Automata Terms in a Lazy WS*k*S Decision Procedure

Vojtěch Havlena, Lukáš Holík, Ondřej Lengál(✉), and Tomáš Vojnar

FIT, IT4I Centre of Excellence,
Brno University of Technology, Brno, Czech Republic
lengal@fit.vutbr.cz

Abstract. We propose a lazy decision procedure for the logic WS*k*S. It builds a term-based symbolic representation of the state space of the tree automaton (TA) constructed by the classical WS*k*S decision procedure. The classical decision procedure transforms the symbolic representation into a TA via a bottom-up traversal and then tests its language non-emptiness, which corresponds to satisfiability of the formula. On the other hand, we start evaluating the representation from the top, construct the state space on the fly, and utilize opportunities to prune away parts of the state space irrelevant to the language emptiness test. In order to do so, we needed to extend the notion of *language terms* (denoting language derivatives) used in our previous procedure for the linear fragment of the logic (the so-called WS1S) into *automata terms*. We implemented our decision procedure and identified classes of formulae on which our prototype implementation is significantly faster than the classical procedure implemented in the MONA tool.

1 Introduction

Weak monadic second-order logic of k successors (WS*k*S) is a logic for describing regular properties of finite k-ary trees. In addition to talking about trees, WS*k*S can also encode complex properties of a rich class of general graphs by referring to their tree backbones [1]. WS*k*S offers extreme succinctness for the price of non-elementary worst-case complexity. As noticed first by the authors of [2] in the context of WS1S (a restriction that speaks about finite words only), the trade-off between complexity and succinctness may, however, be turned significantly favourable in many practical cases through a use of clever implementation techniques and heuristics. Such techniques were then elaborated in the tool MONA [3,4], the best-known implementation of decision procedures for WS1S and WS2S. MONA has found numerous applications in verification of programs with complex dynamic linked data structures [1,5–8], string programs [9], array programs [10], parametric systems [11–13], distributed systems [14,15], hardware verification [16], automated synthesis [17–19], and even computational linguistics [20].

Despite the extensive research and engineering effort invested into MONA, due to which it still offers the best all-around performance among existing WS1S/WS2S decision procedures, it is, however, easy to reach its scalability

P. Fontaine (Ed.): CADE 2019, LNAI 11716, pp. 300–318, 2019.
https://doi.org/10.1007/978-3-030-29436-6_18

limits. Particularly, MONA implements the classical WS1S/WS2S decision procedures that build a word/tree automaton representing models of the given formula and then check emptiness of the automaton's language. The non-elementary complexity manifests in that the size of the automaton is prone to explode, which is caused mainly by the repeated determinisation (needed to handle negation and alternation of quantifiers) and synchronous product construction (used to handle conjunctions and disjunctions). Users of WSkS are then forced to either find workarounds, such as in [6], or, often restricting the input of their approach, give up using WSkS altogether [21].

As in MONA, we further consider WS2S only (this does not change the expressive power of the logic since k-ary trees can be easily encoded into binary ones). We revisit the use of tree automata (TAs) in the WS2S decision procedure and obtain a new decision procedure that is much more efficient in certain cases. It is inspired by works on *antichain algorithms* for efficient testing of universality and language inclusion of finite automata [22–25], which implement the operations of testing emptiness of a complement (universality) or emptiness of a product of one automaton with the complement of the other one (language inclusion) via an *on-the-fly* determinisation and product construction. The on-the-fly approach allows one to achieve significant savings by pruning the state space that is irrelevant for the language emptiness test. The pruning is achieved by early termination when detecting non-emptiness (which represents a simple form of *lazy evaluation*), and *subsumption* (which basically allows one to disregard proof obligations that are implied by other ones). Antichain algorithms and their generalizations have shown great efficiency improvements in applications such as abstract regular model checking [24], shape analysis [26], LTL model checking [27], or game solving [28].

Our work generalizes the above mentioned approaches of on-the-fly automata construction, subsumption, and lazy evaluation for the needs of deciding WS2S. In our procedure, the TAs that are constructed explicitly by the classical procedure are represented symbolically by the so-called *automata terms*. More precisely, we build automata terms for subformulae that start with a quantifier (and for the top-level formula) only—unlike the classical procedure, which builds a TA for every subformula. Intuitively, automata terms specify the set of leaf states of the TAs of the appropriate (sub)formulae. The leaf states themselves are then represented by *state terms*, whose structure records the automata constructions (corresponding to Boolean operations and quantification on the formula level) used to create the given TAs from base TAs corresponding to atomic formulae. The leaves of the terms correspond to states of the base automata. Automata terms may be used as state terms over which further automata terms of an even higher level are built. Non-leaf states, the transition relation, and root states are then given implicitly by the transition relations of the base automata and the structure of the state terms.

