor$ is computed in about 120s. Moreover, Figure 4b shows the efficiency of our online algorithm (less than 0.8s, see also Appendix D.2).

Acknowledgements

We thank Grégoire Sutre for useful discussions, and the QuiXProc team for translating XPath expressions of our benchmark to VPAs. We also thank one of the reviewers who helped us to improve the presentation of this article. The second author is supported by a grant from FRIA. This work was also partially supported by CNRS SOSP project.

References

[AEM04] R. Alur, K. Etessami, and P. Madhusudan, *A temporal logic of nested calls and returns*, Proc. TACAS, LNCS **2988**, Springer, 2004, 67–481.

[AM04] R. Alur and P. Madhusudan, *Visibly pushdown languages*, Proc. STOC, ACM-Press, 2004, pp. 202–211.

[AM09] ———, *Adding nesting structure to words*, J. ACM **56** (2009), 1–43.

[BYFJ05] Z. Bar-Yossef, M. Fontoura, and V. Josifovski, *Buffering in query evaluation over XML streams*, Proc. PODS, ACM-Press, 2005, 216–227.

[BJLW08] M. Benedikt, A. Jeffrey, and R. Ley-Wild, *Stream Firewalling of XML Constraints*, Proc. SIGMOD Conference, ACM-Press, 2008, 487–498.

[BHH+08] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar, *Antichain-based universality and inclusion testing over nondeterministic finite tree automata*, Proc. CIAA. LNCS **5148**, Springer, 2008, 57–67.

[BDG12] V. Bruyère, M. Ducobu, and O. Gauwin, *Visibly pushdown automata on trees: universality and u-universality*, CoRR abs/1205.2841, 2012.

[BDG13] ———, *Visibly pushdown automata: Universality and inclusion via antichains*, Proc. LATA, LNCS **7810**, Springer, 2013, 190–201.

[BYBC10] T. Bultan, F. Yu, and A. Betin-Can, *Modular verification of synchronization with reentrant locks*, Proc. MEMOCODE, IEEE Computer Society, 2010, 59–68.

[CA07] S. Chaudhuri and R. Alur, *Instrumenting C programs with nested word monitors*, Proc. SPIN, LNCS **4595**, Springer, 2007, 279–283.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, *Tree automata techniques and applications*, Available on: http://www.grappa.univ-lille3.fr/tata, 2007.

[DDHR06] M. De Wulf, L. Doyen, T. Henzinger, J.-F. Raskin, *Antichains: A new algorithm for checking universality of finite automata*, Proc. CAV., LNCS **4144**, Springer, 2006, 17–30.

[FJJ+12] M. Fredrikson, R. Joiner, S. Jha, T. Reps, P. Porras, H. Saïdi, and V. Yegneswaran, *Efficient runtime policy enforcement using counterexample-guided abstraction refinement*, Proc. CAV, LNCS **7358**, Springer, 2012, 548–563.

[FKL13] O. Friedmann, F. Klaedtke, and M. Lange, *Ramsey goes visibly pushdown*, Proc. ICALP, LNCS **7966**, Springer, 2013, 224-237.

[GN11] O. Gauwin and J. Niehren, *Streamable fragments of forward XPath*, Proc. CIAA, LNCS **6807**, Springer, 2011, 3–15.

[GNR08] O. Gauwin, J. Niehren, and Y. Roos, *Streaming tree automata*, Information Processing Letters **109** (2008), 13–17.

[GNT09] O. Gauwin, J. Niehren, and S. Tison, *Earliest query answering for deterministic nested word automata*, Proc. FCT, LNCS **5699**, Springer, 2009 121–132.

[Glucose] Glucose, `www.lri.fr/~simon/?page=glucose`.

[RCB08] G. Roşu, F. Chen, and T. Ball, *Runtime verification*, Proc. RV, LNCS **5289**, Springer, 2008, 51–68.

[TO12] N.V. Tang and H. Ohsaki, *On model checking for visibly pushdown automata*, Proc. LATA, LNCS **7183**, Springer, 2012, 408–419.

A Complements to Section 3

A.1 Proof of Theorem 1

($\Rightarrow$) Assume that $\mathcal{A}$ is u-universal. Consider the set $\mathscr{C}$ of configurations (q,σ) of $\mathcal{A}$ such that there exists $v \in (\Sigma \cup \overline{\Sigma})^*$, $q_i \in Q_i$ and $q_f \in Q_f$ verifying $uv \in [T_\Sigma]$ and $(q_i,\epsilon) \xrightarrow{u} (q,\sigma) \xrightarrow{v} (q_f,\epsilon)$. We have $\mathscr{C} \subseteq \mathit{Reach}(u)$.

