Ramsey Goes Visibly Pushdown[⋆]

Oliver Friedmann[1], Felix Klaedtke[2], and Martin Lange[3]

[1] LMU Munich, [2] ETH Zurich, and [3] University of Kassel

Abstract. Checking whether one formal language is included in another is vital to many verification tasks. In this paper, we provide solutions for checking the inclusion of the languages given by visibly pushdown automata over both finite and infinite words. Visibly pushdown automata are a richer automaton model than the classical finite-state automata, which allows one, e.g., to reason about the nesting of procedure calls in the executions of recursive imperative programs. The highlight of our solutions is that they do not comprise automata constructions for determinization and complementation. Instead, our solutions are more direct and generalize the so-called Ramsey-based inclusion-checking algorithms, which apply to classical finite-state automata and proved effective there, to visibly pushdown automata. We also experimentally evaluate our algorithms thereby demonstrating the virtues of avoiding determinization and complementation constructions.

1 Introduction

Various verification tasks can be stated more or less directly as inclusion problems of formal languages or comprise inclusion problems as subtasks. For example, the model-checking problem of non-terminating finite-state systems with respect to trace properties boils down to the question whether the inclusion $L(\mathcal{A}) \subseteq L(\mathcal{B})$ for two Büchi automata $\mathcal{A}$ and $\mathcal{B}$ holds, where $\mathcal{A}$ describes the traces of the system and $\mathcal{B}$ the property [22]. Another application of checking language inclusion for Büchi automata appears in size-change termination analysis [13, 19]. Inclusion problems are in general difficult. For Büchi automata it is PSPACE-complete.

From the closure properties of the class of ω-regular languages, i.e., those languages that are recognizable by Büchi automata it is obvious that questions like the one above for model checking non-terminating finite-state systems can be effectively reduced to an emptiness question, namely, $L(\mathcal{A}) \cap L(\mathcal{C}) = \emptyset$, where $\mathcal{C}$ is a Büchi automaton that accepts the complement of $\mathcal{B}$. Building a Büchi automaton for the intersection of the languages and checking its emptiness is fairly easy: the automaton accepting the intersection can be quadratically bigger, the emptiness problem is NLOGSPACE-complete, and it admits efficient implementations, e.g., by a nested depth-first search. However, complementing Büchi automata is challenging. One intuitive reason for this is that not every Büchi automaton has an equivalent deterministic counterpart. Switching to a richer acceptance condition like the parity condition so that determinization would be possible is currently not an option in practice. The known determinization constructions for richer

[⋆] Extended abstract. Omitted details can be found in the full version [15], which is available from the authors' web pages.

acceptance conditions are intricate, although complementation would then be easy by dualizing the acceptance condition. A lower bound on the complementation problem with respect to the automaton size is $2^{\Omega(n \log n)}$. Known constructions for complementing Büchi automata that match this lower bound are also intricate. As a matter of fact, all attempts so far that explicitly construct the automaton $\mathcal{C}$ from $\mathcal{B}$ scale poorly. Often, the implementations produce automata for the complement language that are huge, or they even fail to produce an output at all in reasonable time and space if the input automaton has more than 20 states, see, e.g., [5, 21].

Other approaches for checking the inclusion of the languages given by Büchi automata or solving the closely related but simpler universality problem for Büchi automata have recently gained considerable attention [1, 2, 8–10, 13, 14, 19]. In the worst case, these algorithms have exponential running times, which are often worse than the $2^{\Omega(n \log n)}$ lower bound on complementing Büchi automata. However, experimental results, in particular, the ones for the so-called Ramsey-based algorithms show that the performance of these algorithms is superior. The name *Ramsey-based* stems from the fact that their correctness is established by relying on Ramsey's theorem [20].[1]

The Ramsey-based algorithms for checking universality $L(\mathcal{B}) = \Sigma^\omega$ iteratively build a set of finite graphs starting from a finite base set and closing it off under a composition operation. These graphs capture $\mathcal{B}$'s essential behavior on finite words. The language of $\mathcal{B}$ is not universal iff this set contains graphs with certain properties that witness the existence of an infinite word that is not accepted by $\mathcal{B}$. First, there must be a graph that is idempotent with respect to the composition operation. This corresponds to the fact that all the runs of $\mathcal{B}$ on the finite words described by the graph loop. We must also require that no accepting state occurs on these loops. Second, there must be another graph for the runs on a finite word that reach that loop. To check the inclusion $L(\mathcal{A}) \subseteq L(\mathcal{B})$ the graphs are annotated with additional information about runs of $\mathcal{A}$ on finite words. Here, in case of $L(\mathcal{A}) \nsubseteq L(\mathcal{B})$, the constructed set of graphs contains graphs that witness the existence of at least one infinite word that is accepted by $\mathcal{A}$ but all runs of $\mathcal{B}$ on that word are rejecting. The Ramsey-based approach generalizes to parity automata [16]. The parity condition is useful in modeling reactive systems in which certain modules are supposed to terminate and others are not supposed to terminate. Also, certain Boolean combinations of Büchi (non-termination) and co-Büchi (termination) conditions can easily be expressed as a parity condition. Although parity automata can be translated into Büchi automata, it algorithmically pays off to handle parity automata directly [16].

In this paper, we extend the Ramsey-based analysis to visibly pushdown automata (VPAs) [4]. This automaton model restricts nondeterministic pushdown automata in the way that the input symbols determine when the pushdown automaton pushes or pops symbols from its stack. In particular, the stack heights are identical at the same positions in every run of any VPA on a given input.