Our approach is a generalization of our earlier work [29] on WS1S. Although the term structure and the generalized algorithm may seem close to [29], the reasoning behind it is significantly more involved. Particularly, [29] is based on

defining the semantics (language) of terms as a function of the semantics of their sub-terms. For instance, the semantics of the term $\{q_1, \ldots, q_n\}$ is defined as the union of languages of the state terms $q_1, \ldots, q_n$, where the language of a state of the base automaton consists of the words *accepted at that state*. With TAs, it is, however, not meaningful to talk about trees accepted from a leaf state, instead, we need to talk about a given state and its *context*, i.e., other states that could be obtained via a bottom-up traversal over the given set of symbols. Indeed, trees have multiple leafs, which may be accepted by a number of different states, and so a tree is *accepted from a set of states*, not from any single one of them alone. We therefore cannot define the semantics of a state term as a tree language, and so we cannot define the semantics of an automata term as the union of the languages of its state sub-terms. This problem seems critical at first because without a sensible notion of the meaning of terms, a straightforward generalization of the algorithm of [29] to trees does not seem possible. The solution we present here is based on defining the semantics of terms via the automata constructions they represent rather then as functions of languages of their sub-terms.

Unlike the classical decision procedure, which builds a TA corresponding to a formula *bottom-up*, i.e. from the atomic formulae, we build automata terms *top-down*, i.e., from the top-level formula. This approach offers a lot of space for various optimisations. Most importantly, we test non-emptiness of the terms *on the fly* during their construction and construct the terms *lazily*. In particular, we use *short-circuiting* for dealing with the $\wedge$ and $\vee$ connectives and *early termination* with possible *continuation* when implementing the fixpoint computations needed when dealing with quantifiers. That is, we terminate the fixpoint computation whenever the emptiness can be decided in the given computation context and continue with the computation when such a need appears once the context is changed on some higher-term level. Further, we define a notion of *subsumption* of terms, which, intuitively, compares the terms w.r.t the sets of trees they represent, and allows us to discard terms that are subsumed by others.

We have implemented our approach in a prototype tool. When experimenting with it, we have identified multiple parametric families of WS2S formulae where our implementation can—despite its prototypical form—significantly outperform MONA. We find this encouraging since there is a lot of space for further optimisations and, moreover, our implementation can be easily combined with MONA by treating automata constructed by MONA in the same way as if they were obtained from atomic predicates.

An extended version of this paper including proofs is available as [30].

2 Preliminaries

In this section, we introduce basic notation, trees, and tree automata, and give a quick introduction to the *weak monadic second-order logic of two successors* (WS2S) and its classical decision procedure. We give the minimal syntax of WS2S only; see, e.g., Comon *et al.* [31] for more details.

Basics, Trees, and Tree Automata. Let Σ be a finite set of symbols, called an *alphabet*. The set Σ^* of *words* over Σ consists of finite sequences of symbols from Σ. The *empty word* is denoted by ϵ, with $\epsilon \notin \Sigma$. The *concatenation* of two words u and v is denoted by $u.v$ or simply uv. The *domain* of a partial function $f : X \to Y$ is the set $\mathrm{dom}(f) = \{x \in X \mid \exists y : x \mapsto y \in f\}$, its *image* is the set $\mathrm{img}(f) = \{y \in Y \mid \exists x : x \mapsto y \in f\}$, and its *restriction* to a set Z is the function $f_{|Z} = f \cap (Z \times Y)$. For a binary operator $\bullet$, we write $A\,[\bullet]\,B$ to denote the augmented product $\{a \bullet b \mid (a, b) \in A \times B\}$ of A and B.