Let v be such that $uv \in [T_\Sigma]$. As $\mathcal{A}$ is right-universal w.r.t. u, there exists a configuration $(q,\sigma) \in \mathscr{C}$ such that $(q,\sigma) \xrightarrow{v} (q_f,\epsilon)$ with $q_f \in Q_f$. Hence $\mathscr{C} \in \mathit{Safe}(u)$. As $\mathit{Safe}(u)$ is $\subseteq$-upward closed, we get $\mathit{Reach}(u) \in \mathit{Safe}(u)$.

($\Leftarrow$) Assume now that $\mathit{Reach}(u) \in \mathit{Safe}(u)$, and let v be such that $uv \in [T_\Sigma]$. As $\mathit{Reach}(u) \in \mathit{Safe}(u)$, there exists $(q,\sigma) \in \mathit{Reach}(u)$ and $p \in Q_f$ such that $(q,\sigma) \xrightarrow{v} (p,\epsilon)$. Thus, $uv \in L(\mathcal{A})$, and $\mathcal{A}$ is right-universal w.r.t. u.

A.2 Proof of Lemma 1

($\supseteq$) Assume $\mathscr{C} \in \mathit{Safe}(u)$, and let v be such that $u[h]v \in [T_\Sigma]$. As h is a hedge, we have $uv \in [T_\Sigma]$. As $\mathscr{C} \in \mathit{Safe}(u)$, there exists $(q,\sigma) \in \mathscr{C}$ such that $(q,\sigma) \xrightarrow{v} (p,\epsilon)$ with $p \in Q_f$. So $\mathscr{C} \in \mathit{Safe}(u[h])$.

($\subseteq$) Conversely, assume $\mathscr{C} \in \mathit{Safe}(u[h])$. Let v be such that $uv \in [T_\Sigma]$. We also have $u[h]v \in [T_\Sigma]$, so there exists $(q,\sigma) \in \mathscr{C}$ such that $(q,\sigma) \xrightarrow{v} (p,\epsilon)$ with $p \in Q_f$. Thus $\mathscr{C} \in \mathit{Safe}(u)$.

The proof is the same for $\mathit{LSafe}(u[h]) = \mathit{LSafe}(u)$, except that we only consider v of the form $\overline{a}v'$.

A.3 Proof of Proposition 1

Before giving the proof of Proposition 1, we introduce the *Post* operator and Proposition 2. From $u \in (\Sigma \cup \overline{\Sigma})^*$ and a set of configurations $\mathscr{C} \subseteq Q \times \Gamma^*$, we define $\mathit{Post}_u(\mathscr{C})$ as the set

$$\mathit{Post}_u(\mathscr{C}) = \{(q',\sigma') \in Q \times \Gamma^* \mid \exists (q,\sigma) \in \mathscr{C}, (q,\sigma) \xrightarrow{u} (q',\sigma')\}.$$

Proposition 2. *Let $ua \in \mathit{Pref}(T_\Sigma)$ with $a \in \Sigma$.*

$$\mathscr{C} \in \mathit{LSafe}(ua) \iff \mathit{Post}_{\overline{a}}(\mathscr{C}) \in \mathit{Safe}(u) \tag{15}$$

$$\mathscr{C} \in \mathit{Safe}(ua) \iff \forall h \in H_\Sigma, \mathit{Post}_{[h]}(\mathscr{C}) \in \mathit{LSafe}(ua) \tag{16}$$

Proof. ($15, \Rightarrow$) Let $\mathscr{C} \in \mathit{LSafe}(ua)$ and $\mathscr{C}' = \mathit{Post}_{\overline{a}}(\mathscr{C})$. Let us show that $\mathscr{C}' \in \mathit{Safe}(u)$. By Lemma 1, it is sufficient to prove that $\mathscr{C}' \in \mathit{Safe}(ua\overline{a})$. Let v such that $ua\overline{a}v \in [T_\Sigma]$. As $\mathscr{C} \in \mathit{LSafe}(ua)$ and $\overline{a}v$ starts with $\overline{a} \in \overline{\Sigma}$, there exists $(q,\sigma) \in \mathscr{C}$ and (q',σ') such that $(q,\sigma) \xrightarrow{\overline{a}} (q',\sigma') \xrightarrow{v} (p,\epsilon)$ for some $p \in Q_f$. By definition of $\mathit{Post}_{\overline{a}}(\mathscr{C})$ we have $(q',\sigma') \in \mathscr{C}'$ and thus $\mathscr{C}' \in \mathit{Safe}(ua\overline{a})$.

