Ramsey-Based Inclusion Checking for Visibly Pushdown Automata

OLIVER FRIEDMANN, Ludwig-Maximilians-Universität München
FELIX KLAEDTKE, NEC Europe Ltd.
MARTIN LANGE, Universität Kassel

Checking whether one formal language is included in another is important in many verification tasks. In this article, we provide solutions for checking the inclusion of 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, for example, to reason about the nesting of procedure calls in the executions of recursive imperative programs. The presented solutions do not rely on explicit automaton constructions for determinization and complementation. Instead, they are more direct and generalize the so-called Ramsey-based inclusion-checking algorithms, which apply to classical finite-state automata and proved to be effective there to visibly pushdown automata. We also experimentally evaluate these algorithms, demonstrating the virtues of avoiding explicit determinization and complementation constructions.

Categories and Subject Descriptors: F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Decision problems*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Automata over finite and infinite words, visibly pushdown languages, nested words, decision problems, verification

ACM Reference Format:
Oliver Friedmann, Felix Klaedtke, and Martin Lange. 2015. Ramsey-based inclusion checking for visibly pushdown automata. ACM Trans. Comput. Logic 16, 4, Article 34 (August 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2774221

1. INTRODUCTION

Various tasks in system verification 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 nonterminating, finite-state systems with respect to trace properties boils down to the question of 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 $\mathcal{B}$

This work was partly done while the second author was at ETH Zurich. Preliminary results of the work were presented at the 40th International Colloquium on Automata, Languages and Programming (ICALP 2013); see Friedmann et al. [2013].
The European Research Council provided financial support under the European Community's 7th Framework Programme (FP7/2007-2013)/ERC grant agreement no. 259267.
Authors' addresses: O. Friedmann, Ludwig-Maximilians-Universität München, Institut für Informatik, Lehr- und Forschungseinheit für Theoretische Informatik, Oettingerstraße 67, 80538 Munich, Germany; email: oliver.friedmann@ifi.lmu.de; F. Klaedtke, NEC Europe Ltd., Kurfürsten-Anlage 36, 69115 Heidelberg, Germany; email: felix.klaedtke@neclab.eu; M. Lange, Universität Kassel, FB Elektrotechnik/Informatik, Wilhelmshöher Allee 71, 34121 Kassel, Germany; email: martin.lange@uni-kassel.de.

describes the property [Vardi and Wolper 1986]. Inclusion checks of the languages given by Büchi automata also appear in the domain of the analysis of program termination [Fogarty and Vardi 2009; Lee et al. 2001]. Inclusion problems are in general difficult. For Büchi automata, the problem is PSPACE-complete [Sistla et al. 1987].

From the closure properties of the class of ω-regular languages, that is, those languages that are recognizable by Büchi automata, it is obvious that questions such as the one posed earlier for model checking nonterminating, 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 [Choueka 1974], the emptiness problem is NLOGSPACE-complete [Vardi and Wolper 1994], and it admits efficient implementations, for example, by a nested depth-first search [Gerth et al. 1996]. However, complementing Büchi automata is challenging [Vardi 2007]. One intuitive reason for this is that not every Büchi automaton has an equivalent deterministic counterpart. Switching to a richer acceptance condition, such as the parity condition, so that determinization would be possible (see Piterman [2007], Kähler and Wilke [2008], and Schewe [2009]) is currently not an option in practice. The known determinization constructions for richer acceptance conditions are intricate, although complementation would then be easy by dualizing the acceptance condition [Muller and Schupp 1987]. A lower bound on the complementation problem with respect to the automaton size is $2^{\Omega(n \log n)}$ [Michel 1988]. 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, for instance, Tsai et al. [2011] and Breuers et al. [2012].

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 [Abdulla et al. 2011, 2010; Dax et al. 2006; De Wulf et al. 2006; Doyen and Raskin 2009; Fogarty and Vardi 2009, 2010; Lee et al. 2001]. 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 [Ramsey 1928].[1]

The Ramsey-based algorithms for checking universality $L(\mathcal{B}) = \Sigma^\omega$, where $\mathcal{B}$ is a Büchi automaton, iteratively build a set of finite graphs starting from a finite base set and close 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 if and only if this set contains a pair of graphs with certain properties that witness the existence of an infinite word of the form uv^ω 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 automaton's runs on the finite word v loop. We must also require that no accepting state occurs on these loops. Second, there must be another graph that describes all the automaton's runs on the finite word u that reach some of the loops of

[1]Büchi's original complementation construction [Büchi 1962], 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 [Abdulla et al. 2011; Breuers et al. 2012].

the first graph from the automaton's initial state. 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 $L(\mathcal{A}) \not\subseteq L(\mathcal{B})$, the constructed set 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 [Friedmann and Lange 2012]. Among all the usual "stronger" acceptance conditions, such as Rabin, Streett, and Muller, the parity acceptance condition provides a very good balance between expressive power and algorithmic feasibility. While translations from such conditions into a parity condition are in general exponential [Dziembowski et al. 1997], the parity condition has the distinct algorithmic advantage of memory-less determinacy [Emerson and Jutla 1991], which Streett and Muller do not enjoy. In the setting of automata theory, memoryless determinacy yields short witnesses of nonemptiness and nonuniversality, since it allows two successive visits of a state to be composed to a loop. Rabin automata enjoy this property only for witnesses of nonemptiness, not for nonuniversality. Parity automata are the only ones besides Büchi automata that have short—in this sense—and thus easier-to-find witnesses of both nonemptiness and nonuniversality.

Parity automata can be translated into Büchi automata at a blow-up that is linear in the number of priorities used in the parity automaton [Löding and Thomas 2000]. Hence, there is no gain in expressiveness in using parity automata over Büchi automata. There is, however, a gain in practicality. It is, for example, easy to express strong fairness conditions of the form "if p holds infinitely often then q holds infinitely often" as a parity condition with three priorities [Fritz and Wilke 2005]. Expressing the same with a Büchi condition would require a blow-up of the underlying automaton's state space. It was also shown that it pays off in terms of complexity to handle parity automata directly in the framework of Ramsey-based automata analysis [Friedmann and Lange 2012]. The algorithms for universality and inclusion checking for finite-state automata are polynomial in the number of priorities when dealing directly with parity automata but exponential when translating them into Büchi automata first.

In this article, we extend the Ramsey-based analysis to visibly pushdown automata (VPAs) [Alur and Madhusudan 2009]. 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. As a consequence, the stack heights are identical at the same positions in every run of any VPA on a given input. It is because of this syntactic restriction that the class of visibly pushdown languages retains many closure properties such as 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. Nonregular properties such as "an acquired lock must be released within the same procedure" are expressible by VPAs. Model checking of recursive state machines [Alur et al. 2005] and Boolean programs [Ball and Rajamani 2000], 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 [Vardi and Wolper 1986], checking the inclusion of the languages of VPAs is crucial here. This time, the respective decision problem is even EXPTIME-complete [Alur and Madhusudan 2009]. Other applications for checking language inclusion of VPAs when reasoning about recursive imperative programs also appear in conformance checking [Driscoll et al. 2011] and in the counterexample-guided-abstraction-refinement loop [Heizmann et al. 2010].

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 considering only those 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 of composing graphs in the Ramsey-based approach to finite-state automata over infinite words. Then the composition operation needs to account only 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 article 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 explicit 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. The evaluation demonstrates that the presented algorithms for inclusion checking are more efficient than methods that are based on determinization and complementation constructions.

The remainder of this article is organized as follows. In Section 2, we recall the framework of VPAs. In Section 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 Section 5, we extend the universality check to a Ramsey-based inclusion check for VPAs. Section 4 is an intermezzo, in which we describe variants of the universality check and compare it to complementation and determinization constructions. In Section 6, we report on the experimental evaluation of our algorithms. Finally, in Section 7, we draw conclusions.

2. PRELIMINARIES

In this section, we present and explain the notation and terminology that we use in the remainder of the text.

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 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, $w = w_0w_1\dots$ if w is infinite and $w = w_0w_1\dots w_{n-1}$ if w is finite and $|w| = n$. With $\mathrm{inf}(w)$, we denote the set of letters of Σ that occur infinitely often in $w \in \Sigma^\omega$.

Nested words [Alur and Madhusudan 2009] 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 $w_i \in \Sigma_{\mathsf{ret}}$. This partitioning of the alphabet's letters determines a matching relation between a word's call positions and its return positions. It is defined as follows. For a word $w \in \Sigma^* \cup \Sigma^\omega$, let $\curvearrowright \subseteq \mathbb{N} \times \mathbb{N}$ be a relation that satisfies the following properties, for every $i \curvearrowright j$.

(i) $0 \le i < j < |w|$, $w_i \in \Sigma_{\mathsf{call}}$, and $w_j \in \Sigma_{\mathsf{ret}}$.
(ii) $|\{k \mid i \curvearrowright k\}| \le 1$ and $|\{k \mid k \curvearrowright j\}| \le 1$.
(iii) There are no i', j' with $i' \curvearrowright j'$ and $i < i' < j < j'$.
A call position i in w for which there is no j with $i \curvearrowright j$ is called *pending*. The pending return positions in w are analogously defined. We call the relation $\curvearrowright$ a *matching relation* if it additionally satisfies the following properties, for all pending positions k, with $0 \le k < |w|$.

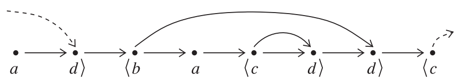

Fig. 1. Nested word *adbacddbc*.

(iv) There are no i, j with $i \curvearrowright j$ and $i < k < j$.
(v) If $w_k \in \Sigma_{\mathsf{call}}$, then there is no pending return position j with $k < j$.
(vi) If $w_k \in \Sigma_{\mathsf{ret}}$, then there is no pending call position i with $i < k$.