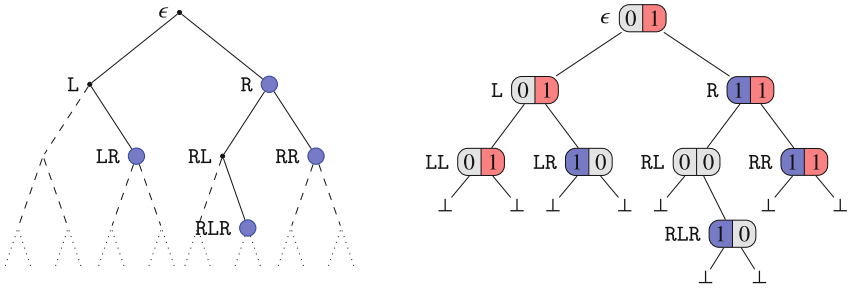
We will consider ordered binary trees. We call a word $p \in \{\mathtt{L}, \mathtt{R}\}^*$ a tree *position* and $p.\mathtt{L}$ and $p.\mathtt{R}$ its *left* and *right child*, respectively. Given an alphabet Σ s.t. $\bot \notin \Sigma$, a *tree* over Σ is a finite partial function $\tau : \{\mathtt{L}, \mathtt{R}\}^* \to (\Sigma \cup \{\bot\})$ such that (i) $\mathrm{dom}(\tau)$ is non-empty and prefix-closed, and (ii) for all positions $p \in \mathrm{dom}(t)$, either $\tau(p) \in \Sigma$ and p has both children, or $\tau(p) = \bot$ and p has no children, in which case it is called a *leaf*. We let $\mathit{leaf}(\tau)$ be the set of all leaves of τ. The position ϵ is called the *root*, and we write $\Sigma^{\text{木}}$ to denote the set of all trees over Σ[1].We abbreviate $\{a\}^{\text{木}}$ as $a^{\text{木}}$ for $a \in \Sigma$.

The *sub-tree* of τ rooted at a position $p \in \mathrm{dom}(\tau)$ is the tree $\tau' = \{p' \mapsto \tau(p.p') \mid p.p' \in \mathrm{dom}(\tau)\}$. A *prefix* of τ is a tree τ' such that $\tau'_{|\mathrm{dom}(\tau') \setminus \mathit{leaf}(\tau')} \subseteq \tau_{|\mathrm{dom}(\tau) \setminus \mathit{leaf}(\tau)}$. The *derivative* of a tree τ wrt a set of trees $S \subseteq \Sigma^{\text{木}}$ is the set $\tau - S$ of all prefixes τ' of τ such that, for each position $p \in \mathit{leaf}(\tau')$, the sub-tree of τ at p either belongs to S or it is a leaf of τ. Intuitively, $\tau - S$ are all prefixes of τ obtained from τ by removing some of the sub-trees in S. The derivative of a set of trees $T \subseteq \Sigma^{\text{木}}$ wrt S is the set $\bigcup_{\tau \in T}(\tau - S)$.

A (binary) *tree automaton* (TA) over an alphabet Σ is a quadruple $\mathcal{A} = (Q, \delta, I, R)$ where Q is a finite set of *states*, $\delta : Q^2 \times \Sigma \to 2^Q$ is a *transition function*, $I \subseteq Q$ is a set of *leaf states*, and $R \subseteq Q$ is a set of *root* states. We use $(q, r) \text{-}\{a\}\!\mapsto s$ to denote that $s \in \delta((q, r), a)$. A *run* of $\mathcal{A}$ on a tree τ is a total map $\rho : \mathrm{dom}(\tau) \to Q$ such that if $\tau(p) = \bot$, then $\rho(p) \in I$, else $(\rho(p.\mathtt{L}), \rho(p.\mathtt{R})) \text{-}\{a\}\!\mapsto \rho(p)$ with $a = \tau(p)$. The run ρ is *accepting* if $\rho(\epsilon) \in R$, and the *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all trees on which $\mathcal{A}$ has an accepting run. $\mathcal{A}$ is *deterministic* if $|I| = 1$ and $\forall q, r \in Q, a \in \Sigma : |\delta((q, r), a)| \leq 1$, and *complete* if $I \geq 1$ and $\forall q, r \in Q, a \in \Sigma : |\delta((q, r), a)| \geq 1$. Last, for $a \in \Sigma$, we shorten $\delta((q, r), a)$ as $\delta_a(q, r)$, and we use $\delta_\Gamma(q, r)$ to denote $\bigcup\{\delta_a(q, r) \mid a \in \Gamma\}$ for a set $\Gamma \subseteq \Sigma$.