[1] Büchi's original complementation construction, which also relies on Ramsey's theorem, shares similarities with these algorithms. However, there is significantly less overhead when checking universality and inclusion directly and additional heuristics and optimizations are applicable [1, 5].

It is because of this syntactic restriction that the class of visibly pushdown languages retains many closure properties like intersection and complementation. VPAs allow one to describe program behavior in more detail than finite-state automata. They can account for the nesting of procedures in executions of recursive imperative programs. Non-regular properties like "an acquired lock must be released within the same procedure" are expressible by VPAs. Model checking of recursive state machines [3] and Boolean programs, which are widely used as abstractions in software model checking, can be carried out in this refined setting by using VPAs for representing the behavior of the programs and the properties. Similar to the automata-theoretic approach to model checking finite-state systems, checking the inclusion of the languages of VPAs is vital here. This time, the respective decision problem is even EXPTIME-complete. Other applications for checking language inclusion of VPAs when reasoning about recursive imperative programs also appear in conformance checking [11] and in the counterexample-guided-abstraction-refinement loop [17].

A generalization of the Ramsey-based approach to VPAs is not straightforward since the graphs that capture the essential behavior of an automaton must also account for the stack content in the runs. Moreover, to guarantee termination of the process that generates these graphs, an automaton's behavior of all runs must be captured within finitely many such graphs. In fact, when considering pushdown automata in general such a generalization is not possible since the universality problem for pushdown automata is undecidable. We circumvent this problem by only considering graphs that differ in their stack height by at most one, and by refining the composition of such graphs in comparison to the unrestricted way that graphs can be composed in the Ramsey-based approach to finite automata. Then the composition operation only needs to account for the top stack symbols in all the runs described by the graphs, which yields a finite set of graphs in the end.

The main contribution of this paper is the generalization of the Ramsey-based approach for checking universality and language inclusion for VPAs over infinite inputs, where the automata's acceptance condition is stated as a parity condition. This approach avoids determinization and complementation constructions. The respective problems where the VPAs operate over finite inputs are special cases thereof. We also experimentally evaluate the performance of our algorithms showing that the Ramsey-based inclusion checking is more efficient than methods that are based on determinization and complementation.

The remainder of this paper is organized as follows. In Sect. 2, we recall the framework of VPAs. In Sect. 3, we provide a Ramsey-based universality check for VPAs. Note that universality is a special case of language inclusion. We treat universality in detail to convey the fundamental ideas first. In Sect. 4, we extend this to a Ramsey-based inclusion check for VPAs. In Sect. 5, we report on the experimental evaluation of our algorithms. In Sect. 6, we draw conclusions.

2 Preliminaries

Words. The set of finite words over the alphabet Σ is Σ^* and the set of infinite words over Σ is Σ^ω. Let $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$, where ε is the empty word. The length

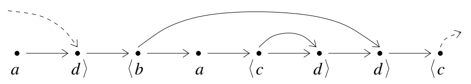

Figure 1. Nested word $w = adbacddbc$ with $\Sigma_{\mathsf{int}} = \{a\}$, $\Sigma_{\mathsf{call}} = \{b, c\}$, and $\Sigma_{\mathsf{ret}} = \{d\}$. Its pending positions are 1 and 7 with $w_1 = d$ and $w_7 = c$. The call position 2 with $w_2 = b$ matches with the return position 6 with $w_6 = d$. The positions 4 and 5 also match.

of a word w is written as $|w|$, where $|w| = \omega$ when w is an infinite word. For a word w, w_i denotes the letter at position $i < |w|$ in w. That is, $w = w_0 w_1 \dots$ if w is infinite and $w = w_0 w_1 \dots w_{n-1}$ if w is finite and $|w| = n$. With $\inf(w)$ we denote the set of letters of Σ that occur infinitely often in $w \in \Sigma^\omega$.

Nested words [4] are linear sequences equipped with a hierarchical structure, which is imposed by partitioning an alphabet Σ into the pairwise disjoint sets Σ_{int}, Σ_{call}, and Σ_{ret}. For a finite or infinite word w over Σ, we say that the position $i \in \mathbb{N}$ with $i < |w|$ is an *internal position* if $w_i \in \Sigma_{\mathsf{int}}$. It is a *call position* if $w_i \in \Sigma_{\mathsf{call}}$ and it is a *return position* if $w_i \in \Sigma_{\mathsf{ret}}$. When attaching an opening bracket $\langle$ to every call position and closing brackets $\rangle$ to the return positions in a word w, we group the word w into subwords. This grouping can be nested. However, not every bracket at a position in w needs to have a matching bracket. The call and return positions in a nested word without matching brackets are called *pending*. To emphasize this hierarchical structure imposed by the brackets $\langle$ and $\rangle$, we also refer to the words in $\Sigma^* \cup \Sigma^\omega$ as *nested words*. See Fig. 1 for illustration.

To ease the exposition, we restrict ourselves in the following to nested words without pending positions. Our results extend to nested words with pending positions; see [15]. For $\sharp \in \{*, \omega\}$, $NW^\sharp(\Sigma)$ denotes the set of words in $\Sigma^\sharp$ with no pending positions. These words are also called *well-matched*.