A matching relation is unique for a word. It structures a word and can be visualized when attaching an opening bracket "⟨" to every call position and a closing bracket "⟩" to every return position in w. This way, we group the word into subwords for which opening and closing brackets match. This grouping can be nested. However, not every bracket at a position in w needs to have a matching bracket. The pending call and return positions in a word are the positions without matching brackets. To emphasize this hierarchical structure imposed by the matching relation and the brackets "⟨" and "⟩," we also refer to the words in $\Sigma^* \cup \Sigma^\omega$ as *nested words*. See the following example for illustration.

Example 2.1. Figure 1 depicts the hierarchical structure of the word $w = adbacddbc$, where the alphabet is $\Sigma_{\mathsf{int}} = \{a\}$, $\Sigma_{\mathsf{call}} = \{b, c\}$, and $\Sigma_{\mathsf{ret}} = \{d\}$. The word's 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. That is, the word's matching relation $\curvearrowright$ is $\{(2, 6), (4, 5)\}$.

We consider the following four sets of infinite words.

—$NW_{\mathsf{match}}(\Sigma)$ is the set of words in Σ^ω that do not contain pending positions. We call these words *well-matched*.
—$NW_{\mathsf{call}}(\Sigma)$ is the set of words in Σ^ω that may contain pending call positions but no pending return positions.
—$NW_{\mathsf{ret}}(\Sigma)$ is the set of words in Σ^ω that may contain pending return positions but no pending call positions.
—$NW_{\mathsf{any}}(\Sigma)$ is the set of words in Σ^ω that may contain pending call positions and pending return positions. Note that $NW_{\mathsf{any}}(\Sigma) = \Sigma^\omega$.

For program-verification purposes, the two sets $NW_{\mathsf{match}}(\Sigma)$ and $NW_{\mathsf{call}}(\Sigma)$ are certainly of most interest. For instance, $NW_{\mathsf{match}}(\Sigma)$ can be used to describe traces of recursive imperative programs in which every call eventually terminates and there is a topmost procedure that runs forever. Similarly, the set $NW_{\mathsf{call}}(\Sigma)$ can be used to describe program traces in which subprocedures may not terminate. The sets $NW_{\mathsf{ret}}(\Sigma)$ and $NW_{\mathsf{any}}(\Sigma)$ are included here because they are not more difficult to handle, and $NW_{\mathsf{any}}(\Sigma)$ may well be useful in specifications about correct call-and-return behavior, that is, when one wants to *assert* rather than *assume* that no return is possible without a corresponding call beforehand. Furthermore, the call–return dualism need not only be used to describe recursive imperative programs but also programs using data structures like stacks or lists. In that case, a pending return position may correspond to a faulty access to the data structure, and it may therefore well be reasonable to account for pending returns.

Automata. A VPA [Alur and Madhusudan 2009] is a nondeterministic pushdown automaton that pushes a stack symbol when reading a letter in Σ_{call}, pops a stack symbol when reading a letter in Σ_{ret} (in the case that it is not the bottom stack symbol), and does not use its stack when reading a letter in Σ_{int} (see also Mehlhorn [1980]).

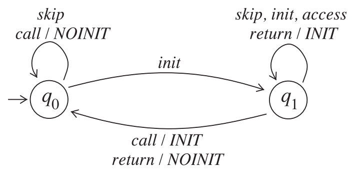

Fig. 2. Visibly pushdown automaton.

Formally, a *VPA* $\mathcal{A}$ is a tuple $(Q, \Gamma, \Sigma, \delta, q_I, \Omega)$, where Q is a finite set of states, Γ is a finite set of stack symbols with $\bot \notin \Gamma$, $\Sigma = \Sigma_{\mathsf{int}} \cup \Sigma_{\mathsf{call}} \cup \Sigma_{\mathsf{ret}}$ 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 $\delta_{\mathsf{ret}} : Q \times (\Gamma \cup \{\bot\}) \times \Sigma_{\mathsf{ret}} \to 2^Q$, $q_I \in Q$ is the initial state, and $\Omega : Q \to \mathbb{N}$ is the priority function. We sometimes write $\Gamma_\bot$ as a short form for $\Gamma \cup \{\bot\}$. We write $\Omega(Q)$ to denote the set of all priorities used in $\mathcal{A}$, that is, $\Omega(Q) := \{\Omega(q) \mid q \in Q\}$. 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 $(q_0, \gamma_0)(q_1, \gamma_1)\ldots \in (Q \times \Gamma_\bot^+)^\omega$ with $(q_0, \gamma_0) = (q_I, \bot)$ 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 $B \in \Gamma_\bot$ and $u \in \Gamma_\bot^*$, then $q_{i+1} \in \delta_{\mathsf{ret}}(q_i, B, w_i)$ and $\gamma_{i+1} = u$ if $u \neq \varepsilon$ and $\gamma_{i+1} = \bot$, otherwise.

Runs of $\mathcal{A}$ on finite words are defined as expected. Note that the bottom stack symbol $\bot$ is only read at a pending return position. The bottom stack symbol $\bot$ is therefore not needed for well-matched words. The run is *accepting* if $\max\{\Omega(q) \mid q \in \inf(q_0q_1\ldots)\}$ is even. For $t \in \{\mathsf{match}, \mathsf{call}, \mathsf{ret}, \mathsf{any}\}$, we define

$$L_t(\mathcal{A}) := \left\{w \in NW_t(\Sigma) \,\middle|\, \text{there is an accepting run of } \mathcal{A} \text{ on } w\right\}.$$

For the sake of brevity, we omit the subscript match. For instance, we write $NW(\Sigma)$ instead of $NW_{\mathsf{match}}(\Sigma)$ and $L(\mathcal{A})$ instead of $L_{\mathsf{match}}(\mathcal{A})$.

Example 2.2. The language of the VPA depicted in Figure 2 describes the property that local variables of a procedure must be initialized before reading from or writing to them. We use the alphabet with $\Sigma_{\mathsf{int}} = \{init, access, skip\}$, $\Sigma_{\mathsf{call}} = \{call\}$, and $\Sigma_{\mathsf{ret}} = \{return\}$. The letters *call* and *return* have the obvious meaning. The letter *init* corresponds to the action of initializing a procedure's local variables, *access* corresponds to the action of reading or writing from the local variables, and *skip* describes any other action.

The state q_0, which is the initial state, represents the fact that the local variables of the procedure that we are currently executing have not yet been initialized. Its counterpart is the state q_1, where the local variables are initialized. Both states have an even priority. In state q_0, we must not read from or write to the local variables. Hence, there is no outgoing transition for the letter *access*. When the local variables become initialized, we switch to state q_1 via the letter *init*. Here, reading from and writing to the local variables is allowed, that is, we loop with the letter *access*. Initializing the local variables again has no effect, that is, we stay in state q_1. Whenever we call a procedure, we reset the status of the local variables, that is, we go to state q_0. Additionally, we use the VPA's stack to store the status of the calling procedure by pushing either *INIT* or *NOINIT* onto the stack. When returning to a procedure, we restore this status.

Note that this property cannot be described by a finite-state automaton, since there is no limit on the number of procedures that can be called.

Priority and Reward Ordering. For an arbitrary set S, we always assume that $\dagger$ is an element not occurring in S. We write $S_\dagger$ for $S \cup \{\dagger\}$. We use $\dagger$ to explicitly speak about partial functions into S, that is, $\dagger$ 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 $\omega + 1$, that is, $0 \sqsubset 1 \sqsubset 2 \sqsubset \cdots \sqsubset \dagger$. 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 nonexistence of a run. This intuition about the reward ordering $\preceq$ is reflected in the following lemma. Its proof is straightforward and therefore omitted. Its consequence is more important: if there are two runs on the same word such that the priorities of the first one are always at most as large as the corresponding ones in the second run with respect to $\preceq$, then the second run is accepting if the first one is.

LEMMA 2.3. *Let $\rho, \rho' \in C^\omega$, where $C \subseteq \mathbb{N}$ is some finite set of priorities. Suppose that $\rho_i \preceq \rho'_i$, for all $i \in \mathbb{N}$. Then $\bigsqcup \inf(\rho) \preceq \bigsqcup \inf(\rho')$.*

The following lemma states that, using the priority ordering $\sqsubseteq$, one can contract an infinite run of a VPA while preserving the maximal priority occurring infinitely often. Its proof is also straightforward and omitted.

LEMMA 2.4. *Let $\rho \in C^\omega$, where $C \subseteq \mathbb{N}$ is some finite set of priorities. Take any strictly increasing sequence $i_0 < i_1 < \ldots$ of natural numbers and consider $\rho' \in C^\omega$ with $\rho'_j := \bigsqcup\{\rho_{i_j}, \ldots, \rho_{i_{j+1}-1}\}$, for $j \in \mathbb{N}$. We have that $\bigsqcup \inf(\rho) = \bigsqcup \inf(\rho')$.*

3. UNIVERSALITY CHECKING

Throughout this section, we fix a VPA $\mathcal{A}$ with $\mathcal{A} = (Q, \Gamma, \Sigma, \delta, q_I, \Omega)$. We describe algorithms that determine whether $\mathcal{A}$ is universal. We first restrict ourselves to well-matched infinite words (Section 3.1 and Section 3.2). That is, we provide an algorithm that checks for a given VPA $\mathcal{A}$ whether $L(\mathcal{A}) = NW(\Sigma)$ holds. We also generalize our approach to infinite words with pending positions (Section 3.3). Afterwards, in Section 4, we present variants of these algorithms, for instance, checking universality of VPAs over finite words, and also present a complementation construction for VPAs based on determinization and compare it to the presented algorithms. In Section 5, we finally extend our algorithms for checking language inclusion.