Syntax and Semantics of WS2S. WS2S is a logic that allows quantification over second-order *variables*, which are denoted by upper-case letters $X, Y, \ldots$ and range over *finite sets* of tree positions in $\{\mathtt{L}, \mathtt{R}\}^*$ (the finiteness of variable assignments is reflected in the name *weak*). See Fig. 1a for an example of a set of positions assigned to a variable. Atomic formulae (atoms) of WS2S are of the form: (i) $X \subseteq Y$, (ii) $X = S_{\mathtt{L}}(Y)$, and (iii) $X = S_{\mathtt{R}}(Y)$. Formulae are constructed from atoms using the logical connectives $\wedge, \neg$, and the quantifier $\exists \mathbb{X}$ where $\mathbb{X}$

[1] Intuitively, the $[\cdot]^{\text{木}}$ operator can be seen as a generalization of the Kleene star to tree languages. The symbol 木 is the Chinese character for a tree, pronounced *mù*, as in English *moo-n*, but shorter and with a falling tone, staccato-like.



(a) Positions assigned to the variable X.

(b) Encoding of ν into a tree τ_ν; a node at a position p has the value $\boxed{x \mid y}$ where $x = 1$ iff $\tau_\nu(p)$ maps X to 1 and $y = 1$ iff $\tau_\nu(p)$ maps Y to 1.

Fig. 1. An example of an assignment ν to a pair of variables $\{X, Y\}$ s.t. $\nu(X) = \{\mathtt{LR}, \mathtt{R}, \mathtt{RLR}, \mathtt{RR}\}$ and $\nu(Y) = \{\epsilon, \mathtt{L}, \mathtt{LL}, \mathtt{R}, \mathtt{RR}\}$ and its encoding into a tree.

is a finite set of variables (we write $\exists X$ when $\mathbb{X}$ is a singleton set $\{X\}$). Other connectives (such as $\vee$ or $\forall$) and predicates (such as the predicate $\mathrm{Sing}(X)$ for a singleton set X) can be obtained as syntactic sugar.

A *model* of a WS2S formula $\varphi(\mathbb{X})$ with the set of free variables $\mathbb{X}$ is an assignment $\nu : \mathbb{X} \to 2^{\{\mathtt{L},\mathtt{R}\}^*}$ of the free variables of φ to finite subsets of $\{\mathtt{L}, \mathtt{R}\}^*$ for which the formula is *satisfied*, written $\nu \models \varphi$. Satisfaction of atomic formulae is defined as follows: (i) $\nu \models X \subseteq Y$ iff $\nu(X) \subseteq \nu(Y)$, (ii) $\nu \models X = S_{\mathtt{L}}(Y)$ iff $\nu(X) = \{p.\mathtt{L} \mid p \in \nu(Y)\}$, and (iii) $\nu \models X = S_{\mathtt{R}}(Y)$ iff $\nu(X) = \{p.\mathtt{R} \mid p \in \nu(Y)\}$. Informally, the $S_{\mathtt{L}}(Y)$ function returns all positions from Y shifted to their left child and the $S_{\mathtt{R}}(Y)$ function returns all positions from Y shifted to their right child. Satisfaction of formulae built using Boolean connectives and the quantifier is defined as usual. A formula φ is *valid*, written $\models \varphi$, iff all assignments of its free variables are its models, and *satisfiable* if it has a model. Wlog, we assume that each variable in a formula either has only free occurrences or is quantified exactly once; we denote the set of (free and quantified) variables occurring in a formula φ as $\mathit{Vars}(\varphi)$.

Representing Models as Trees. We fix a formula φ with variables $\mathit{Vars}(\varphi) = \mathbb{X}$. A *symbol* ξ over $\mathbb{X}$ is a (total) function $\xi : \mathbb{X} \to \{0, 1\}$, e.g., $\xi = \{X \mapsto 0, Y \mapsto 1\}$ is a symbol over $\mathbb{X} = \{X, Y\}$. We use $\Sigma_{\mathbb{X}}$ to denote the set of all symbols over $\mathbb{X}$ and $\vec{0}$ to denote the symbol mapping all variables in $\mathbb{X}$ to 0, i.e., $\vec{0} = \{X \mapsto 0 \mid X \in \mathbb{X}\}$.