Automata. A *visibly pushdown automaton* [4], VPA for short, is a tuple $\mathcal{A} = (Q, \Gamma, \Sigma, \delta, q_I, \Omega)$, where Q is a finite set of states, Γ is a finite set of stack symbols, $\Sigma = \Sigma_{\mathsf{int}} \cup \Sigma_{\mathsf{call}} \cup \Sigma_{\mathsf{ret}}$ is the input alphabet, δ consists of three transition functions $\delta_{\mathsf{int}} : Q \times \Sigma_{\mathsf{int}} \to 2^Q$, $\delta_{\mathsf{call}} : Q \times \Sigma_{\mathsf{call}} \to 2^{Q \times \Gamma}$, and $\delta_{\mathsf{ret}} : Q \times \Gamma \times \Sigma_{\mathsf{ret}} \to 2^Q$, $q_I \in Q$ is the initial state, and $\Omega : Q \to \mathbb{N}$ is the priority function. Since we restrict ourselves here to well-matched words, we do not need to consider a bottom stack symbol $\perp$. We write $\Omega(Q)$ to denote the set of all priorities used in $\mathcal{A}$, i.e. $\Omega(Q) := \{\Omega(q) \mid q \in Q\}$. The *size* of $\mathcal{A}$ is $|Q|$ and its *index* is $|\Omega(Q)|$.

A *run* of $\mathcal{A}$ on $w \in \Sigma^\omega$ is a word $(q_0, \gamma_0)(q_1, \gamma_1) \dots \in (Q \times \Gamma^*)^\omega$ with $(q_0, \gamma_0) = (q_I, \varepsilon)$ and for each $i \in \mathbb{N}$, the following conditions hold:

1. If $w_i \in \Sigma_{\mathsf{int}}$ then $q_{i+1} \in \delta_{\mathsf{int}}(q_i, w_i)$ and $\gamma_{i+1} = \gamma_i$.
2. If $w_i \in \Sigma_{\mathsf{call}}$ then $(q_{i+1}, B) \in \delta_{\mathsf{call}}(q_i, w_i)$ and $\gamma_{i+1} = B\gamma_i$, for some $B \in \Gamma$.
3. If $w_i \in \Sigma_{\mathsf{ret}}$ and $\gamma_i = Bu$ with $B \in \Gamma$ and $u \in \Gamma^*$ then $q_{i+1} \in \delta_{\mathsf{ret}}(q_i, B, w_i)$ and $\gamma_{i+1} = u$.

The run is *accepting* if $\max\{\Omega(q) \mid q \in \inf(q_0 q_1 \dots)\}$ is even. Runs of $\mathcal{A}$ on finite words are defined as expected. In particular, a run on a finite word is accepting if the last state in the run has an even priority. For $\sharp \in \{*, \omega\}$, we define

$$L^\sharp(\mathcal{A}) := \big\{ w \in NW^\sharp(\Sigma) \,\big|\, \text{there is an accepting run of } \mathcal{A} \text{ on } w \big\} .$$

Priority and Reward Ordering. For an arbitrary set S, we always assume that $\dagger$ is a distinct element not occurring in S. We write $S_\dagger$ for $S \cup \{\dagger\}$. We use $\dagger$ to explicitly speak about partial functions into S, i.e., $\dagger$ denotes undefinedness.

We define the following two orders on $\mathbb{N}_\dagger$. The *priority ordering* is denoted $\sqsubseteq$ and is the standard order of type $\omega + 1$. Thus, we have $0 \sqsubset 1 \sqsubset 2 \sqsubset \cdots \sqsubset \dagger$. The *reward ordering* $\preceq$ is defined by $\dagger \prec \cdots \prec 5 \prec 3 \prec 1 \prec 0 \prec 2 \prec 4 \prec \cdots$. Note that $\dagger$ is maximal for $\sqsubseteq$ but minimal for $\preceq$. For a finite nonempty set $S \subseteq \mathbb{N}_\dagger$, $\bigsqcup S$ and $\bigcurlyvee S$ denote the maxima with respect to the priority ordering $\sqsubseteq$ and the reward ordering $\preceq$, respectively. Furthermore, we write $c \sqcup c'$ for $\bigsqcup\{c, c'\}$.

The reward ordering reflects the intuition of how valuable a priority of a VPA's state is for acceptance: even priorities are better than odd ones, and the bigger an even one is the better, while small odd priorities are better than bigger ones because it is easier to subsume them in a run with an even priority elsewhere. The element $\dagger$ stands for the non-existence of a run.

3 Universality Checking

Throughout this section, we fix a VPA $\mathcal{A} = (Q, \Gamma, \Sigma, \delta, q_I, \Omega)$. We describe an algorithm that determines whether $L^\omega(\mathcal{A}) = NW^\omega(\Sigma)$, i.e., whether $\mathcal{A}$ accepts all well-matched infinite nested words over Σ. An extension of the algorithm to account for non-well-matched nested words and a universality check for VPAs over finite nested words is given in [15]. Moreover, in [15], we present a complementation construction for VPAs based on determinization and compare it to the presented algorithm.

Central to the algorithm are so-called transition profiles. They capture $\mathcal{A}$'s essential behavior on finite words.

Definition 1. There are three kinds of *transition profiles*, TP for short. The first one is an int-TP, which is a function of type $Q \times Q \to \Omega(Q)_\dagger$. We associate with a symbol $a \in \Sigma_{\mathsf{int}}$ the int-TP f_a. It is defined as

$$f_a(q, q') \;:=\; \begin{cases} \Omega(q') & \text{if } q' \in \delta_{\mathsf{int}}(q, a) \text{ and} \\ \dagger & \text{otherwise.} \end{cases}$$

A call-TP is a function of type $Q \times \Gamma \times Q \to \Omega(Q)_\dagger$. With a symbol $a \in \Sigma_{\mathsf{call}}$ we associate the call-TP f_a. It is defined as

$$f_a(q, B, q') \;:=\; \begin{cases} \Omega(q') & \text{if } (q', B) \in \delta_{\mathsf{call}}(q, a) \text{ and} \\ \dagger & \text{otherwise.} \end{cases}$$

Finally, a ret-TP is a function of type $Q \times \Gamma \times Q \to \Omega(Q)_\dagger$. With a symbol $a \in \Sigma_{\mathsf{ret}}$ we associate the ret-TP f_a. It is defined as

$$f_a(q, B, q') \;:=\; \begin{cases} \Omega(q') & \text{if } q' \in \delta_{\mathsf{ret}}(q, B, a) \text{ and} \\ \dagger & \text{otherwise.} \end{cases}$$

A TP of the form f_a for an $a \in \Sigma$ is also called *atomic*. For $\tau \in \{\mathsf{int}, \mathsf{call}, \mathsf{ret}\}$, we define the set of atomic TPs as $T_\tau := \{f_a \mid a \in \Sigma_\tau\}$.