3.1. Transition Profiles

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

Definition 3.1. 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

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

Table I. Compositions of Transition Profiles

f \ g	call	int	ret
call	–	call	int
int	call	int	ret
ret	–	ret	–

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 $Q \times \Gamma_\perp \times Q \to \Omega(Q)_\dagger$. 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 $T_\tau := \{f_a \mid a \in \Sigma_\tau\}$.

These 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 as $f \circ g$, can be applied only 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.

Definition 3.2. 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. See Table I, which shows when the composition of f and g is defined and the kind of the resulting TP $f \circ g$. If f and g are both int-TPs, we define

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

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

$$(f \circ g)(q, B, q') := \bigvee \left\{ f(q, q'') \sqcup g(q'', B, q') \,\middle|\, q'' \in Q \right\}$$

and

$$(g \circ f)(q, B, q') := \bigvee \left\{ g(q, B, q'') \sqcup f(q'', q') \,\middle|\, q'' \in Q \right\}.$$

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

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

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 in the following. With a letter $a \in \Sigma$, we associate the TP f_a as already done in Definition 3.1. One expects that if the words $u, v \in \Sigma^+$ are associated with the TPs f and g, respectively, then the TP $f \circ g$ is associated to the word uv, provided that $f \circ g$ is defined. However, since words can be factorized in multiple ways, we need to convince ourselves that a word cannot be associated with two distinct TPs. To this end, we first observe that the $\circ$ is associative whenever the compositions are defined.

LEMMA 3.3. *Let f, g, and h be TPs. If $(f \circ g) \circ h$ and $f \circ (g \circ h)$ are both defined, then $(f \circ g) \circ h = f \circ (g \circ h)$.*

For illustration, consider the well-matched word *aabbab* with $a \in \Sigma_{\mathsf{call}}$ and $b \in \Sigma_{\mathsf{ret}}$. Even when respecting the order of the letters in this word, there are multiple ways to compose the TPs associated to the individual letters. It turns out that all these compositions, if defined, yield the same TP. For example, using Lemma 3.3, we have that

$$\begin{aligned} f_a \circ ((f_a \circ f_b) \circ (f_a \circ (f_a \circ f_b))) &= f_a \circ (((f_a \circ f_b) \circ f_a) \circ (f_a \circ f_b)) \\ &= (f_a \circ ((f_a \circ f_b) \circ f_a)) \circ (f_a \circ f_b) \\ &= ((f_a \circ (f_a \circ f_b)) \circ f_a) \circ (f_a \circ f_b) . \end{aligned}$$

However, note that, for example, $((((f_a \circ f_a) \circ f_b) \circ f_a) \circ f_a) \circ f_b$ is not defined, since f_a is a call-TP and already $f_a \circ f_a$ is not defined.

The next lemma shows that we can indeed associate finite words with TPs. (i) For $a \in \Sigma$, we define $f_{(a)} := \{f_a\}$, and (ii) for $w \in \Sigma^+$ with $|w| > 1$, we define $f_{(w)} := \{f \circ g \mid f \circ g \text{ is defined, and } f \in f_{(u)} \text{ and } g \in f_{(v)}, \text{ for } u, v \in \Sigma^+ \text{ with } uv = w\}$. If $f_{(w)}$ is a singleton, we denote the element in $f_{(w)}$ by f_w.

LEMMA 3.4. *Let $w \in \Sigma^+$. If $f_{(w)}$ has more than one pending position, then $f_{(w)}$ is the empty set. If w has at most one pending position, then $f_{(w)}$ is a singleton. More specifically: if w has no pending positions, then f_w is an* int*-TP; if w has one pending call position, then f_w is a* call*-TP; and if w has one pending return position, then f_w is a* ret*-TP.*

PROOF. We prove the lemma by induction over the length of w. The base case $|w| = 1$ is obvious. For the step case, assume that $|w| > 1$. If w has more than one pending position, it is easy to see, using the induction hypothesis, that we cannot find words $u, v \in \Sigma^+$ such that $w = uv$ for which $f_{(u)}$ and $f_{(v)}$ are singletons, and $f_u \circ f_v$ is defined. Hence, $f_{(w)}$ is the empty set.

For the remainder of the proof, assume that w has at most one pending position. Let $u, v, u', v' \in \Sigma^+$ be words with $w = uv = u'v'$ and $|u| < |u'|$. Furthermore, assume that each of these words has at most one pending position, and $f_u \circ f_v$ and $f_{u'} \circ f_{v'}$ are defined. It suffices to show that $f_u \circ f_v = f_{u'} \circ f_{v'}$.

Since $|u| < |u'|$, there is a word $x \in \Sigma^+$ such that $ux = u'$ and $xv' = v$. By inspecting all the different cases, it is not hard to see that the word x has at most one pending position. For example, if u has one pending call position and u' has no pending positions, then x has a pending return position. From the induction hypothesis, it follows that $f_{(x)}$ is a singleton. Furthermore, we obtain from the induction hypothesis the kind of the TP f_x. It is easy to check that $f_u \circ f_x$ and $f_x \circ f_{v'}$ are both defined, and, again by the induction hypothesis, $f_{u'} = f_u \circ f_x$ and $f_v = f_x \circ f_{v'}$. With Lemma 3.3, we conclude that $f_u \circ f_v = f_u \circ (f_x \circ f_{v'}) = (f_u \circ f_x) \circ f_{v'} = f_{u'} \circ f_{v'}$. □

Note that two distinct words can be associated with the same TP, that is, 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 3.5. Consider the VPA on the left in Figure 3 with the states q_0, q_1, q_2, and q_3. The states' priorities are the same as their indices. We assume that $\Sigma_{\mathsf{int}} = \{a\}$,

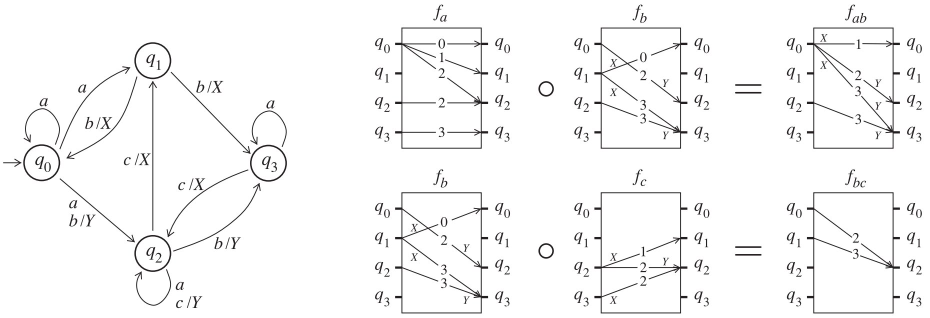

Fig. 3. VPA (left) and the TPs (right) from Example 3.5.

$\Sigma_{\text{call}} = \{b\}$, and $\Sigma_{\text{ret}} = \{c\}$. The stack alphabet is $\Gamma = \{X, Y\}$. We can ignore the stack symbol $\perp$ since the VPA has no transitions for c and $\perp$.

Figure 3 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, that is, 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 3.6. Define $\mathfrak{T}$ as the least solution to the equation

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

Note that the operations $\circ$ and $\cup$ are monotonic for inclusion ordering, 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 3.7. *Let f be a TP. We have $f \in \mathfrak{T}$ if and only if there is a well-matched word $w \in \Sigma^+$ with $f = f_w$.*

PROOF. If w is a well-matched word, then a straightforward induction over the length of w shows that $f_w \in \mathfrak{T}$. Recall Lemma 3.4, from which follows that f_w is defined. If $f \in \mathfrak{T}$, then the existence of a word $w \in \Sigma^+$ with $f_w = f$ is an immediate consequence of the fact that $\mathfrak{T}$ is the least solution of the equation. A simple induction on the length of words shows that $\mathfrak{T}$ contains only well-matched words. □

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

Definition 3.8. Let f be an int-TP.

(1) f is *idempotent* if $f \circ f = f$. Note that only an int-TP can be idempotent.
(2) For $q \in Q$, we write $f(q)$ for the set of all $q' \in Q$ that are connected to q in this TP, that is, $f(q) := \{q' \in Q \mid f(q, q') \neq \dagger\}$. Moreover, for $Q' \subseteq Q$, we define $f(Q') := \bigcup_{q \in Q'} f(q)$.
(3) 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 3.9. Reconsider the VPA from Example 3.5 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 $Q' \subseteq \{q_0, q_1, q_2, q_3\}$ 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 $g(q_3, q_3) = 3$. Thus, if there is prefix $v \in \Sigma^+$ for which all runs that start in the initial state and end either in q_1 or q_3, then $v(aa)^\omega$ is not accepted by the VPA. Another TP that is idempotent 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 Definition 3.6.

THEOREM 3.10. *$L(\mathcal{A}) \neq NW(\Sigma)$ if and only if there are TPs $f, g \in \mathfrak{T}$ such that g is idempotent and bad for $f(q_I)$.*

PROOF. "$\Leftarrow$" Let f and g be TPs in $\mathfrak{T}$ with g idempotent and bad for $f(q_I)$. By Lemma 3.7, there are $u, v \in \Sigma^+$ such that $f = f_u$ and $g = f_v$ and both u and v contain no pending positions. Then uv^ω contains no pending positions either. It remains to be seen that $uv^\omega \notin L(\mathcal{A})$.

For the sake of contradiction, assume that $w := uv^\omega \in L(\mathcal{A})$. Thus, there is an accepting run $\rho = (q_0, \gamma_0)(q_1, \gamma_1)\ldots$ of $\mathcal{A}$ on w such that $q_0 = q_I$.