($15, \Leftarrow$) For the converse, let $\mathscr{C}' = \mathit{Post}_{\overline{a}}(\mathscr{C}) \in \mathit{Safe}(u) = \mathit{Safe}(ua\overline{a})$. Let us show that $\mathscr{C} \in \mathit{LSafe}(ua)$. Let v be such that $uav \in [T_\Sigma]$ and $v = \overline{b}v'$. We necessarily have $\overline{a} = \overline{b}$. As $\mathscr{C}' \in \mathit{Safe}(ua\overline{a})$, there exists $(q',\sigma') \in \mathscr{C}'$ such

that $(q', \sigma') \xrightarrow{v'} (p, \epsilon)$ with $p \in Q_f$. By definition of $Post_{\overline{a}}(\mathscr{C})$, there also exists $(q, \sigma) \in \mathscr{C}$ such that $(q, \sigma) \xrightarrow{\overline{a}} (q', \sigma')$ and thus $(q, \sigma) \xrightarrow{v=\overline{a}v'} (p, \epsilon)$ with $p \in Q_f$.

$(16, \Rightarrow)$ Let $\mathscr{C} \in Safe(ua)$ and $h \in H_\Sigma$. Let us show that $\mathscr{C}' = Post_{[h]}(\mathscr{C})$ is in $LSafe(ua)$. Let v such that $uav \in [T_\Sigma]$ and $v = \overline{b}v'$. We must have $\overline{a} = \overline{b}$. We also have $ua[h]\overline{a}v' \in [T_\Sigma]$. As $\mathscr{C} \in Safe(ua)$, there exists $(q, \sigma) \in \mathscr{C}$ such that $(q, \sigma) \xrightarrow{[h]} (q', \sigma') \xrightarrow{v=\overline{a}v'} (p, \epsilon)$ with $p \in Q_f$. By definition of $Post_{[h]}(\mathscr{C})$, $(q', \sigma') \in \mathscr{C}'$.

$(16, \Leftarrow)$ Let us assume that for every hedge $h \in H_\Sigma$, $Post_{[h]}(\mathscr{C}) \in LSafe(ua)$. Let us show that $\mathscr{C} \in Safe(ua)$. Let v be such that $uav \in [T_\Sigma]$. Then we have $uav = ua[h]\overline{a}v'$ for some $h \in H_\Sigma$. As $\mathscr{C}' = Post_{[h]}(\mathscr{C}) \in LSafe(ua)$, $\overline{a}v'$ starts with $\overline{a} \in \overline{\Sigma}$ and $ua\overline{a}v' \in [T_\Sigma]$, there exists $(q', \sigma') \in \mathscr{C}'$ such that $(q', \sigma') \xrightarrow{\overline{a}v'} (p, \epsilon)$ for some $p \in Q_f$. Hence, by definition of $Post_{[h]}(\mathscr{C})$, there also exists $(q, \sigma) \in \mathscr{C}$ such that $(q, \sigma) \xrightarrow{[h]} (q', \sigma') \xrightarrow{\overline{a}v'} (p, \epsilon)$ with $p \in Q_f$. □

Predecessors closely relate to the *Post* operator, in the following sense.

Lemma 4. *$\mathscr{C}$ is an $\overline{a}$-predecessor of $Post_{\overline{a}}(\mathscr{C})$. If $\mathscr{C} \in Pred_{\overline{a}}(\mathscr{C}')$, then $\mathscr{C}' \subseteq Post_{\overline{a}}(\mathscr{C})$. Both properties also hold for $Post_{[h]}(\mathscr{C})$.*

The proof of Proposition 1 is then the following one.

Proof (of Proposition 1). $(3, \Rightarrow)$ Let $\mathscr{C} \in LSafe(ua)$. Then by Proposition 2, $Post_{\overline{a}}(\mathscr{C}) \in Safe(u)$. Moreover, $\mathscr{C}$ is an $\overline{a}$-predecessor of $Post_{\overline{a}}(\mathscr{C})$ by Lemma 4.

$(3, \Leftarrow)$ Let $\mathscr{C}$ be an $\overline{a}$-predecessor of $\mathscr{C}'$, with $\mathscr{C}' \in Safe(u)$. By Lemma 4, $\mathscr{C}' \subseteq Post_{\overline{a}}(\mathscr{C})$. As $Safe(u)$ is $\subseteq$-upward closed, we also have $Post_{\overline{a}}(\mathscr{C}) \in Safe(u)$, so $\mathscr{C} \in LSafe(ua)$ by Proposition 2.