The above TPs describe $\mathcal{A}$'s behavior when $\mathcal{A}$ reads a single letter. In the following, we define how TPs can be composed to describe $\mathcal{A}$'s behavior on words of finite length. The composition, written $f \circ g$, can only be applied to TPs of certain kinds. This ensures that the resulting TP describes the behavior on a word w such that, after reading w, $\mathcal{A}$'s stack height has changed by at most one.

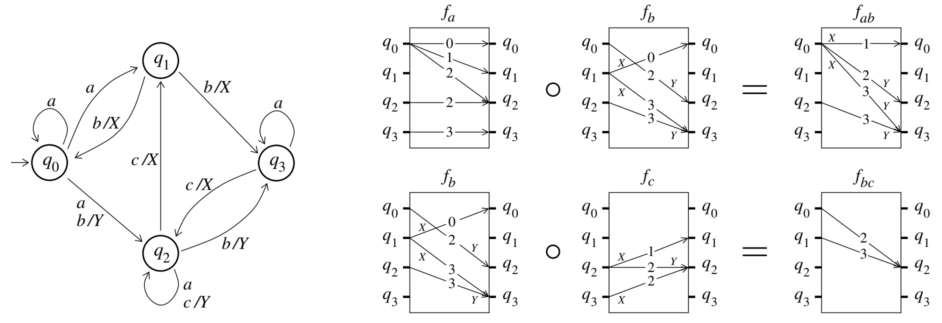

Figure 2. VPA (left) and the TPs (right) from Example 4.

Definition 2. Let f and g be TPs. There are six different kinds of compositions, depending on the TPs' kind of f and g, which we define in the following. If f and g are both int-TPs, we define

$$(f \circ g)(q,q') \;:=\; \bigvee \big\{ f(q,q'') \sqcup g(q'',q') \,\big|\, q'' \in Q \big\}\,.$$

If f is an int-TP and g is either a call-TP or a ret-TP, we define

$$(f \circ g)(q,B,q') \;:=\; \bigvee \big\{ f(q,q'') \sqcup g(q'',B,q') \,\big|\, q'' \in Q \big\} \qquad \text{and}$$
$$(g \circ f)(q,B,q') \;:=\; \bigvee \big\{ g(q,B,q'') \sqcup f(q'',q') \,\big|\, q'' \in Q \big\}\,.$$

If f is a call-TP and g a ret-TP, we define

$$(f \circ g)(q,q') \;:=\; \bigvee \big\{ f(q,B,q'') \sqcup g(q'',B,q') \,\big|\, q'' \in Q \text{ and } B \in \Gamma \big\}\,.$$

Intuitively, the composition of two TPs f and g is obtained by following any edge through f from some state q to another state q'', then following any edge through g to some other state q'. The value of this path is the maximum of the two values encountered in f and g with respect to the priority ordering $\sqsubseteq$. Then one takes the maximum over all such possible values with respect to the reward ordering $\preceq$ and obtains a weighted path from q to q' in the composition.

We associate finite words with TPs as follows. With a letter $a \in \Sigma$ we associate the TP f_a as done in Def. 1. If the words $u, v \in \Sigma^+$ are associated with the TPs f and g, respectively, we associate the word uv with the TP $f \circ g$, provided that $f \circ g$ is defined. A word cannot be associated with two distinct TPs. This follows from the following lemma, which is easy to prove.

Lemma 3. *Let f, g, h, and k be TPs. If $(h \circ f) \circ (g \circ k)$ and $h \circ \big((f \circ g) \circ k\big)$ are both defined then $(h \circ f) \circ (g \circ k) = h \circ \big((f \circ g) \circ k\big)$.*

If the word $u \in \Sigma^+$ is associated with the TP f, we write f_u for f. Note that two distinct words can be associated with the same TP, i.e., it can be the case that $f_u = f_v$, for $u, v \in \Sigma^+$ with $u \neq v$. Intuitively, if this is the case then $\mathcal{A}$'s behavior on u is identical to $\mathcal{A}$'s behavior on v.

The following example illustrates TPs and their composition.

Example 4. Consider the VPA on the left in Fig. 2 with the states q_0, q_1, q_2, and q_3. The states' priorities are the same as their indices. We assume that $\Sigma_{\mathsf{int}} = \{a\}$, $\Sigma_{\mathsf{call}} = \{b\}$, and $\Sigma_{\mathsf{ret}} = \{c\}$. The stack alphabet is $\Gamma = \{X, Y\}$.

Fig. 2 also depicts the TPs f_a, f_b, f_c and their compositions $f_a \circ f_b = f_{ab}$ and $f_b \circ f_c = f_{bc}$. The VPA's states are in-ports and out-ports of a TP. Assume that f is a call-TP. An in-port q is connected with an out-port q' if $f(q, B, q') \neq \dagger$, for some $B \in \Gamma$. Moreover, this connection of the two ports is labeled with the stack symbol B and the priority. The number of a connection between an in-port and an out-port specifies its priority. For example, the connection in the TP f_a from the in-port q_0 to the out-port q_0 has priority 0 since $f_a(q_0, q_0) = 0$. Since f_a is an int-TP, connections are not labeled with stack symbols.