A finite assignment $\nu : \mathbb{X} \to 2^{\{\mathtt{L},\mathtt{R}\}^*}$ of φ's variables can be encoded as a finite tree τ_ν of symbols over $\mathbb{X}$ where every position $p \in \{\mathtt{L}, \mathtt{R}\}^*$ satisfies the following conditions: (a) if $p \in \nu(X)$, then $\tau_\nu(p)$ contains $\{X \mapsto 1\}$, and (b) if $p \notin \nu(X)$, then either $\tau_\nu(p)$ contains $\{X \mapsto 0\}$ or $\tau_\nu(p) = \bot$ (note that the occurrences of $\bot$ in τ are limited since τ still needs to be a tree). Observe that ν can have multiple encodings: the unique minimum one τ_ν^{min} and (infinitely many)

extensions of τ_ν^{min} with $\vec{0}$-only trees. The *language* of φ is defined as the set of all encodings of its models $\mathcal{L}(\varphi) = \{\tau_\nu \in \Sigma_\mathbb{X}^\circledast \mid \nu \models \varphi \text{and } \tau_\nu \text{ is an encoding of } \nu\}$.

Let ξ be a symbol over $\mathbb{X}$. For a set of variables $\mathbb{Y} \subseteq \mathbb{X}$, we define the *projection* of ξ wrt $\mathbb{Y}$ as the set of symbols $\pi_\mathbb{Y}(\xi) = \{\xi' \in \Sigma_\mathbb{X} \mid \xi_{|\mathbb{X}\setminus\mathbb{Y}} \subseteq \xi'\}$. Intuitively, the projection removes the original assignments of variables from $\mathbb{Y}$ and allows them to be substituted by any possible value. We define $\pi_\mathbb{Y}(\bot) = \bot$ and write π_Y if $\mathbb{Y}$ is a singleton set $\{Y\}$. As an example, for $\mathbb{X} = \{X, Y\}$ the projection of $\vec{0}$ wrt $\{X\}$ is given as $\pi_X(\vec{0}) = \{\{X \mapsto 0, Y \mapsto 0\}, \{X \mapsto 1, Y \mapsto 0\}\}$.[2] The definition of projection can be extended to trees τ over $\Sigma_\mathbb{X}$ so that $\pi_\mathbb{Y}(\tau)$ is the set of trees $\{\tau' \in \Sigma_\mathbb{X}^\circledast \mid \forall p \in \text{pos}(\tau) : \text{if } \tau(p) = \bot, \text{ then } \tau'(p) = \bot, \text{ else } \tau'(p) \in \pi_\mathbb{Y}(\tau(p))\}$ and subsequently to languages L so that $\pi_\mathbb{Y}(L) = \bigcup\{\pi_\mathbb{Y}(\tau) \mid \tau \in L\}$.

The Classical Decision Procedure for WS2S. The classical decision procedure for the WS2S logic goes through a direct construction of a TA $\mathcal{A}_\varphi$ having the same language as a given formula φ. Let us briefly recall the automata constructions used (cf. [31]). Given a complete TA $\mathcal{A} = (Q, \delta, I, R)$, the *complement* assumes that $\mathcal{A}$ is deterministic and returns $\mathcal{A}^\complement = (Q, \delta, I, Q \setminus R)$, the projection returns $\pi_X(\mathcal{A}) = (Q, \delta^{\pi_X}, I, R)$ with $\delta_a^{\pi_X}(q, r) = \delta_{\pi_X(a)}(q, r)$, and the *subset construction* returns the deterministic and complete automaton $\mathcal{A}^\mathcal{D} = (2^Q, \delta^\mathcal{D}, \{I\}, R^\mathcal{D})$ where $\delta_a^\mathcal{D}(S, S') = \bigcup_{q \in S, q' \in S'} \delta_a(q, q')$ and $R^\mathcal{D} = \{S \subseteq Q \mid S \cap R \neq \emptyset\}$. The binary operators $\circ \in \{\cup, \cap\}$ are implemented through a *product construction*, which—given the TA $\mathcal{A}$ and another complete TA $\mathcal{A}' = (Q', \delta', I', R')$—returns the automaton $\mathcal{A} \circ \mathcal{A}' = (Q \times Q', \Delta^\times, I^\times, R^\circ)$ where $\Delta_a^\times((q, r), (q', r')) = \Delta_a(q, q') \times \Delta'_a(r, r')$, $I^\times = I \times I'$, and for $(q, r) \in Q \times Q'$, $(q, r) \in R^\cap \Leftrightarrow q \in R \wedge r \in R'$ and $(q, r) \in R^\cup \Leftrightarrow q \in R \vee r \in R'$. The language non-emptiness test can be implemented through the equivalence $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff $\mathit{reach}_\delta(I) \cap R \neq \emptyset$ where the set $\mathit{reach}_\delta(S)$ of states *reachable* from a set $S \subseteq Q$ through δ-transitions is computed as the least fixpoint

$$\mathit{reach}_\delta(S) = \mu Z.\ S \cup \bigcup_{q,r \in Z} \delta(q, r). \tag{1}$$

The same fixpoint computation is used to compute the derivative wrt $a^\circledast$ for some $a \in \Sigma$ as $\mathcal{A} - a^\circledast = (Q, \delta, \mathit{reach}_{\delta_a}(I), R)$: the new leaf states are all those reachable from I through a-transitions.