(4) Same proofs, except that $\overline{a}$ has to be replaced by h, for all $h \in H_\Sigma$.

□

B Complements to Section 4

B.1 Proof of Lemma 2

Let us recall the definition of h-predecessor: $\mathscr{C}$ is a h-predecessor of $\mathscr{C}'$ if $\forall (q', \sigma) \in \mathscr{C}',\ \exists (q, \sigma) \in \mathscr{C},\ (q, \sigma) \xrightarrow{h} (q', \sigma)$. Hence if $h \sim h'$, then $\mathscr{C}$ is a h-predecessor of $\mathscr{C}'$ iff $\mathscr{C}$ is a h'-predecessor of $\mathscr{C}'$.

B.2 Algorithm computing H

We propose an algorithm for computing such a set H from a VPA $\mathcal{A}$. Algorithm 3 is based on the definition of hedges, adapted to their associated functions:

- ϵ is the empty hedge, and $\mathsf{rel}_\epsilon(q) = \{q\}$ for every $q \in Q$. We write this function id_Q.
- if h_1, h_2 are two hedges, then $h_1 h_2$ is a hedge, and $\mathsf{rel}_{h_1 h_2} = \mathsf{rel}_{h_2} \circ \mathsf{rel}_{h_1}$.

– if h is a hedge and $a \in \Sigma$, then $ah\overline{a}$ is a hedge, and $\mathsf{rel}_{ah\overline{a}}(q)$ is the set of states q' such that there exists $\gamma \in \Gamma$ verifying:

$$(q, \epsilon) \xrightarrow{a} (p, \gamma) \quad \text{and} \quad (p', \gamma) \xrightarrow{\overline{a}} (q', \epsilon) \quad \text{with } p' \in \mathsf{rel}_h(p).$$

Algorithm 3 uses the variables *ToProcess* and *Functions* with the following meaning. *Functions* contains initially the identity relation id_Q; at the end of the computation, it contains all functions rel_h, for $h \in H_\Sigma$. *ToProcess* contains all the newly constructed functions, and these functions are used to create other new functions as described in the previous definition by induction.

Algorithm 3 Computing all functions rel_h, for $h \in H_\Sigma$.

```
function HedgeFunctions(A)
    Functions ← {id_Q}
    ToProcess ← {id_Q}
    while ToProcess ≠ ∅ do
        fct ← Pop(ToProcess)
        NewFunctions ← ∅
        for f ∈ Functions do
            NewFunctions ← NewFunctions ∪ {f ∘ fct, fct ∘ f}
        end for
        for a ∈ Σ do
            f ← f_∅        // f_∅ maps every q ∈ Q to ∅
            for q -a:γ-> p ∈ Δ and p' -ā:γ-> q' ∈ Δ with p' ∈ fct(p) do
                f(q) ← f(q) ∪ {q'}
            end for
            NewFunctions ← NewFunctions ∪ {f}
        end for
        ToProcess ← ToProcess ∪ (NewFunctions \ Functions)
        Functions ← Functions ∪ NewFunctions
    end while
    return Functions
end function
```

Proposition 3. *Algorithm 3 computes the set* $\{\mathsf{rel}_h \mid h \in H_\Sigma\}$.

Proof. Let *Functions* be the set computed by Algorithm 3. Clearly, *Functions* $\subseteq \{\mathsf{rel}_h \mid h \in H_\Sigma\}$. Assume for contradiction that there exists $r = \mathsf{rel}_h$ with $h \in H_\Sigma$ such that $r \notin$ *Functions*. Clearly, $r \neq id_Q$, and we can suppose wlog that either $r = r'_2 \circ r'_1$ with $r'_1, r'_2 \in$ *Functions* $\setminus \{id_Q\}$, or there exists $r' \in$ *Functions* such that for all q, $r(q)$ is the set of q' with $q \xrightarrow{a:\gamma} p \in \Delta$, $p' \xrightarrow{\overline{a}:\gamma} q' \in \Delta$ and $p' \in r'(p)$. Consider the first case. When they have been constructed by Algorithm 3, both r'_1 and r'_2 have been added to *ToProcess* and to *Functions*. After the last element (among r'_1 and r'_2) is popped from *ToProcess*, then $r = r'_2 \circ r'_1$ is built during the loop on $f \in$ *Functions*, which leads to a contradiction. We also have a contradiction in the second case by considering the loop on $a \in \Sigma$. □