In a composition $f \circ g$, we plug f's out-ports with g's in-ports together. The priority from an in-port of $f \circ g$ to an out-port of $f \circ g$ is the maximum with respect to the priority ordering $\sqsubseteq$ of the priorities of the two connections in f and g. However, if f is a call-TP and g a ret-TP, we are only allowed to connect the ports in $f \circ g$, if the stack symbols of the connections in f and g match. Finally, since there can be more than one connection between ports in $f \circ g$, we take the maximum with respect to reward ordering $\preceq$.

We extend the composition operation $\circ$ to sets of TPs in the natural way, i.e., we define $F \circ G := \{f \circ g \mid f \in F \text{ and } g \in G \text{ for which } f \circ g \text{ is defined}\}$.

Definition 5. Define $\mathfrak{T}$ as the least solution to the equation

$$\mathfrak{T} \;=\; T_{\mathsf{int}} \,\cup\, T_{\mathsf{call}} \circ T_{\mathsf{ret}} \,\cup\, T_{\mathsf{call}} \circ \mathfrak{T} \circ T_{\mathsf{ret}} \,\cup\, \mathfrak{T} \circ \mathfrak{T}\,.$$

Note that the operations $\circ$ and $\cup$ are monotonic, and the underlying lattice of the powerset of all int-TPs is finite. Thus, the least solution always exists and can be found using fixpoint iteration in a finite number of steps.

The following lemma is helpful in proving that the elements of $\mathfrak{T}$ can be used to characterize (non-)universality of $\mathcal{A}$.

Lemma 6. *For every TP f, we have $f \in \mathfrak{T}$ only if there is a well-matched $w \in \Sigma^+$ with $f = f_w$.*

We need the following notions to characterize universality in terms of the existence of TPs with certain properties.

Definition 7. Let f be an int-TP.

(i) f is *idempotent* if $f \circ f = f$. Note that only an int-TP can be idempotent.

(ii) For $q \in Q$, we write $f(q)$ for the set of all $q' \in Q$ that are connected to q in this TP, i.e., $f(q) := \{q' \in Q \mid f(q, q') \neq \dagger\}$. Moreover, for $Q' \subseteq Q$, we define $f(Q') := \bigcup_{q \in Q'} f(q)$.

(iii) f is *bad* for the set $Q' \subseteq Q$ if $f(q, q)$ is either $\dagger$ or odd, for every $q \in f(Q')$. A *good* TP is a TP that is not bad. Note that any TP is bad for $\emptyset$. In the following, we consider bad TPs only in the context of idempotent TPs.

Example 8. Reconsider the VPA from Example 4 and its TPs. It is easy to see that TP $g := f_a \circ f_a$ is idempotent. Since $g(q_2, q_2) = 2$, g is good for any

```
1 $N \leftarrow T_{\mathsf{int}} \cup T_{\mathsf{call}} \circ T_{\mathsf{ret}}$
2 $T \leftarrow N$
3 while $N \neq \emptyset$ do
4     forall $(f_u, f_v) \in N \times T \cup T \times N$ do
5         if $f_v$ idempotent and $f_v$ bad for $f_u(q_I)$ then
6             return universality does not hold, witnessed by $uv^\omega$
7     $N \leftarrow \big(N \circ T \cup T \circ N \cup T_{\mathsf{call}} \circ N \circ T_{\mathsf{ret}}\big) \setminus T$
8     $T \leftarrow T \cup N$
9 return universality holds
```

Figure 3. Universality check UNIV for VPAs with respect to well-matched words.

$Q' \subseteq \{q_0, q_1, q_2, q_3\}$ with $q_2 \in Q'$. The intuition is that there is at least one run on $(aa)^\omega$ that starts in q_2 and loops infinitely often through q_2. Moreover, on this run 2 is the highest priority that occurs infinitely often. So, if there is a prefix $v \in \Sigma^+$ with a run that starts in the initial state and ends in q_2, we have that $v(aa)^\omega$ is accepted by the VPA. The TP g is bad for $\{q_1, q_3\}$, since $g(q_1, q_1) = \dagger$ and $g(q_3, q_3) = 3$. So, if there is prefix $v \in \Sigma^+$ for which all runs that start in the initial state and end in q_1 or q_3 then $v(aa)^\omega$ is not accepted by the VPA. Another TP that is idempodent is the TP $g' := f_b \circ \big((f_b \circ f_c) \circ f_c\big)$. Here, we have that $g'(q_1, q_1) = 2$ and $g'(q, q') = \dagger$, for all $q, q' \in \{q_0, q_1, q_2, q_3\}$ with not $q = q' = q_1$. Thus, g' is bad for every $Q' \subseteq Q$ with $q_1 \notin Q'$.

The following theorem characterizes universality of the VPA $\mathcal{A}$ in terms of the TPs that are contained in the least solution of the equation from Def. 5.

Theorem 9. *$L^\omega(\mathcal{A}) \neq NW^\omega(\Sigma)$ iff there are TPs $f, g \in \mathfrak{T}$ such that g is idempotent and bad for $f(q_I)$.*

Thm. 9 can be used to decide universality for VPAs with respect to the set of well-matched infinite words. The resulting algorithm, which we name UNIV, is depicted in Fig. 3. It computes $\mathfrak{T}$ by least-fixpoint iteration and checks at each stage whether two TPs exist that witness non-universality according to Thm. 9. The variable T stores the generated TPs and the variable N stores the newly generated TPs in an iteration. UNIV terminates if no new TPs are generated in an iteration. Termination is guaranteed since there are only finitely many TPs. For returning a witness of the VPA's non-universality, we assume that we have a word associated with a TP at hand. UNIV's asymptotic time complexity is as follows, where we assume that we use hash tables to represent T and N.