Since $f = f_u$, we have $f(q_0, q_{|u|}) \neq \dagger$. Hence, $q_{|u|} \in f(q_0)$. Similarly, we conclude that $q_{|uv|} \in g(q_{|u|})$. This can be iterated to show that $g(q_{|u|+i|v|}, q_{|u|+(i+1)|v|}) \neq \dagger$ for all $i \geq 1$. Since Q is finite, there is a state $q \in Q$ such that $q = q_{|u|+i|v|}$ for infinitely many i. Assume that $i_0 < i_1 < \cdots$ is such a sequence of indices. Define a sequence $c_0, c_1, \ldots$ by $c_j := \bigsqcup\{\Omega(q_{i_j}), \ldots, \Omega(q_{i_{j+1}-1})\}$, for $j \in \mathbb{N}$. According to Lemma 2.4, we have $\bigsqcup\inf_{j\to\infty} c_j = \bigsqcup\inf_{i\to\infty}(\Omega(q_i))$. Note that i_0 can be chosen large enough such that $\bigsqcup\inf_{j\to\infty} c_j = \bigsqcup_{i=j}^{\infty} c_j$.

Since g is idempotent, we have $g^{i_{j+1}-i_j} = g$ for every $j \in \mathbb{N}$ and therefore $c_j \preceq g(q, q)$ for every $j \in \mathbb{N}$. Since ρ was assumed to be accepting, $\bigsqcup_{j\to\infty} c_j$ is even. According to Lemma 2.3, $g(q, q)$ would have to be even as well, which contradicts the assumption that g is bad.

"$\Rightarrow$" Suppose that $w = a_0a_1a_2\ldots \in NW(\Sigma) \setminus L(\mathcal{A})$. Note that w does not contain any pending positions by assumption. Then there must be infinitely many positions $i_0 < i_1 < \cdots$ in which the stack in any run on w becomes empty. Then the sequence $i_0 < i_1 < \cdots$ splits the infinite nested word w into infinitely many finite nested words $a_{i_0}\ldots a_{i_1-1}, a_{i_1}\ldots a_{i_2-1}, \ldots$, which are all well-matched.

Let $I := \{i_j \mid j \in \mathbb{N}\}$ and $I^{(2)} := \{(i_j, i_{j'}) \mid j, j' \in \mathbb{N} \text{ with } j < j'\}$, and consider the coloring $\chi : I^{(2)} \to \mathfrak{T}$ defined as $\chi(i, i') := f_{a_i \ldots a_{i'-1}}$. Note that χ is well-defined, since $a_i \ldots a_{i'-1}$ is a well-matched nested word, for $(i, i') \in I^{(2)}$, hence $f_{a_i\ldots a_{i'-1}} \in \mathfrak{T}$.

```
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
```

Fig. 4. Universality check UNIV for VPAs with respect to well-matched words.

Furthermore, note that $\mathfrak{T}$ is finite. By Ramsey's Theorem [Ramsey 1928], there is an infinite subset J of I and a TP $g \in \mathfrak{T}$ such that $\chi(j, j') = g$, for all $j, j' \in J$ with $j < j'$. Let $J = \{j_0, j_1, \ldots\}$ with $0 < j_0 < j_1 < \ldots$, and let f be the int-TP $\chi(0, j_0)$. Recall that $i_0 = 0$. Let $u = a_0 \ldots a_{j_0-1}$ and $v_i = a_{j_i} \ldots a_{j_{i+1}-1}$ for all $i \geq 0$. Note that we have $f = f_u$ and $g = g_{v_i}$ for every $i \geq 0$.

We first observe that g is idempotent because we have

$$g \circ g = \chi(j_0, j_1) \circ \chi(j_1, j_2) = \chi(j_0, j_2) = g\,.$$

Note that the composition of $\chi(j_0, j_1)$ and $\chi(j_1, j_2)$ is defined since g is an int-TP.

It remains to be seen that g is bad for $f(q_I)$. Suppose that it is not. Then there is some $q' \in f(q_I)$ and some $q \in g(q')$ such that $g(q, q) = c$ for some even c. Let $c' := f(q_I, q')$, $c'' := g(q', q)$, $u := a_0 \ldots a_{j_0-1}$ and $v_i := a_{j_i} \ldots a_{j_{i+1}-1}$ for every $i \geq 0$. Note that $w = uv_0v_1v_2v_3\ldots$.

A run of $\mathcal{A}$ on w is easily obtained as

$$(q_I, \bot) \xrightarrow{u,c'}_{f_u} (q', \bot) \xrightarrow{v_0,c''}_{g_{v_0}} (q, \bot) \xrightarrow{v_1,c}_{g_{v_1}} (q, \bot) \xrightarrow{v_2,c}_{g_{v_2}} \cdots$$

It is accepting because the maximal color occurring infinitely often in it is c, which was assumed to be even.

3.2. Algorithmic Realization

Theorem 3.10 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 Figure 4. It computes $\mathfrak{T}$ by least-fixpoint iteration and checks at each stage whether two TPs exist that witness nonuniversality according to Theorem 3.10. 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 a finite number of TPs. For returning a witness of the VPA's nonuniversality, 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 3.11. *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 $m^3 \cdot 2^{\mathcal{O}(n^2 \cdot \log k)}$.*

PROOF. We assume the following time complexities of the following operations. (a) Checking whether two int-TPs are equal is in $\mathcal{O}(n^2)$. Note that for int-TPs f and g, we need to check for all tuples $(q, q') \in Q \times Q$ if the equality $f(q, q') = g(q, q')$ holds. (b) It follows that adding an int-TP to T or N costs $\mathcal{O}(n^2)$ time, since we need to compute the TP's hash value and make a lookup if the TP is already stored in the table.

(c) TP composition is carried out in $\mathcal{O}(n^3 \cdot m)$ time. (d) Checking whether an int-TP is idempotent is in $\mathcal{O}(n^3)$. (e) Finally, checking for badness of an int-TP is in $\mathcal{O}(n)$.

We observe that the number of int-TPs is bounded by $(k+1)^{n^2}$. Thus, N and T never store more than $(k+1)^{n^2}$ elements. It is easy to see that an int-TP is stored at most once in N at the beginning of the while loop starting at Line 3 of the algorithm. It follows that Lines 4 to 6 of the algorithm are executed at most once for a pair of int-TPs. In summary, Lines 4 to 6 take at most $2^{\mathcal{O}(n^2 \cdot \log k)}$ time.

It remains to analyze the time complexity of updating N and T in Lines 7 and 8 of the algorithm. The number of carried-out composition operations in an iteration is bounded by $\mathcal{O}(|N| \cdot |T| + |T_{\mathsf{call}}| \cdot |N| \cdot |T_{\mathsf{ret}}|)$. Since each int-TP appears at most once in N, the number composition operations in total is bounded by $\mathcal{O}(|T|^2 + |T| \cdot |\Sigma|^2)$. Note that $|T_{\mathsf{call}}|, |T_{\mathsf{ret}}| \leq |\Sigma|$. Since $|T| \leq (k+1)^{n^2}$ and the $\mathcal{O}(n^3 \cdot m)$ time complexity of TP composition, it follows that Line 7 (without removing the elements that are also in T) takes in total at most $m^3 \cdot 2^{\mathcal{O}(n^2 \cdot \log k)}$ time. Removing the elements that are also in T in Line 7 and T's update in Line 8 take in one iteration at most $2^{\mathcal{O}(n^2 \cdot \log k)}$ time. Since the algorithm never removes elements from T, the number of iterations of the algorithm is bounded by $2^{\mathcal{O}(n^2 \cdot \log k)}$.

Overall, we obtain the time complexity $m^3 \cdot 2^{\mathcal{O}(n^2 \cdot \log k)}$. □

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 Lines 4 to 6. Observe that it is sufficient to know the sets $f_u(q_I)$, for $f_u \in T$, that is, 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 $R \cup \{(uv, f_v(Q')) \,|\, (u, Q') \in R \text{ and } f_v \in T\}$ 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.

3.3. Extended Universality Check

For extending our universality check UNIV to account for infinite words that are not well matched, we introduce a new operation on TPs, the so-called *collapse* operation $(\cdot){\downarrow}$. It turns call-TPs and ret-TPs into int-TPs. For a call-TP f, we define

$$f{\downarrow}(q, q') \;:=\; \bigvee \left\{ f(q, B, q') \,\middle|\, B \in \Gamma \right\}$$

and

$$f{\downarrow}(q, q') \;:=\; f(q, \bot, q'),$$

when f is a ret-TP. For int-TPs, $(\cdot){\downarrow}$ is the identity. Intuitively, for call-TPs, the collapse operation chooses the stack symbol for which the value is best with respect to the reward ordering. For ret-TPs, the collapse operation takes the value for $\bot$, which occurs at the top of the stack when reading a symbol in Σ_{ret} if and only if the position is pending. The collapse operation extends in the natural way to sets, that is, $F{\downarrow} := \{ f{\downarrow} \mid f \in F\}$.

Definition 3.12. In addition to the sets $\mathfrak{T}$, we define the sets $\mathfrak{T}^{\mathsf{call}*}$ and $\mathfrak{T}^{\mathsf{ret}*}$ of TPs as the least solution of the following system of equations.

$$\begin{aligned}\mathfrak{T} &= T_{\mathsf{int}} \cup T_{\mathsf{call}} \circ T_{\mathsf{ret}} \cup T_{\mathsf{call}} \circ \mathfrak{T} \circ T_{\mathsf{ret}} \cup \mathfrak{T} \circ \mathfrak{T}\\ \mathfrak{T}^{\mathsf{call}*} &= T_{\mathsf{call}}\!\downarrow \cup \mathfrak{T} \cup \mathfrak{T}^{\mathsf{call}*} \circ \mathfrak{T}^{\mathsf{call}*}\\ \mathfrak{T}^{\mathsf{ret}*} &= T_{\mathsf{ret}}\!\downarrow \cup \mathfrak{T} \cup \mathfrak{T}^{\mathsf{ret}*} \circ \mathfrak{T}^{\mathsf{ret}*}\end{aligned}$$

The set $\mathfrak{T}$ is the same as before. $\mathfrak{T}^{\mathsf{call}*}$ and $\mathfrak{T}^{\mathsf{ret}*}$ subsume $\mathfrak{T}$. Additionally, they contain the collapsed atomic call-TPs and ret-TPs, respectively, and they are closed under the composition operation $\circ$. The intuition is the following. $\mathfrak{T}^{\mathsf{call}*}$ contains TPs that describe runs like TPs from $\mathfrak{T}$. In particular, the stack content is ignored. The positions for which the runs of the TPs in $\mathfrak{T}^{\mathsf{call}*}$ are connected by the composition are pending call positions. Although the VPA pushes at these positions in each run a symbol on the stack, it never pops these symbols later. Thus, the actual symbols are irrelevant. The intuition for $\mathfrak{T}^{\mathsf{ret}*}$ is similar. Here, the positions are pending return positions and the stack symbol is always $\bot$.