B.3 Antichains $\lfloor Safe(u) \rfloor$ and $\lfloor LSafe(u) \rfloor$ are Finite

Proposition 4. *$\lfloor Safe(u) \rfloor$ and $\lfloor LSafe(u) \rfloor$ are finite and only contain finite sets of configurations.*

Proof. We begin with the following observation. Let v be such that $[uv] \in T_\Sigma$ and $(q, \sigma) \xrightarrow{v} (p, \epsilon)$ with $p \in Q_f$. Let u' (resp. v') the word obtained from u (resp. v) by removing all factors that are linearizations of hedges. Then $|u'| = |v'|$ and $|u'| = |\sigma|$.

Let $\mathscr{C} \in Safe(u)$. Then by definition

$$\forall v,\ uv \in [T_\Sigma] \implies \exists (q, \sigma) \in \mathscr{C},\ (q, \sigma) \xrightarrow{v} (p, \epsilon) \text{ with } p \in Q_f.$$

If $\mathscr{C}$ is $\subseteq$-minimal, then every $(q, \sigma) \in \mathscr{C}$ is used for at least one v in the previous definition. Now by the previous observation, each such (q, σ) belongs to $Q \times \Gamma^{|u'|}$. Hence $\mathscr{C} \subseteq Q \times \Gamma^{|u'|}$, and thus both $\mathscr{C}$ and $\lfloor Safe(u) \rfloor$ are finite.

The same arguments hold for proving that $\lfloor LSafe(u) \rfloor$ is finite and contains only finite sets of configurations. □

B.4 Proof of Implication (6)

Let $\mathscr{C} \in \lfloor LSafe(ua) \rfloor$ and let $\mathscr{C}' = Post_{\overline{a}}(\mathscr{C})$. We know from Proposition 2 that $\mathscr{C}' \in Safe(u)$. Let $\mathscr{C}'_0 \subseteq \mathscr{C}'$ such that $\mathscr{C}'_0 \in \lfloor Safe(u) \rfloor$. From the definition of $\mathscr{C}'$ we get:

$$\forall c' \in \mathscr{C}',\ \exists c \in \mathscr{C},\ c \xrightarrow{\overline{a}} c'$$

We build $\mathscr{C}_0$ from these $c \in \mathscr{C}$ but for $c' \in \mathscr{C}'_0$:

$$\mathscr{C}_0 = \{c \in \mathscr{C} \ \mid\ \exists c' \in \mathscr{C}'_0,\ c \xrightarrow{\overline{a}} c'\}$$

Figure 5 illustrates the construction. The set $\mathscr{C}_0$ is an $\overline{a}$-predecessor of $\mathscr{C}'_0$, so us-

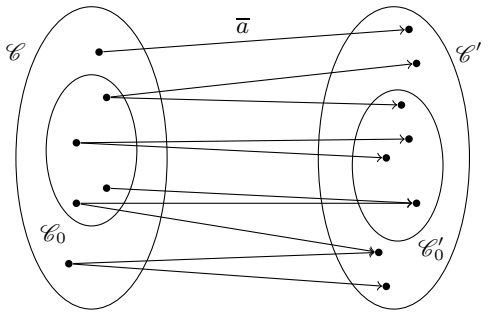


Fig. 5: Construction of $\mathscr{C}_0$

ing Proposition 1, we get $\mathscr{C}_0 \in LSafe(ua)$. Furthermore, $\mathscr{C}_0 \subseteq \mathscr{C} \in \lfloor LSafe(ua) \rfloor$, so $\mathscr{C}_0 = \mathscr{C}$, and $\mathscr{C}$ is obtained as an $\overline{a}$-predecessor of $\mathscr{C}'_0 \in \lfloor Safe(u) \rfloor$.

B.5 Proof of Implication (8)