Theorem 10. *Assume that the given VPA $\mathcal{A}$ has $n \geq 1$ states, index $k \geq 2$, and $m = \max\{1, |\Sigma|, |\Gamma|\}$, where Σ is the VPA's input alphabet and Γ its stack alphabet. The running time of the algorithm UNIV is in $m^3 \cdot 2^{\mathcal{O}(n^2 \cdot \log k)}$.*

There are various ways to tune UNIV. For instance, we can store the TPs in a single hash table and store pointers to the newly generated TPs. Furthermore, we can store pointers to idempotent TPs. Another optimization also concerns the badness check in the line 4 to 6. Observe that it is sufficient to know the sets $f_u(q_I)$, for $f_u \in T$, i.e, the sets $Q' \subseteq Q$ for which all runs for some well-matched word end in a state in Q'. We can maintain a set R to store this information. We initialize R with the singleton set $\{(\varepsilon, \{q_I\})\}$. We update it after line 8 in each iteration by assigning the set $R \cup \{(uv, f_v(Q')) \mid (u, Q') \in R \text{ and } f_v \in T\}$ to it. After this update, we can optimize R by removing an element (u, Q') from it if there is another element (u', Q'') in R with $Q'' \subseteq Q'$. These optimizations do not improve UNIV's worst-case complexity but they are of great practical value.

4 Inclusion Checking

In this section, we describe how to check language inclusion for VPAs. For the sake of simplicity, we assume a single VPA and check for inclusion of the languages that are defined by two states q_I^1 and q_I^2. It should be clear that it is always possible to reduce the case for two VPAs to this one by forming the disjoint union of the two VPAs. Thus, for $i \in \{1, 2\}$, let $\mathcal{A}_i = (Q, \Gamma, \Sigma, \delta, q_I^i, \Omega)$ be the respective VPA. We describe how to check whether $L^\omega(\mathcal{A}_1) \subseteq L^\omega(\mathcal{A}_2)$ holds.

Transition profiles for inclusion checking extend those for universality checking. A *tagged transition profile* (TTP) of the int-type is an element of

$$\big(Q \times \Omega(Q) \times Q\big) \times \big(Q \times Q \to \Omega(Q)_\dagger\big) .$$

We write it as $f^{\langle p,c,p' \rangle}$ instead of (p, c, p', f) in order to emphasize the fact that the TP f is extended with a tuple of states and priorities. A call-TTP is of type

$$\big(Q \times \Gamma \times \Omega(Q) \times Q\big) \times \big(Q \times \Gamma \times Q \to \Omega(Q)_\dagger\big)$$

and a ret-TTP is of type

$$\big(Q \times \Omega(Q) \times \Gamma \times Q\big) \times \big(Q \times \Gamma \times Q \to \Omega(Q)_\dagger\big) .$$

Accordingly, they are written $f^{\langle p,B,c,p' \rangle}$ and $f^{\langle p,c,B,p' \rangle}$, respectively.

The intuition of an int-TTP $f^{\langle p,c,p' \rangle}$ is as follows. The TP f describes the essential information of *all* runs of the VPA $\mathcal{A}_2$ on a well-matched word $u \in \Sigma^+$. The attached information $\langle p, c, p' \rangle$ describes the existence of *some* run of the VPA $\mathcal{A}_1$ on u. This run starts in state p, ends in state p', and the maximal occurring priority on it is c. The intuition behind a call-TTP or a ret-TTP is similar. The symbol B in the annotation is the topmost stack symbol that is pushed or popped in the run of $\mathcal{A}_2$ for the pending position in the word u.

For $a \in \Sigma$, we now associate a set F_a of TTPs with the appropriate type. Recall that f_a stands for the TP associated to the letter a as defined in Def. 1.

– If $a \in \Sigma_{\mathsf{int}}$, let $F_a := \{f_a^{\langle p,\Omega(p'),p' \rangle} \mid p, p' \in Q \text{ and } p' \in \delta_{\mathsf{int}}(p, a)\}$.
– If $a \in \Sigma_{\mathsf{call}}$, let $F_a := \{f_a^{\langle p,B,\Omega(p'),p' \rangle} \mid p, p' \in Q,\ B \in \Gamma, \text{ and } (p', B) \in \delta_{\mathsf{call}}(p, a)\}$.
– If $a \in \Sigma_{\mathsf{ret}}$, let $F_a := \{f_a^{\langle p,\Omega(p'),B,p' \rangle} \mid p, p' \in Q,\ B \in \Gamma, \text{ and } p' \in \delta_{\mathsf{ret}}(p, B, a)\}$.

As with TPs, the composition of TTPs is only allowed in certain cases. They are the same as for TPs, e.g., the composition of a call-TTP with an int-TTP results in a call-TTP, and with a ret-TTP it results in an int-TTP. However, the composition of TTPs is not a monoid operation but behaves like the composition of morphisms in a category in which the states in Q, respectively pairs of states and stack symbols in Γ, act as objects. A TTP $f^{\langle p,c,p' \rangle}$ for instance can be seen as a morphism from p to p', and it can therefore only be composed with a morphism from p' to anything else.