The following theorem characterizes (non)universality of the VPA $\mathcal{A}$ with respect to the sets $NW_{\mathsf{call}}(\Sigma)$, $NW_{\mathsf{ret}}(\Sigma)$, and $NW_{\mathsf{any}}(\Sigma)$. Its proof proceeds along the same lines as the proof of Theorem 3.10, thus is omitted.

THEOREM 3.13.

(1) $L_{\mathsf{call}}(\mathcal{A}) \neq NW_{\mathsf{call}}(\Sigma)$ *if and only if there are TPs* $f, g \in \mathfrak{T}^{\mathsf{call}*}$ *such that g is idempotent and bad for* $f(q_I)$.
(2) $L_{\mathsf{ret}}(\mathcal{A}) \neq NW_{\mathsf{ret}}(\Sigma)$ *if and only if there are TPs* $f, g \in \mathfrak{T}^{\mathsf{ret}*}$ *such that g is idempotent and bad for* $f(q_I)$.
(3) $L_{\mathsf{any}}(\mathcal{A}) \neq NW_{\mathsf{any}}(\Sigma)$ *if and only if there are TPs* $f, g \in \mathfrak{T}^{\mathsf{ret}*}$ *or* $f \in \mathfrak{T}^{\mathsf{call}*} \cup \mathfrak{T}^{\mathsf{ret}*} \cup (\mathfrak{T}^{\mathsf{ret}*} \circ \mathfrak{T}^{\mathsf{call}*})$ *and* $g \in \mathfrak{T}^{\mathsf{call}*}$ *such that g is idempotent and bad for* $f(q_I)$.

Note that a word in $NW_{\mathsf{any}}(\Sigma)$ might contain pending call and pending return positions. However, all pending return positions must occur before the pending call positions. In this case, the pending positions must occur in a finite prefix of the infinite word.

With Definition 3.12 and Theorem 3.13, it is straightforward to adapt the algorithm UNIV from Figure 4 so that it checks universality with respect to the sets $NW_{\mathsf{call}}(\Sigma)$, $NW_{\mathsf{ret}}(\Sigma)$, and $NW_{\mathsf{any}}(\Sigma)$. The asymptotic time complexity does not alter.

4. VARIANTS

In this section, we describe variants of the universality check presented in Section 3. In particular, we adapt it to stair VPAs [Löding et al. 2004] and to VPAs over finite words. Finally, we discuss its relation to complementation and determinization constructions for VPAs.

4.1. Universality Check for Stair VPAs

Löding et al. [2004] propose another acceptance notion for nested words. For example, for a run on a well-matched word, only the priorities of the states of a run at the positions that are not positions of a nested subword are relevant for the run's acceptance. The underlying intuition is that acceptance is determined only by the priorities of the states visited by the topmost procedure; the priorities of the states seen in subprocedures are irrelevant. These VPAs are dubbed *stair VPAs*. We first recall their definition before we describe how to adapt our universality check to stair VPAs.

Let $P = \{i_0, i_1, \dots\}$ be an infinite set of natural numbers with $i_0 < i_1 < \cdots$. For $w \in \Sigma^\omega$, we define $w\!\restriction_P$ as the word $w_{i_0} w_{i_1} \ldots \in \Sigma^\omega$. Moreover, we define $P_w := \{i \in \mathbb{N} \mid \mathrm{sh}(w_0 w_1 \ldots w_{j-1}) \geq \mathrm{sh}(w_0 w_1 \ldots w_{i-1}),\ \text{for all } j \in \mathbb{N} \text{ with } j \geq i\}$, where

$\mathrm{sh}(\varepsilon) := 0$ and

$$\mathrm{sh}(ua) := \begin{cases} \mathrm{sh}(u) & \text{if } a \in \Sigma_{\mathsf{int}}, \\ \mathrm{sh}(u) + 1 & \text{if } a \in \Sigma_{\mathsf{call}}, \\ \max\{0, \mathrm{sh}(u) - 1\} & \text{if } a \in \Sigma_{\mathsf{ret}}, \end{cases}$$

for $u \in \Sigma^*$ and $a \in \Sigma$. Intuitively, $\mathrm{sh}(u)$ is the stack height (excluding the bottom stack symbol $\bot$) of a VPA after reading the finite word $u \in \Sigma^*$. The set P_w factorizes the word $w \in \Sigma^\omega$ into $u^{(1)}u^{(2)}\dots$ such that each $u^{(i)} \in \Sigma^+$ is either (1) a well-matched word of the form cur with $c \in \Sigma_{\mathsf{call}}$, $u \in \Sigma^*$, $r \in \Sigma_{\mathsf{ret}}$, and maximal in w or (2) a letter in Σ and pending in w if it is a call or return. Note that P_w is always an infinite set. Let $\mathcal{A} = (Q, \Gamma, \Sigma, \delta, q_I, \Omega)$ be a VPA and $\varrho \in (Q \times \Gamma_\bot^+)^\omega$ a run of $\mathcal{A}$ on $w \in \Sigma^\omega$ with $\varrho = (q_0, \gamma_0)(q_1, \gamma_1)\cdots$. We say that ϱ is *stair accepting* if $\max\{\Omega(q) \mid q \in \inf(q_0q_1 \dots \upharpoonright_{P_w})\}$ is even. For $t \in \{\mathsf{match}, \mathsf{call}, \mathsf{ret}, \mathsf{any}\}$, we define

$$S_t(\mathcal{A}) := \{w \in NW_t(\Sigma) \mid \text{there is a stair accepting run of } \mathcal{A} \text{ on } w\}.$$

As for $L_{\mathsf{match}}(\mathcal{A})$, we omit the subscript match in $S_{\mathsf{match}}(\mathcal{A})$, for the sake of brevity.

In the following, we sketch the changes to the algorithm UNIV from Section 3 so that it allows us to check universality with respect to the well-matched words for a given stair VPA $\mathcal{A}$, that is, $S(\mathcal{A}) = NW(\Sigma)$. The other cases for nonwell-matched words with $t \in \{\mathsf{call}, \mathsf{ret}, \mathsf{any}\}$ are similar to the ones described in Section 3.3.

We start with adapting the characterization of Theorem 3.10 for stair VPAs. Recall that $\mathfrak{T}$ is the least solution of the equation

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

From $\mathfrak{T}$, we obtain the set of int-TPs $S := \{f_\Omega \mid f \in T_{\mathsf{call}} \circ \mathfrak{T} \circ T_{\mathsf{ret}}\}$, where $f_\Omega : Q \times Q \to \Omega(Q)_\dagger$ is the int-TP obtained from the int-TP $f : Q \times Q \to \Omega(Q)_\dagger$ defined as $f_\Omega(q, q') := \Omega(q')$ if $f(q, q') \in \Omega(Q)$ and $f_\Omega(q, q') := \dagger$ if $f(q, q') = \dagger$, for $q, q' \in Q$. That is, S only contains modified int-TPs that correspond to well-matched words of the form cur with $c \in \Sigma_{\mathsf{call}}$, $u \in \Sigma^*$, and $r \in \Sigma_{\mathsf{ret}}$. The modification to the int-TP f_{cur} is the overwrite of each of the priorities seen in the runs on the word cur by the priority of the state in which the runs end. The symbol $\dagger$ still denotes the nonexistence of such a run. Let $\mathfrak{S}$ be the least solution of the equation

$$\mathfrak{S} \;=\; S \,\cup\, T_{\mathsf{int}} \,\cup\, \mathfrak{S} \circ (S \cup T_{\mathsf{int}})\,.$$

THEOREM 4.1. *$S(\mathcal{A}) \neq NW(\Sigma)$ if and only if there are TPs $f, g \in \mathfrak{S}$ such that g is idempotent and bad for $f(q_I)$.*

The modifications to the algorithm UNIV are straightforward. The modified algorithm computes $\mathfrak{S}$ iteratively similar to UNIV's computation of $\mathfrak{T}$. In each iteration, we compose every newly encountered int-TP f_u with every atomic call-TP f_c and every atomic ret-TP f_r. Furthermore, we overwrite the priorities of the resulting int-TP f_{cur}. We then update the set that underapproximates $\mathfrak{S}$. That is, we compose the containing TPs with the newly obtained int-TPs f_{cur}. The check in an iteration for the existence of a pair of int-TPs (f, g) witnessing the VPA's nonuniversality does not alter. The worst-case complexity from Theorem 3.11 carries over to this variant of UNIV.

4.2. Universality Check for VPAs over Finite Nested Words

The set $NW_t^*(\Sigma)$ of finite nested words over an alphabet Σ, for $t \in \{\mathsf{match}, \mathsf{call}, \mathsf{ret}, \mathsf{any}\}$, is defined as expected to its corresponding counterpart $NW_t(\Sigma)$ of infinite nested words. Also, the finite-word language $L_t^*(\mathcal{A})$ of a VPA $\mathcal{A}$ is defined as expected. For uniformity, instead introducing a set of final states, we define a run on a finite word as *accepting* if the last state in the run has an even parity. As for infinite words, we omit, for the sake of brevity, the subscript match and write $NW^*(\Sigma)$ and $L^*(\mathcal{A})$ for $NW^*_{\mathsf{match}}(\Sigma)$ and $L^*_{\mathsf{match}}(\mathcal{A})$, respectively.