The proof is in the same vein as for Implication (6). Let $\mathscr{C} \in \lfloor \mathit{Safe}(ua) \rfloor$, and $h \in H_\Sigma$. Let $\mathscr{C}'_h = \mathit{Post}_{[h]}(\mathscr{C})$. By Proposition 2, $\mathscr{C}'_h \in \mathit{LSafe}(ua)$. Let $\mathscr{C}''_h \subseteq \mathscr{C}'_h$ such that $\mathscr{C}''_h \in \lfloor \mathit{LSafe}(ua) \rfloor$. We know that $\forall c' \in \mathscr{C}'_h$, $\exists c \in \mathscr{C}$ such that $c \xrightarrow{[h]} c'$. We define $\mathscr{C}_h = \{c \in \mathscr{C} \mid \exists c' \in \mathscr{C}''_h,\ c \xrightarrow{[h]} c'\}$. For every $h \in H_\Sigma$, $\mathscr{C}_h$ is a h-predecessor of $\mathscr{C}''_h \in \mathit{LSafe}(ua)$. Consider $\mathscr{C}_\cup = \bigcup_{h \in H_\Sigma} \mathscr{C}_h$, then $\mathscr{C}_\cup$ is also a h-predecessor of $\mathscr{C}''_h$. Using Proposition 1, we have $\mathscr{C}_\cup \in \mathit{Safe}(ua)$. As $\mathscr{C}_\cup \subseteq \mathscr{C}$ and $\mathscr{C} \in \lfloor \mathit{Safe}(ua) \rfloor$, we also have that $\mathscr{C}_\cup = \mathscr{C}$. Hence $\mathscr{C}$ verifies that $\forall h \in H_\Sigma$, $\exists \mathscr{C}'' \in \lfloor \mathit{LSafe}(ua) \rfloor$ such that $\mathscr{C}$ is a h-predecessor of $\mathscr{C}''$.

C Complements to Section 5

C.1 Proof of Equation (11)

Let S denote the set

$$\left\{ \mathscr{C} \mid \mathscr{C} = \bigcup_{h \in H} \mathscr{C}_h \text{ with } \mathscr{C}_h \text{ a minimal } h\text{-predecessor of } \mathscr{C}' \in \lfloor \mathit{LSafe}(ua) \rfloor \right\}$$

Let $\mathscr{C} \in S$. We have: $\mathscr{C} = \underbrace{\mathscr{C}_{h_1} \cup \cdots \cup \mathscr{C}_{h_k}}_{h_i \in \lfloor H \rfloor} \cup \underbrace{\mathscr{C}_{h'_1} \cup \cdots \cup \mathscr{C}_{h'_n}}_{h'_i \in H \setminus \lfloor H \rfloor}$. Let us show that $\mathscr{C}_{h_1} \cup \cdots \cup \mathscr{C}_{h_k} \cup \mathscr{C}_{h'_1} \cup \cdots \cup \mathscr{C}_{h'_{n-1}} \in S$ ($\mathscr{C}_{h'_n}$ has been erased). By induction, this will prove that $\mathscr{C}_{h_1} \cup \cdots \cup \mathscr{C}_{h_k} \in S$. We have $h'_n \in H \setminus \lfloor H \rfloor$, so there exists $h_i \in \lfloor H \rfloor$ such that $h_i \leq h'_n$. As $\mathscr{C}_{h_i}$ is a minimal h_i-predecessor of an element $\mathscr{C}'$ in $\lfloor \mathit{LSafe}(ua) \rfloor$, it follows from (10) that $\mathscr{C}_{h_i}$ is also a minimal h'_n-predecessor of $\mathscr{C}'$. So $\mathscr{C}_{h_1} \cup \cdots \cup \mathscr{C}_{h_k} \cup \mathscr{C}_{h'_1} \cup \cdots \cup \mathscr{C}_{h'_{n-1}} \cup \mathscr{C}_{h_i} \in S$.

C.2 Proof of Equation (13)

($\supseteq$) Let $\mathscr{C} \in \lfloor (A \cap B) \cup (A \setminus B \sqcup B \setminus A) \rfloor$. Then $\mathscr{C} \in A \sqcup B$. For contradiction, let us assume that there exists $\mathscr{C}' \subsetneq \mathscr{C}$ such that $\mathscr{C}' \in \lfloor A \sqcup B \rfloor$. If $\mathscr{C}' \in A \cap B$, then $\mathscr{C}' \in (A \cap B) \cup (A \setminus B \sqcup B \setminus A)$, which contradicts $\mathscr{C}$. So $\mathscr{C}' \notin A \cap B$, and assume wlog that $\mathscr{C}' = a \cup b$ with $a \in A \setminus B$ and $b \in B$ (recall that a, b are non empty sets by definition of $\sqcup$ operator). If $b \in A$, then $b \in A \cap B \subseteq A \sqcup B$ and $b \subsetneq \mathscr{C}'$, but this contradicts $\mathscr{C}'$. If $b \notin A$, then $\mathscr{C}' \in A \setminus B \sqcup B \setminus A$, so $\mathscr{C}' \in A \cap B \cup (A \setminus B \sqcup B \setminus A)$, and $\mathscr{C}' \subsetneq \mathscr{C}$, which contradicts $\mathscr{C}$.