The composition of two TTPs extends the composition of the underlying TPs by explaining how the tag of the resulting TTP is obtained. For int-TTPs $f^{\langle p,c,p'\rangle}$ and $g^{\langle p',c',p''\rangle}$, we define

$$f^{\langle p,c,p'\rangle} \circ g^{\langle p',c',p''\rangle} \;:=\; (f \circ g)^{\langle p,c\sqcup c',p''\rangle}\,.$$

Composing an int-TTP $f^{\langle p,c,p'\rangle}$ and a call-TTP $g^{\langle q,B,c',q'\rangle}$ yields call-TTPs:

$$\begin{aligned} f^{\langle p,c,p'\rangle} \circ g^{\langle q,B,c',q'\rangle} &:= (f \circ g)^{\langle p,B,c\sqcup c',q'\rangle} && \text{if } p' = q \\ g^{\langle q,B,c',q'\rangle} \circ f^{\langle p,c,p'\rangle} &:= (g \circ f)^{\langle q,B,c\sqcup c',p'\rangle} && \text{if } q' = p\,. \end{aligned}$$

The two possible compositions of an int-TTP with a ret-TTP are defined in exactly the same way. Finally, the composition of a call-TTP $f^{\langle p,B,c,p'\rangle}$ and a ret-TTP $g^{\langle p',c',B,p''\rangle}$ is defined as

$$f^{\langle p,B,c,p'\rangle} \circ g^{\langle p',c',B,p''\rangle} \;:=\; (f \circ g)^{\langle p,c\sqcup c',p''\rangle}\,.$$

Note that the stack symbol B is the same in both annotations. As for sets of TPs, we extend the composition of TTPs to sets.

Similar to Def. 5, we define a set $\hat{\mathfrak{T}}$ to be the least solution to the equation

$$\hat{\mathfrak{T}} \;=\; \hat{T}_{\mathsf{int}} \;\cup\; \hat{T}_{\mathsf{call}} \circ \hat{T}_{\mathsf{ret}} \;\cup\; \hat{T}_{\mathsf{call}} \circ \hat{\mathfrak{T}} \circ \hat{T}_{\mathsf{ret}} \;\cup\; \hat{\mathfrak{T}} \circ \hat{\mathfrak{T}}\,,$$

where $\hat{T}_\tau := \bigcup\{F_a \mid a \in \Sigma_\tau\}$, for $\tau \in \{\mathsf{int}, \mathsf{call}, \mathsf{ret}\}$. This allows us to characterize language inclusion between two VPAs in terms of the existence of certain TTPs.

Theorem 11. *$L^\omega(\mathcal{A}_1) \not\subseteq L^\omega(\mathcal{A}_2)$ iff there are TTPs $f^{\langle q_I^1,c,p\rangle}$ and $g^{\langle p,d,p\rangle}$ in $\hat{\mathfrak{T}}$ fulfilling the following properties:*
(1) The priority d is even.
(2) The TP g is idempotent and bad for $f(q_I^2)$.

Thm. 11 yields an algorithm INCL to check $L^\omega(\mathcal{A}_1) \not\subseteq L^\omega(\mathcal{A}_2)$, for given VPAs $\mathcal{A}_1$ and $\mathcal{A}_2$. It is along the same lines as the algorithm UNIV and we omit it. The essential difference lies in the sets $\hat{T}_{\mathsf{int}}$, $\hat{T}_{\mathsf{call}}$, and $\hat{T}_{\mathsf{ret}}$, which contain TTPs instead of TPs, and the refined way in which they are being composed. Each iteration now searches for two TTPs that witness the existence of some word of the form uv^ω that is accepted by $\mathcal{A}_1$ but not accepted by $\mathcal{A}_2$. Similar optimizations that we sketch for UNIV at the end of Sect. 3 also apply to INCL.

For the complexity analysis of the algorithm INCL below, we do not assume that the VPAs $\mathcal{A}_1$ and $\mathcal{A}_2$ necessarily share the state set, the priority function, the stack alphabet, and the transition functions as assumed at the beginning of this subsection. Only the input alphabet Σ is the same for $\mathcal{A}_1$ and $\mathcal{A}_2$.

Theorem 12. *Assume that for $i \in \{1,2\}$, the number of states of the VPA $\mathcal{A}_i$ is $n_i \geq 1$, $k_i \geq 2$ its index, and $m_i = \max\{1, |\Sigma|, |\Gamma_i|\}$, where Σ is the VPA's input alphabet and Γ_i its stack alphabet. The running time of the algorithm INCL is in $n_1^4 \cdot k_1^2 \cdot m_1 \cdot m_2^3 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$.*

5 Evaluation

Our prototype tool FADecider implements the presented algorithms in the programming language OCaml.[2] To evaluate the tool's performance we carried

[2] The tool (version 0.4) is publicly available at `www2.tcs.ifi.lmu.de/fadecider`.

Table 1. Statistics on the input instances. The first row lists the number of states of the VPAs from an input instance and their alphabet sizes. The number of stack symbols of a VPA and its index are not listed, since in these examples the VPA's stack symbol set equals its state set and states are either accepting or non-accepting. The second row lists whether the inclusions $L^*(\mathcal{A}) \subseteq L^*(\mathcal{B})$ and $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$ of the respective VPAs hold.

	ex	*ex-§2.5*	*gzip*	*gzip-fix*	*png2ico*
size $\mathcal{A}$ / size $\mathcal{B}$ / alphabet size	9 / 5 / 4	10 / 5 / 5	51 / 71 / 4	51 / 73 / 4	22 / 26 / 5
language relation	$\subseteq$ / $\subseteq$	$\not\subseteq$ / $\subseteq$	$\not\subseteq$ / ?	$\subseteq$ / $\subseteq$	$\subseteq$ / $\subseteq$