Checking universality of the VPA $\mathcal{A}$ with respect to finite words using transition profiles is straightforward with the machinery developed in Section 3.1. Without loss of generality, we assume that $\varepsilon \in L^*(\mathcal{A})$. This special case can be checked separately. We first characterize the nonuniversality of $\mathcal{A}$'s finite word language in terms of the existence of certain transition profiles.

Theorem 4.2. *$L^*(\mathcal{A}) \neq NW^*(\Sigma)$ if and only if there is a TP $f \in \mathfrak{T}$ such that $f(q_I, q) = \dagger$ or $\Omega(q)$ is odd, for all states $q \in Q$.*

The characterizations with respect to the other sets $NW^*_{\mathsf{call}}(\Sigma)$, $NW^*_{\mathsf{ret}}(\Sigma)$, and $NW^*_{\mathsf{any}}(\Sigma)$ can easily be derived in a similar manner, thus are omitted here.

The algorithmic realization is a straightforward adaptation of the algorithm UNIV in Figure 4. Note that further optimizations are possible. For instance, the TPs do not need to keep track of the maximal occurred priorities in the runs that they represent. The resulting asymptotic complexity is $m^3 \cdot 2^{\mathcal{O}(n^2)}$, where n and m are as in Theorem 3.11. The number of iterations is bounded by $2^{\mathcal{O}(n^2)}$.

4.3. Relation to Determinization and Complementation

In this section, we derive complementation constructions for VPAs from the machinery developed in Section 3.1. We start with a complementation construction for VPAs over finite well-matched words and extend it afterwards to infinite well-matched words.

With the machinery developed in Section 3.1, a complementation construction is straightforward. For a VPA $\mathcal{A} = (Q, \Gamma, \Sigma, \delta, q_I, \Omega)$, we define the VPA $\mathcal{C}$ as the tuple $(Q', \Gamma'_\perp, \Sigma, \delta', q'_I, \Omega')$, where its components are:

—$Q' := \{f \mid f \text{ int-TP}\}$,
—$\Gamma' := \{f \mid f \text{ call-TP}\}$,
—for $f, g \in Q'$ and $a \in \Sigma$, the transitions are $\delta'_{\mathsf{int}}(f, a) := \{f \circ f_a\}$ if $a \in \Sigma_{\mathsf{int}}$, $\delta'_{\mathsf{call}}(f, a) := \{(q'_I, f \circ f_a)\}$ if $a \in \Sigma_{\mathsf{call}}$, and $\delta'_{\mathsf{ret}}(f, g, a) := \{(g \circ f) \circ f_a\}$ and $\delta'_{\mathsf{ret}}(f, \perp, a) := \emptyset$ if $a \in \Sigma_{\mathsf{ret}}$,
—$q'_I(q, q) := 0$ and $q'_I(q, q') := \dagger$, for $q, q' \in Q$ with $q \neq q'$, and
—for $f \in Q'$, we define $\Omega'(f) := 1$ if $f(q_I, q) \neq \dagger$ and $\Omega(q)$ is even, for some $q \in Q$, and $\Omega'(f) := 0$, otherwise.

The value of $\delta'_{\mathsf{ret}}(f, \perp, a)$ is actually irrelevant, since we assume inputs to be well-matched words.

Note that $\mathcal{C}$ is deterministic. This construction is similar to Alur and Madhusudan's determinization construction for nested-word automata over finite inputs [Alur and Madhusudan 2009], with some minor differences.[2] One difference is due to our objective to obtain a VPA that accepts the complement of $\mathcal{A}$'s language. This is reflected in the definition of the priority function Ω'. The state set Q' and the stack alphabet Γ' are also different from Alur and Madhusudan's construction. We use sets of TPs and Alur and Madhusudan use the sets $2^{Q \times Q}$ and $2^{Q \times Q} \times \Sigma_{\mathsf{call}}$, respectively. Elements from these sets represent similar information about the runs of $\mathcal{A}$. Finally, Alur and Madhusudan's determinization construction deals with pending positions in inputs and produces VPAs with slightly fewer states.

Proposition 4.3. $L^*(\mathcal{C}) = NW^*(\Sigma) \setminus L^*(\mathcal{A})$.

[2]The determinization construction of the printed version of the article is flawed. The error has been corrected; see the erratum at the publisher's website of the article.

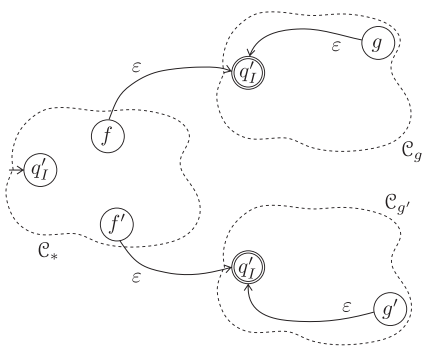

Fig. 5. Illustration of the complementation construction. States with a single circle have priority 1 and states with a double circle have priority 2.

The detailed proof proceeds along the lines of the one in Alur and Madhusudan [2009]. It does not rely on Ramsey's Theorem. Here, we give some intuition about the correctness of this construction. The VPA $\mathcal{C}$ uses the int-TPs to keep track of the essential behavior of all of $\mathcal{A}$'s runs on the input processed so far. In addition to the classical subset construction [Rabin and Scott 1959], an int-TP f stores the information about the existence of a run from a state q to a state q', that is, $f(q, q') \neq \dagger$. This information is needed when returning from a call, where the information f about $\mathcal{A}$'s runs on the subword is combined with (1) the information g about $\mathcal{A}$'s runs on the prefix up to the matching call position and (2) the information f_a about $\mathcal{A}$'s runs on the current letter $a \in \Sigma_{\mathsf{ret}}$. $\mathcal{C}$ has pushed the call-TP g on its stack when reading the letter at the corresponding call position. Now, it pops it from the stack and puts the information of the different parts correctly together, that is, $\mathcal{C}$'s new state h after reading the return letter a is the composition of the TPs $h := (g \circ f) \circ f_a$. Note that this composition is always defined, since g is a call-TP, f an int-TP, and f_a a ret-TP. Furthermore, note that composing a TP with the int-TP q'_I does not alter the TP.

In the following, we sketch a complementation construction for VPAs on infinite well-matched words. In fact, our construction is a direct extension of the Ramsey-based complementation construction for Büchi automata (see, e.g., Büchi [1962] and Breuers et al. [2012]). It is different from the one given by Alur and Madhusudan [2009]. There, the complementation construction is split into three construction steps. One automaton flattens the hierarchical structure of the inputs by transforming nested words into so-called pseudo-runs. Another automaton reads such pseudo-runs and decides whether to accept or reject the input. The final construction step combines both automata to yield an automaton that accepts the complemented language.

Here, we construct a VPA $\mathcal{C}'$ for the complement of $L(\mathcal{A})$ directly. Let $\mathcal{C}$ be the VPA obtained from previously described complementation construction for finite words. We assume, without loss of generality, that $\mathcal{C}$'s initial state q'_I has only outgoing transitions. The VPA $\mathcal{C}'$ consists of several slightly modified copies of the VPA $\mathcal{C}$ defined earlier, namely, $\mathcal{C}_*$ and $\mathcal{C}_g$, for each int-TP g. See Figure 5 for illustration.

The component $\mathcal{C}_*$ takes care of finite prefixes of the inputs, and a component $\mathcal{C}_g$ takes care of infinite suffixes of the inputs on which $\mathcal{A}$'s runs are looping with respect to the TP g. The initial state of $\mathcal{C}'$ is the initial state q'_I in the copy $\mathcal{C}_*$ of $\mathcal{C}$. All states in $\mathcal{C}_*$ have the odd priority 1. The copy of $\mathcal{C}$'s initial state q'_I in the component $\mathcal{C}_g$ has the even priority 2; all the other states in this component have the odd priority 1. Furthermore, we add an ε-transition from $\mathcal{C}_g$'s state g to its copy of $\mathcal{C}$'s initial state q'_I. The component $\mathcal{C}_*$ is connected to the other components as follows. For each pair (f, g) of int-TPs with $f \circ g = f$, and g idempotent and bad for $f(q_I)$, we connect the state f

in $\mathcal{C}_*$ with the state q'_I in $\mathcal{C}_g$ by an ε-transition. We can delete the component $\mathcal{C}_g$ if $\mathcal{C}_*$ is not connected to $\mathcal{C}_g$ by some ε-transition. Note that the ε-transitions can be eliminated in the standard way.

We remark that $\mathcal{C}''$s acceptance condition is essentially a Büchi acceptance condition since the only priorities that are assigned to $\mathcal{C}''$s states are 1 and 2. However, note that we do not make any requirements about the priorities that are assigned to $\mathcal{A}$'s states.

Proposition 4.4. $L(\mathcal{C}') = NW(\Sigma) \setminus L(\mathcal{A})$.

The proof proceeds along the lines of the proof of Büchi's complementation construction and uses similar arguments as in the proof of Theorem 3.10. In particular, showing that the language on the right-hand side is a subset of the language on the left-hand side relies on Ramsey's Theorem. Details are omitted.