($\subseteq$) Let $\mathscr{C} \in \lfloor A \sqcup B \rfloor$. Let us first show that $\mathscr{C} \in (A \cap B) \cup (A \setminus B \sqcup B \setminus A)$. If $\mathscr{C} \in A \cap B$ this is direct. Otherwise $\mathscr{C} = a \cup b$ with $a \in A \setminus B$ and $b \in B$ (the other case is symmetric). If $b \in A$ then $b \in A \cap B \subseteq A \sqcup B$ and $b \subsetneq \mathscr{C}$, which contradicts the definition of $\mathscr{C}$. So $b \in B \setminus A$, and $\mathscr{C} \in A \setminus B \sqcup B \setminus A$. Secondly, assume for contradiction that there exists $\mathscr{C}' \subsetneq \mathscr{C}$ such that $\mathscr{C}' \in \lfloor (A \cap B) \cup (A \setminus B \sqcup B \setminus A) \rfloor$. Then, according to ($\supseteq$), $\mathscr{C}' \in \lfloor A \sqcup B \rfloor$, which contradicts the definition of $\mathscr{C}$.

We get the next corollary of Equation (13):

Corollary 2. *If $A \subseteq B$, then $\lfloor A \sqcup B \rfloor = \lfloor A \rfloor$.*

C.3 *Safe*(a) and Operator $\sqcup$

The $\sqcup$ operator also simplifies the definition of $\lfloor Safe(a)\rfloor$.

Proposition 5. $\lfloor Safe(a)\rfloor = \left\lfloor \bigsqcup_{h\in\lfloor H\rfloor} A_h \right\rfloor$ *with*

$$A_h = \left\{ \{(q,\sigma)\} \mid \exists q_f \in Q_f : (q,\sigma) \xrightarrow{h\overline{a}} (q_f,\epsilon) \right\}.$$

Proof. We consider $\lfloor Safe(a)\rfloor$ as defined in Equation (12). Let S denote the set $\left\{ \mathscr{C} \mid \forall h \in \lfloor H\rfloor, \exists q_f \in Q_f, \exists (q,\sigma) \in \mathscr{C} : (q,\sigma) \xrightarrow{h\overline{a}} (q_f,\epsilon) \right\}$.

1. Every element of $\bigsqcup_{h\in\lfloor H\rfloor} A_h$ belongs to S. Thus $\left\lfloor \bigsqcup_{h\in\lfloor H\rfloor} A_h \right\rfloor \subseteq S$.
2. Let us show that for each $\mathscr{C}$ in S, there exists $\mathscr{C}' \in \bigsqcup_{h\in\lfloor H\rfloor} A_h$ such that $\mathscr{C}' \subseteq \mathscr{C}$. Let $\mathscr{C} \in S$. By definition of S, for all $h \in \lfloor H\rfloor$ there exists $(q_h,\sigma_h) \in \mathscr{C}$ and $q_f \in Q_f$ such that $(q_h,\sigma_h) \xrightarrow{h\overline{a}} (q_f,\epsilon)$. Let $\mathscr{C}' = \{(q_h,\sigma_h) \mid h \in \lfloor H\rfloor\}$. Then $\mathscr{C}' \subseteq \mathscr{C}$ and $\mathscr{C}' \in \bigsqcup_{h\in\lfloor H\rfloor} A_h$ because $\{(q_h,\sigma_h)\} \in A_h, \forall h$.
3. Assume that there exists $\mathscr{C}_* \in \left\lfloor \bigsqcup_{h\in\lfloor H\rfloor} A_h \right\rfloor \setminus \lfloor S\rfloor$. By 1., there exists $\mathscr{C}$ in $\lfloor S\rfloor$ such that $\mathscr{C} \subsetneq \mathscr{C}_*$; and by 2., there exists $\mathscr{C}' \in \bigsqcup_{h\in\lfloor H\rfloor} A_h$ such that $\mathscr{C}' \subseteq \mathscr{C} \subsetneq \mathscr{C}_*$ in contradiction with the definition of $\mathscr{C}_*$. Therefore $\left\lfloor \bigsqcup_{h\in\lfloor H\rfloor} A_h \right\rfloor \subseteq \lfloor S\rfloor$.
4. Let $\mathscr{C} \in \lfloor S\rfloor$. By 2., there exists $\mathscr{C}' \in \left\lfloor \bigsqcup_{h\in\lfloor H\rfloor} A_h \right\rfloor$ such that $\mathscr{C}' \subseteq \mathscr{C}$. By 3., it follows that $\mathscr{C} = \mathscr{C}'$ and thus $\lfloor S\rfloor \subseteq \left\lfloor \bigsqcup_{h\in\lfloor H\rfloor} A_h \right\rfloor$.