The classical WSkS decision procedure uses the above operations to constructs the automaton $\mathcal{A}_\varphi$ inductively to the structure of φ as follows: (i) If φ is an atomic formula, then $\mathcal{A}_\varphi$ is a pre-defined *base* TA over $\Sigma_\mathbb{X}$ (the particular base automata for our atomic predicates can be found, e.g., in [31], and we list them also in [30]). (ii) If $\varphi = \varphi_1 \wedge \varphi_2$, then $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cap \mathcal{A}_{\varphi_2}$. (iii) If $\varphi = \varphi_1 \vee \varphi_2$, then $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cup \mathcal{A}_{\varphi_2}$. (iv) If $\varphi = \neg\psi$, then $\mathcal{A}_\varphi = \mathcal{A}_\psi^\complement$. (v) Finally, if $\varphi = \exists X.\ \psi$, then $\mathcal{A}_\varphi = (\pi_X(\mathcal{A}_\psi))^\mathcal{D} - \vec{0}^\circledast$.

[2] Note that our definition of projection differs from the usual one, which would in the example produce a single symbol $\{Y \mapsto 0\}$ over a different alphabet (the alphabet of symbols over $\{Y\}$).

Points (i) to (iv) are self-explanatory. In point (v), the projection implements the quantification by forgetting the values of the X component of all symbols. Since this yields non-determinism, projection is followed by determinisation by the subset construction. Further, the projection can produce some new trees that contain $\vec{0}$-only labelled sub-trees, which need not be present in some smaller encodings of the same model. Consider, for example, a formula ψ having the language $\mathcal{L}(\psi)$ given by the tree τ_ν in Fig. 1b and all its $\vec{0}$-extensions. To obtain $\mathcal{L}(\exists X.\psi)$, it is not sufficient to make the projection $\pi_X(\mathcal{L}(\psi))$ because the projected language does not contain the minimum encoding τ_ν^{min} of $\nu : Y \mapsto \{\epsilon, \mathtt{L}, \mathtt{LL}, \mathtt{R}, \mathtt{RR}\}$, but only those encodings ν' such that $\nu'(\mathtt{RLR}) = \{Y \mapsto 0\}$. Therefore, the $\vec{0}$-derivative is needed to saturate the language with *all* encodings of the encoded models (if some of these encodings were missing, the inductive construction could produce a wrong result, for instance, if the language were subsequently complemented). Note that the same effect can be achieved by replacing the set of leaf states I of $\mathcal{A}_\varphi$ by $reach_{\Delta_{\vec{0}}}(I)$ where Δ is the transition function of $\mathcal{A}_\varphi$. See [31] for more details.

3 Automata Terms

Our algorithm for deciding WS2S may be seen as an alternative implementation of the classical procedure from Sect. 2. The main innovation is the data structure of *automata terms*, which implicitly represent the automata constructed by the automata operations. Unlike the classical procedure—which proceeds by a bottom-up traversal on the formula structure, building an automaton for each sub-formula before proceeding upwards—automata terms allow for constructing parts of automata at higher levels from parts of automata on the lower levels even though the construction of the lower level automata has not yet finished. This allows one to test the language emptiness on the fly and use techniques of state space pruning, which will be discussed later in Sect. 4.

Syntax of Automata Terms. Terms are created according to the grammar in Fig. 2 starting from states $q \in Q_i$, denoted as *atomic states*, of a given finite set of *base automata* $\mathcal{B}_i = (Q_i, \delta_i, I_i, R_i)$ with pairwise disjoint sets of states. For simplicity, we assume that the base automata are complete, and we denote by $\mathcal{B} = (Q^{\mathcal{B}}, \delta^{\mathcal{B}}, I^{\mathcal{B}}, R^{\mathcal{B}})$ their component-wise union. *Automata terms* A specify the set of leaf states of an automaton. *Set terms* S list a finite number of the leaf

$$
\begin{aligned}
A &::= S \mid D && \textit{(automata term)}\\
S &::= \{t, \ldots, t\} && \textit{(set term)}\\
D &::= S - \vec{0}^{\star} && \textit{(derivative term)}\\
t &::= q \mid t + t \mid t \,\&\, t \mid \\
&\quad\ \ \overline{t} \mid \pi_X(t) \mid S \mid D && \textit{(state term)}
\end{aligned}
$$