We conclude this section by commenting on the differences and similarities of the complementation construction and algorithm UNIV for checking universality of a given VPA. TPs are the basic building blocks of the complementation construction presented earlier, and they are also at the core of UNIV. This is actually not surprising since, for both problems, one needs to investigate all runs on any input. TPs are an appropriate entity for this purpose. However, the search space for UNIV is more concise and explored with less overhead. The complementation construction involves more bookkeeping. In order to build the complement automaton, we must determine and store the transitions between its states, which are essentially int-TPs. First, we need to store multiple copies of an int-TP (or pointers to it) for the states in the different copies of the VPA $\mathcal{C}$. Similarly, a call-TP might occur as a stack symbol in several transitions for call and return letters. Second, in the complementation construction, we keep track of how exactly a state corresponding to an int-TP f is reachable, which might be different for well-matched words $u, v \in \Sigma^+$ with $f_u = f_v = f$ but $u \neq v$. In contrast, the universality check UNIV stores only the TPs and iteratively composes them. Finally, in the complementation construction, we combine TPs f with atomic TPs f_a only, for determining the successor states of states for the letters $a \in \Sigma$. The universality check UNIV constructs the TPs less stringently in the sense that in each iteration already constructed TPs f_u and f_v, with $u, v \in \Sigma^+$, are composed whenever their composition $f_u \circ f_v$ is defined.

5. 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 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 $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ holds.

TPs 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

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

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 Definition 3.1.

—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 $F_a := \{ f_a^{\langle p,\Omega(p'),B,p'\rangle} \mid p, p' \in Q,\ B \in \Gamma_\perp, \text{ and } p' \in \delta_{\mathsf{ret}}(p, B, a)\}$.

As with TPs, the composition of TTPs is only allowed in certain cases, which are even more restrictive now. For instance, it is still not possible to compose a call-TTP with a call-TTP. Moreover, the tags that contain information about some run in $\mathcal{A}_1$ have to match: a TTP $f^{\langle p,c,p'\rangle}$, for instance, can be composed only with a TTP, where its tag describes the existence of a run of $\mathcal{A}_1$ that starts from the state p'.

The composition of two TTPs extends the composition of the underlying TPs by also 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 Definition 3.6, 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 5.1. *$L(\mathcal{A}_1) \not\subseteq L(\mathcal{A}_2)$ if and only if 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 $f(q_I^2)$.*

PROOF. "$\Leftarrow$" Suppose that there are TTPs $f^{\langle q_I^1,c,p\rangle}$ and $g^{\langle p,d,p\rangle}$ with the Properties (1) and (2). Assume that $f = f_u$ and $g = f_v$, for some well-matched $u, v \in \Sigma^+$. It is easy to see that there is a run $(q_0^1, \gamma_0^1)(q_1^1, \gamma_1^1)\ldots$ of $\mathcal{A}_1$ on uv^ω with $(q_0^1, \gamma_0^1) = (q_I^1, \perp)$ and $(q^1_{|u|+i|v|}, \gamma^1_{|u|+i|v|}) = (p, \perp)$, for all $i \in \mathbb{N}$. In particular, the stack content is $\perp$ after reading uv^i since u and v are well matched. Furthermore, $d = \bigsqcup\{\Omega(q) \mid q \in \inf(q_0^1 q_1^1 \ldots)\}$.

```
1 $N \leftarrow \hat{T}_{\mathsf{int}} \cup \hat{T}_{\mathsf{call}} \circ \hat{T}_{\mathsf{ret}}$
2 $T \leftarrow N$
3 while $N \neq \emptyset$ do
4     forall $(f_u^{\langle p,c,p' \rangle}, f_v^{\langle q,d,q' \rangle}) \in N \times T \cup T \times N$ do
5         if $p = q_I^1$, $p' = q = q'$, $d$ even, $f_v$ idempotent, and $f_v$ bad for $f_u(q_I^2)$ then
6             return inclusion does not hold, witnessed by $uv^\omega$
7     $N \leftarrow \big(N \circ T \cup T \circ N \cup \hat{T}_{\mathsf{call}} \circ N \circ \hat{T}_{\mathsf{ret}}\big) \setminus T$
8     $T \leftarrow T \cup N$
9 return inclusion holds
```

Fig. 6. Inclusion check INCL for VPAs with respect to well-matched words.

It follows that $uv^\omega \in L(\mathcal{A}_1)$. The fact that $uv^\omega \notin L(\mathcal{A}_2)$ is a simple consequence of Theorem 3.10. Note that Property (2) is exactly the condition that is sufficient for $\mathcal{A}_2$ not to accept this uv^ω according to that theorem.

"$\Rightarrow$" Suppose that there is a well-matched word $w = a_0a_1 \cdots \in L(\mathcal{A}_1) \cap NW(\Sigma) \setminus L(\mathcal{A}_2)$. Let $(q_0^1, \gamma_0^1)(q_1^1, \gamma_1^1)\dots$ be an accepting run of $\mathcal{A}_1$ on w. Thus, there is an even priority d and a $j_0 \in \mathbb{N}$ such that d is the maximal priority occurring infinitely often in this run, and no greater priority occurs after position j_0. Let c be the maximal priority occurring before position j_0.

As in the proof of Theorem 3.10, we define an infinite sequence $i_0 < i_1 < \dots$ with $i_0 = 0$ such that $a_{i_j} \dots a_{i_{j+1}-1}$ is a well-matched word, for each $j \in \mathbb{N}$. However, we additionally require $i_1 \geq j_0$. Now, consider the coloring $\chi(i_j, i_{j'}) := f_v$ with $v = a_{i_j} \dots a_{i_{j'}-1}$. As in the proof of the direction from left to right of Theorem 3.10, Ramsey's Theorem yields TPs f' and g' in $\mathfrak{T}$ such that g' is idempotent and bad for $f'(q_I^2)$. By the pigeonhole principle, there must be some $j, j' \in \mathbb{N}$ such that $j < j'$ and $q_{i_j} = q_{i_{j'}} = p$ for some $p \in Q$. Define the TPs

$$f := f' \circ \underbrace{g' \circ \dots \circ g'}_{j \text{ times}} \quad \text{and} \quad g := \underbrace{g' \circ \dots \circ g'}_{j'-j \text{ times}} .$$

Since g' was supposed to be idempotent and $j' - j \geq 1$, we have in fact $g = g'$. Furthermore, depending on whether or not $j = 0$, we have either $f = f' \circ g$ or $f = f'$. Then, clearly, we have that g is bad for $f(q_I^2)$.

The atomic TPs that compose f and g can now be tagged with single transitions of $\mathcal{A}_1$'s accepting run such that their compositions become the TTPs $f^{\langle q_I^1, c, p \rangle}$ and $g^{\langle p,d,p \rangle}$, which finishes the proof. $\square$

Theorem 5.1 yields an algorithm INCL to check $L(\mathcal{A}_1) \not\subseteq L(\mathcal{A}_2)$, for given VPAs $\mathcal{A}_1$ and $\mathcal{A}_2$. It is along the same lines as the algorithm UNIV, and shown in Figure 6 for well-matched words. 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 Section 3 also apply to INCL.

For the complexity analysis of the algorithm INCL to follow, 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 section. Only the input alphabet Σ is the same for $\mathcal{A}_1$ and $\mathcal{A}_2$.

THEOREM 5.2. *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 $n_1^4 \cdot k_1^2 \cdot m_1 \cdot m_2^3 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$.*

Table II. Statistics on the Input Instances

	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	$\subseteq$ / $\subseteq$	$\not\subseteq$ / $\subseteq$	$\not\subseteq$ / ?	$\subseteq$ / $\subseteq$	$\subseteq$ / $\subseteq$

Proof. We observe that there are at most $n_1^2 \cdot k_1 \cdot (k_2+1)^{n_2^2}$ int-TTPs. Similar to the algorithm UNIV, the total time of the check in the Lines 4 to 6 of the algorithm INCL is dominated by the number of int-TTP pairs, which is bounded by $n_1^4 \cdot k_1^2 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$.

For the update N in Line 7, we need to compose TTPs in N with TTPs in T, $\hat{T}_{\mathsf{call}}$, and $\hat{T}_{\mathsf{ret}}$. Note that a TTP consists of a TP and information about $\mathcal{A}_1$'s behavior. A requirement for composing TTPs is that their information on $\mathcal{A}_1$'s behavior fits together. For instance, the composition of the int-TTPs $f^{\langle p,c,p'\rangle}$ and $g^{\langle q,d,q'\rangle}$ is defined only when $p' = q$. We can reduce the number TTP compositions by grouping TTPs in T, N, $\hat{T}_{\mathsf{call}}$, and $\hat{T}_{\mathsf{ret}}$ with respect to their information about $\mathcal{A}$'s behavior. For example, when int-TTPs $f^{\langle p,c,p'\rangle}$ and $g^{\langle q,d,q'\rangle}$ are both in T, we group them together in the case that $p = q$ and $p' = q'$. It follows that the total number of TTP compositions is bounded by $n_1^4 \cdot k_1 \cdot m_1 \cdot m_2^2 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$. Note that each int-TTP in T is at most once in N and since the states determine the priority in the information about $\mathcal{A}_1$'s behavior in the TTPs in $\hat{T}_{\mathsf{call}}$ and $\hat{T}_{\mathsf{ret}}$, we have that $|\hat{T}_{\mathsf{call}}|, |\hat{T}_{\mathsf{ret}}| \in \mathcal{O}(n_1^2 \cdot m_1 \cdot |\Sigma|)$. Since equality between two int-TTPs can be checked in $\mathcal{O}(n_2^2)$ time and TTP composition can be carried out in $\mathcal{O}(n_2^3 \cdot m_2)$ time, the updates of N (without removing elements that are also in T) take $n_1^4 \cdot k_1 \cdot m_1 \cdot m_2^3 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$ time in total. Removing the elements that are also in T in Line 8 and updating T in Line 8 take $n_1^2 \cdot k_1 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$ time in one iteration and $n_1^4 \cdot k_1^2 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$ in total, since the number of iterations is bounded $n_1^2 \cdot k_1 \cdot 2^{\mathcal{O}(n_2^2 \cdot \log k_2)}$.