In conclusion, $\lfloor Safe(a)\rfloor = \lfloor S\rfloor = \left\lfloor \bigsqcup_{h\in\lfloor H\rfloor} A_h \right\rfloor$ □

C.4 Step 2 with SAT Solvers

The computation of the set $\lfloor Safe(ua)\rfloor$ from $\lfloor LSafe(ua)\rfloor$ can be done with a SAT solver as described in Equation (11) :

$$\lfloor Safe(ua)\rfloor = \left\lfloor \left\{ \mathscr{C} \;\middle|\; \mathscr{C} = \bigcup_{h\in\lfloor H\rfloor} \mathscr{C}_h \text{ with } \mathscr{C}_h \text{ a } h\text{-predecessor of } \mathscr{C}' \in \lfloor LSafe(ua)\rfloor \right\} \right\rfloor.$$

The method is the same as for Step 1, with the only difference that we use the next formula instead of formula $\varphi_{\overline{a}}$:

$$\bigwedge_{h\in\lfloor H\rfloor} \bigvee_{\mathscr{C}'\in\lfloor LSafe(ua)\rfloor} \bigwedge_{c'\in\mathscr{C}'} \bigvee_{c \xrightarrow{h} c'} x_c,$$

D Complements to Section 6

We provide in this section additional experimental results.

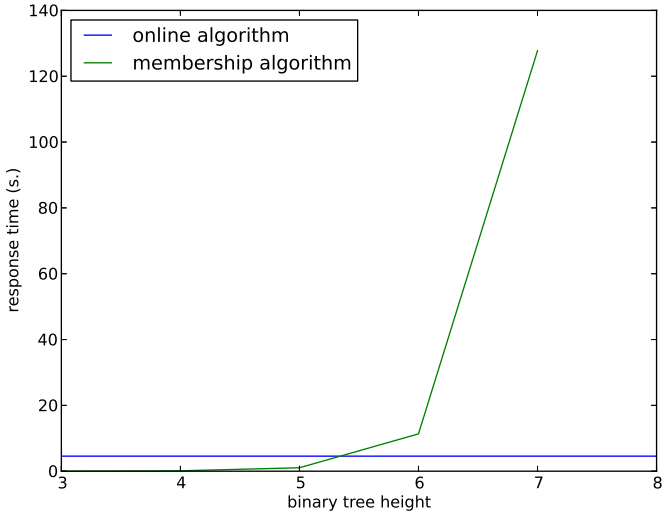


Fig. 6: Execution time of the membership algorithm for a binary tree t_0 of increasing height.

D.1 Randomly Generated VPAs

Figure 6 indicates the execution time of the membership algorithm for a random complete binary tree t_0 of increasing height.

D.2 VPAs resulting from XPath translation

Figure 7 indicates the computation time of our online algorithm for 15 random trees, with a height bounded by 10, and a number of children by node bounded by 15.

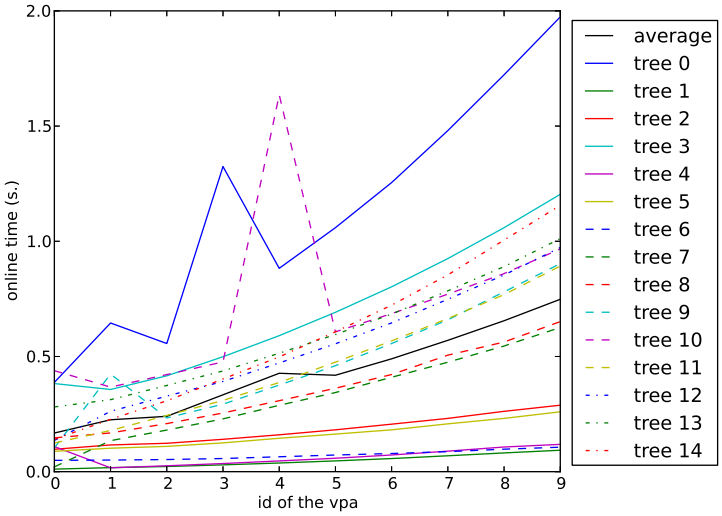


Fig. 7: Computation time of the online algorithm for each random tree.