Fig. 2. Syntax of terms.

states explicitly, while *derivative terms* D specify them symbolically as states reachable from a set of states S via $\vec{0}$s. The states themselves are represented by *state terms* t (notice that set terms S and derivate terms D can both be automata and state terms). Intuitively, the structure of state terms records the automata constructions used to create the top-level automaton from states of the base automata. Non-leaf state terms, the state terms' transition function, and root state terms are then defined inductively from base automata as described below in detail. We will normally use t, u to denote terms of all types (unless the type of the term needs to be emphasized).

Example 1. Consider a formula $\varphi \equiv \neg\exists X.\ \mathrm{Sing}(X) \wedge X = \{\epsilon\}$ and its corresponding automata term $t_\varphi = \left\{ \overline{\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^*} \right\}$ (we will show how t_φ was obtained from φ later). For the sake of presentation, we will consider the following base automata for the predicates $\mathrm{Sing}(X)$ and $X = \{\epsilon\}$: $\mathcal{A}_{\mathrm{Sing}(X)} = (\{q_0, q_1, q_s\}, \delta, \{q_0\}, \{q_1\})$ and $\mathcal{A}_{X=\{\epsilon\}} = (\{p_0, p_1, p_s\}, \delta', \{p_0\}, \{p_1\})$ where δ and δ' have the following sets of transitions (transitions not defined below go to the sink states q_s and p_s, respectively):

$$\delta : (q_0, q_0) \text{-}\{\{X \mapsto 0\}\}\!\to q_0,\ (q_0, q_1) \text{-}\{\{X \mapsto 0\}\}\!\to q_1,\ (q_0, q_0) \text{-}\{\{X \mapsto 1\}\}\!\to q_1,\ (q_1, q_0) \text{-}\{\{X \mapsto 0\}\}\!\to q_1$$

$$\delta' : (p_0, p_0) \text{-}\{\{X \mapsto 0\}\}\!\to p_0,\ (p_0, p_0) \text{-}\{\{X \mapsto 1\}\}\!\to p_1.$$

The term t_φ denotes the TA $\left((\pi_X(\mathcal{A}_{\mathrm{Sing}(X)} \cap \mathcal{A}_{X=\{\epsilon\}}) - \vec{0}^*)^{\mathcal{D}}\right)^{\complement}$ constructed by intersection, projection, derivative, subset construction, and complement. □

Semantics of Terms. We will define the denotation of an automata term t as the automaton $\mathcal{A}_t = (Q, \Delta, I, R)$. For a set automata term $t = S$, we define $I = S$, $Q = \mathit{reach}_\Delta(S)$ (i.e., Q is the set of state terms reachable from the leaf state terms), and Δ and R are defined inductively to the structure of t. Particularly, R contains the terms of Q that satisfy the predicate $\mathcal{R}$ defined in Fig. 3, and Δ is defined in Fig. 4, with the addition that whenever the rules in Fig. 4 do not apply, then we let $\Delta_a(t, t') = \{\emptyset\}$. The $\emptyset$ here is used as a universal sink state in order to maintain Δ complete, which is needed for automata terms representing complements to yield the expected language.

$$\mathcal{R}(t + u) \Leftrightarrow \mathcal{R}(t) \vee \mathcal{R}(u) \quad (2)$$
$$\mathcal{R}(t \,\&\, u) \Leftrightarrow \mathcal{R}(t) \wedge \mathcal{R}(u) \quad (3)$$
$$\mathcal{R}(\pi_X(t)) \Leftrightarrow \mathcal{R}(t) \quad (4)$$
$$\mathcal{R}(\overline{t}) \Leftrightarrow \neg\mathcal{R}(t) \quad (5)$$
$$\mathcal{R}(S) \Leftrightarrow \exists t \in S.\, \mathcal{R}(t) \quad (6)$$
$$\mathcal{R}(q) \Leftrightarrow q \in R^{\mathcal{B}} \quad (7)$$

Fig. 3. Root term states.