By putting these upper bounds together, we obtain the claimed upper bound on the time complexity of the algorithm INCL. □

An extension to check inclusion to nested words with pending positions is along the same lines as the corresponding extensions of the universality check in Section 3. We omit the details here.

6. EVALUATION

Our prototype tool FADecider implements the presented algorithms in the programming language OCaml [Leroy et al. 2011].[3] To evaluate the tool's performance, we carried out the following experiments for which we used a 64bit Linux machine with 4GB of main memory and two dual-core Xeon 5110 CPUs, each with 1.6GHz.

Our benchmark suite consists of VPAs from Driscoll et al. [2011], which are extracted from real-world recursive imperative programs. Table II describes the instances, each consisting of two VPAs $\mathcal{A}$ and $\mathcal{B}$, in more detail. 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 nonaccepting. The second row lists whether the inclusions $L^*(\mathcal{A}) \subseteq L^*(\mathcal{B})$ and $L(\mathcal{A}) \subseteq L(\mathcal{B})$ of the respective VPAs hold.

Table III shows FADecider's running times for the inclusion checks $L^*(\mathcal{A}) \subseteq L^*(\mathcal{B})$ and $L(\mathcal{A}) \subseteq L(\mathcal{B})$. The row "FADecider" lists the running times for the tool FADecider for checking $L^*(\mathcal{A}) \subseteq L^*(\mathcal{B})$ and $L(\mathcal{A}) \subseteq L(\mathcal{B})$. The row "#TTPs" lists the number of encountered TTPs. The symbol ‡ indicates that FADecider ran out of time (2 hours).

[3]The tool (version 0.4) is publicly available at github.com/oliverfriedmann/fadecider.

Table III. Experimental Results for the Language-Inclusion Checks

	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

For comparison, we used the OpenNWA library [Driscoll et al. 2012]. See the row "OpenNWA" in Table III. It 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 $\mathcal{B}$. 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 therefore used 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.

7. CONCLUSION

Checking universality and language inclusion for automata by avoiding explicit determinization and complementation has attracted a lot of attention (see, e.g., Abdulla et al. [2011], Fogarty and Vardi [2009], Doyen and Raskin [2009], De Wulf et al. [2006], and Friedmann and Lange [2012]). 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 been extended to VPAs, however, only to VPAs over finite words [Bruyère et al. 2013]. It remains to be seen if optimizations for the Ramsey-based algorithms for Büchi automata [Abdulla et al. 2011] 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 multistack VPAs [La Torre et al. 2007] (see also Madhusudan and Parlato [2011]).

ACKNOWLEDGMENTS

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

REFERENCES

P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. 2011. Advanced Ramsey-based Büchi automata inclusion testing. In *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR'11), Lecture Notes in Computer Science*, Vol. 6901. Springer, Berlin, 187–202.

P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. 2010. When simulation meets antichains. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10), Lecture Notes in Computer Science*, Vol. 6015. Springer, Berlin, 158–174.

R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. 2005. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems* 27, 4, 786–818.

R. Alur and P. Madhusudan. 2009. Adding nesting structure to words. *Journal of the ACM* 56, 3, 1–43.

T. Ball and S. K. Rajamani. 2000. *Boolean programs: A model and process for software analysis*. Technical Report MSR-TR-2000-14. Microsoft Research.

S. Breuers, C. Löding, and J. Olschewski. 2012. Improved Ramsey-based Büchi complementation. In *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'12), Lecture Notes in Computer Science*, Vol. 7213. Springer, Berlin, 150–164.

V. Bruyère, M. Ducobu, and O. Gauwin. 2013. Visibly pushdown automata: Universality and inclusion via antichains. In *Proceedings of the 7th International Conference on Language and Automata Theory and Applications (LATA'13), Lecture Notes in Computer Science*, Vol. 7810. Springer, Berlin, 190–201.

J. R. Büchi. 1962. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Method, and Philosophy of Science*. Stanford University Press, Stanford, CA. 1–11.

Y. Choueka. 1974. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer Systems and Sciences* 8, 2, 117–141.

C. Dax, M. Hofmann, and M. Lange. 2006. A proof system for the linear time μ-calculus. In *Proceedings of the 26th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06), Lecture Notes in Computer Science*, Vol. 4337. Springer, Berlin, 273–284.

M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. 2006. Antichains: A new algorithm for checking universality of finite automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06), Lecture Notes in Computer Science*, Vol. 4144. Springer, Berlin, 17–30.

L. Doyen and J.-F. Raskin. 2009. Antichains for the automata-based approach to model-checking. *Logical Methods in Computer Science* 5, 1:5, 1–20.

E. Driscoll, A. Burton, and T. Reps. 2011. Checking conformance of a producer and a consumer. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference (FSE/ESEC'11)*. ACM Press, 113–123.

E. Driscoll, A. Thakur, and T. Reps. 2012. OpenNWA: A nested-word-automaton library. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12), Lecture Notes in Computer Science*, Vol. 7358. Springer, Berlin, 665–671.

S. Dziembowski, M. Jurdziński, and I. Walukiewicz. 1997. How much memory is needed to win infinite games? In *Proceedings of the 12th Symposium on Logic in Computer Science (LICS'97)*. IEEE Computer Society, 99–110.

E. A. Emerson and C. S. Jutla. 1991. Tree automata, μ-calculus and determinacy. In *Proceedings of the 32nd Symposium on Foundations of Computer Science (FOCS'91)*. IEEE Computer Society, 368–377.

S. Fogarty and M. Y. Vardi. 2009. Büchi complementation and size-change termination. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09), Lecture Notes in Computer Science*, Vol. 5505. Springer, Berlin, 16–30.

S. Fogarty and M. Y. Vardi. 2010. Efficient Büchi universality checking. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10), Lecture Notes in Computer Science*, Vol. 6015. Springer, Berlin, 205–220.

O. Friedmann, F. Klaedtke, and M. Lange. 2013. Ramsey goes visibly pushdown. In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP'13), Lecture Notes in Computer Science*, Vol. 7966. Springer, Berlin, 224–237.

O. Friedmann and M. Lange. 2012. Ramsey-based analysis of parity automata. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12), Lecture Notes in Computer Science*, Vol. 7214. Springer, Berlin, 64–78.

C. Fritz and T. Wilke. 2005. Simulation relations for alternating Büchi automata. *Theoretical Computer Science* 338, 1–3, 275–314.

R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. 1996. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV'95), IFIP Conference Proceedings*, Vol. 38. Chapman & Hall, London, 3–18.

M. Heizmann, J. Hoenicke, and A. Podelski. 2010. Nested interpolants. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*. ACM Press, 471–482.

D. Kähler and T. Wilke. 2008. Complementation, disambiguation, and determinization of Büchi automata unified. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08), Lecture Notes in Computer Science*, Vol. 5125. Springer, Berlin, 724–735.

S. La Torre, P. Madhusudan, and G. Parlato. 2007. A robust class of context-sensitive languages. In *Proceedings of the 22nd Symposium on Logic in Computer Science (LICS'07)*. IEEE Computer Society, 161–170.

C. S. Lee, N. D. Jones, and A. M. Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM Press, 81–92.

X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. 2011. *The OCaml system (release 3.12): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique (INRIA). http://caml.inria.fr.

C. Löding, P. Madhusudan, and O. Serre. 2004. Visibly pushdown games. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04), Lecture Notes in Computer Science*, Vol. 3328. Springer, Berlin, 408–420.

C. Löding and W. Thomas. 2000. Alternating automata and logics over infinite words. In *Proceedings of the IFIP International Conference on Theoretical Computer Science (IFIP TCS'00), Lecture Notes in Computer Science*, Vol. 1872. Springer, Berlin, 521–535.

P. Madhusudan and G. Parlato. 2011. The tree width of auxiliary storage. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, 283–294.

K. Mehlhorn. 1980. Pebbling mountain ranges and its application to DCFL-recognition. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming (ICALP'80), Lecture Notes in Computer Science*, Vol. 85. Springer, Berlin, 422–435.

M. Michel. 1988. Complementation is more difficult with automata on infinite words. CNET, Paris.

D. E. Muller and P. E. Schupp. 1987. Alternating automata on infinite trees. *Theoretical Computer Science* 54, 2–3, 267–276.

N. Piterman. 2007. From nondeterministic Büchi and Streett automata to deterministic parity automata. *Logical Methods in Computer Science* 3, 3:5, 1–21.

M. O. Rabin and D. Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 3, 2, 114–125.

F. P. Ramsey. 1928. On a problem of formal logic. *Proceedings of the London Mathematical Society* 30, 264–286.

S. Schewe. 2009. Tighter bounds for the determinisation of Büchi automata. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'09), Lecture Notes in Computer Science*, Vol. 5504. Springer, Berlin, 167–181.

A. P. Sistla, M. Y. Vardi, and P. Wolper. 1987. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science* 49, 2–3, 217–237.

M.-H. Tsai, S. Fogarty, M. Y. Vardi, and Y.-K. Tsay. 2011. State of Büchi complementation. In *Proceedings of the 15th International Conference on Implementation and Application of Automata (CIAA'10), Lecture Notes in Computer Science*, Vol. 6482. Springer, Berlin, 261–271.

M. Y. Vardi. 2007. The Büchi complementation saga. In *Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science (STACS'07), Lecture Notes on Computer Science*, Vol. 4393. Springer, Berlin, 12–22.

M. Y. Vardi and P. Wolper. 1986. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the 1st Symposium on Logic in Computer Science (LICS'86)*. IEEE Computer Society, 332–344.

M. Y. Vardi and P. Wolper. 1994. Reasoning about infinite computations. *Information and Computation* 115, 1, 1–37.

Received October 2013; revised March 2015; accepted May 2015