Table 2. Experimental results for the language-inclusion checks. The row "FADecider" lists the running times for the tool FADecider for checking $L^*(\mathcal{A}) \subseteq L^*(\mathcal{B})$ and $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$. The row "#TTPs" lists the number of encountered TTPs. The symbol ‡ indicates that FADecider ran out of time (2 hours). The row "OpenNWA" lists the running times for the implementation based on the OpenNWA library for checking inclusion on finite words and the VPA's size obtained by complementing $\mathcal{B}$.

	ex	*ex-§2.5*	*gzip*	*gzip-fix*	*png2ico*
FADecider	0.00s / 0.00s	0.00s / 0.00s	36s / ‡	42s / 294s	0.10s / 0.11s
#TTPs	6 / 6	18 / 19	694 / ‡	518 / 1,117	586 / 609
OpenNWA	0.16s / 27	0.04s / 11	49s / 27	1,104s / 176	74.70s / 543

out the following experiments for which we used a 64-bit Linux machine with 4 GB of main memory and two dual-core Xeon 5110 CPUs, each with 1.6 GHz. Our benchmark suite consists of VPAs from [11], which are extracted from real-world recursive imperative programs. Tab. 1 describes the instances, each consisting of two VPAs $\mathcal{A}$ and $\mathcal{B}$, in more detail. Tab. 2 shows FADecider's running times for the inclusion checks $L^*(\mathcal{A}) \subseteq L^*(\mathcal{B})$ and $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$. For comparison, we used the OpenNWA library [12]. The inclusion check there is implemented by a reduction to an emptiness check via a complementation construction. Note that OpenNWA does not support infinite nested words at all and has no direct support for only considering well-matched nested words. We used therefore OpenNWA to perform the language-inclusion checks with respect to all finite nested words.

FADecider outperforms OpenNWA on these examples. Profiling the inclusion check based on the OpenNWA library yields that complementation requires about 90% of the overall running time. FADecider spends about 90% of its time on composing TPs and about 5% on checking equality of TPs. The experiments also show that FADecider's performance on inclusion checks for infinite words can be worse than for finite words. Note that checking inclusion for infinite-word languages is more expensive than for finite-word languages, since, in addition to reachability, one needs to account for loops.

6 Conclusion

Checking universality and language inclusion for automata by avoiding determinization and complementation has recently attracted a lot of attention, see, e.g., [1, 9, 10, 13, 16]. We have shown that Ramsey-based methods for Büchi automata generalize to the richer automaton model of VPAs with a parity acceptance condition. Another competitive approach based on antichains has recently also been extended to VPAs, however, only to VPAs over finite words [6]. It remains to be seen if optimizations for the Ramsey-based algorithms for Büchi automata [1] extend, with similar speed-ups, to this richer setting. Another direction of future work is to investigate Ramsey-based approaches for automaton models that extend VPAs like multi-stack VPAs [18].

Acknowledgments. We are grateful to Evan Driscoll for providing us with VPAs.

References

1. P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. Advanced Ramsey-based Büchi automata inclusion testing. In *CONCUR'11*, LNCS 6901, pp. 187–202.
2. P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *TACAS'10*, LNCS 6015, pp. 158–174.
3. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Progr. Lang. Syst.*, 27(4):786–818, 2005.
4. R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):1–43, 2009.
5. S. Breuers, C. Löding, and J. Olschewski. Improved Ramsey-based Büchi complementation. In *FOSSACS'12*, LNCS 7213, pp. 150–164.
6. V. Bruyère, M. Ducobu, and O. Gauwin. Visibly pushdown automata: universality and inclusion via antichains. In *LATA'13*, LNCS 7810, pp. 190–201.
7. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. of the 1960 Internat. Congr. on Logic, Method, and Philosophy of Science*, pp. 1–11.
8. C. Dax, M. Hofmann, and M. Lange. A proof system for the linear time μ-calculus. In *FSTTCS'06*, LNCS 4337, pp. 273–284.
9. M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV'06*, LNCS 4144, pp. 17–30.
10. L. Doyen and J.-F. Raskin. Antichains for the automata-based approach to model-checking. *Log. Methods Comput. Sci.*, 5(1), 2009.
11. E. Driscoll, A. Burton, and T. Reps. Checking conformance of a producer and a consumer. In *ESEC/FSE'11*, pp. 113–123.
12. E. Driscoll, A. Thakur, and T. Reps. OpenNWA: A nested-word-automaton library. In *CAV'12*, LNCS 7358, pp. 665–671.
13. S. Fogarty and M. Y. Vardi. Büchi complementation and size-change termination. In *TACAS'09*, LNCS 5505, pp. 16–30.
14. S. Fogarty and M. Y. Vardi. Efficient Büchi universality checking. In *TACAS'10*, LNCS 6015, pp. 205–220.
15. O. Friedmann F. Klaedtke, and M. Lange. Ramsey goes visibly pushdown. Manuscript, 2012. Available at authors' web pages.
16. O. Friedmann and M. Lange. Ramsey-based analysis of parity automata. In *TACAS'12*, LNCS 7214, pp. 64–78.
17. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL'10*, pp. 471–482.
18. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS'07*, pp. 161–170.
19. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'01*, pp. 81–92.
20. F. P. Ramsey. On a problem of formal logic. *Proc. London Math. Soc.*, 30:264–286, 1928.
21. M.-H. Tsai, S. Fogarty, M. Y. Vardi, and Y.-K. Tsay. State of Büchi complementation. In *CIAA'10*, LNCS 6482, pp. 261–271.
22. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS'86*, pp. 332–344.