$$\Delta_a(t+u,t'+u') = \Delta_a(t,t')\,[+]\,\Delta_a(u,u') \quad (8)$$

$$\Delta_a(t\,\&\,u,t'\,\&\,u') = \Delta_a(t,t')\,[\&]\,\Delta_a(u,u') \quad (9)$$

$$\Delta_a(\pi_X(t),\pi_X(t')) = \{\pi_X(u) \mid u \in \Delta_{\pi_X(a)}(t,t')\} \quad (10)$$

$$\Delta_a(\overline{t},\overline{t'}) = \left\{\overline{u} \mid u \in \Delta_a(t,t')\right\} \quad (11)$$

$$\Delta_a(S,S') = \left\{ \bigcup_{t\in S, t'\in S'} \Delta_a(t,t') \right\} \quad (12)$$

$$\Delta_a(q,r) = \delta_a^{\mathcal{B}}(q,r) \quad (13)$$

Fig. 4. Transitions among compatible state terms.

The transitions of Δ for terms of the type $+$, $\&$, π_X, $\overline{\cdot}$, and S are built from the transition function of their subterms analogously to how the automata operations of the product union, product intersection, projection, complement, and subset construction, respectively, build the transition function from the transition functions of their arguments (cf. Sect. 2). The only difference is that the state terms stay *annotated* with the particular operation by which they were made (the annotation of the set state terms are the set brackets). The root states are also defined analogously as in the classical constructions. In Figs. 3 and 4, the terms t, t', u, u' are arbitrary terms, S, S' are set terms, and $q, r \in Q^{\mathcal{B}}$.

Finally, we complete the definition of the term semantics by adding the definition of semantics for the derivative term $S - \vec{0}^{\circledast}$. This term is a symbolic representation of the set term that contains all state terms upward-reachable from S in $\mathcal{A}_S$ over $\vec{0}$. Formally, we first define the so-called *saturation* of $\mathcal{A}_S$ as

$$(S - \vec{0}^{\circledast})^{\mathsf{s}} = reach_{\Delta_{\vec{0}}}(S) \quad (14)$$

(with $reach_{\Delta_{\vec{0}}}(S)$ defined as the fixpoint (1)), and we complete the definition of Δ and $\mathcal{R}$ in Figs. 3 and 4 with three new rules to be used with a derivative term D:

$$\Delta_a(D,u) = \Delta_a(D^{\mathsf{s}},u) \quad (15) \qquad \Delta_a(u,D) = \Delta_a(u,D^{\mathsf{s}}) \quad (16) \qquad \mathcal{R}(D) \Leftrightarrow \mathcal{R}(D^{\mathsf{s}}) \quad (17)$$

The automaton $\mathcal{A}_D$ then equals $\mathcal{A}_{D^{\mathsf{s}}}$, i.e., the semantics of a derivative term is defined by its saturation.

Example 2. Let us consider a derivative term $t = \{\pi_X(\{q_0\}\,\&\{p_0\})\} - \vec{0}^{\circledast}$, which occurs within the nested automata term t_φ of Example 1. The set term representing all terms reachable upward from t is then the term

$$t^{\mathsf{s}} = \{\pi_X(\{q_0\}\,\&\{p_0\}), \pi_X(\{q_1\}\,\&\{p_1\}), \pi_X(\{q_s\}\,\&\{p_s\}), \\ \pi_X(\{q_1\}\,\&\{p_s\}), \pi_X(\{q_0\}\,\&\{p_s\})\}.$$

The semantics of t is therefore the TA $\mathcal{A}_t$ with the set of states given by t^{s}. □

Properties of Terms. An implication of the definitions above, essential for termination of our algorithm in Sect. 4, is that the automata represented by the terms indeed have finitely many states. This is the direct consequence of Lemma 1.

Lemma 1. *The size of $reach_{\Delta}(t)$ is finite for any automata term t.*

Intuitively, the terms are built over a finite set of states $Q^{\mathcal{B}}$, they are finitely branching, and the transition function on terms does not increase their depth.

Let us further denote by $\mathcal{L}(t)$ the language $\mathcal{L}(\mathcal{A}_t)$ of the automaton induced by a term t. Lemma 2 below shows that languages of terms can be defined from the languages of their sub-terms if the sub-terms are set terms of derivative terms. The terms on the left-hand sides are implicit representations of the automata operations of the respective language operators on the right-hand sides. The main reason why the lemma cannot be extended to all types of sub-terms and yield an inductive definition of term languages is that it is not meaningful to talk about the bottom-up language of an isolated state term that is neither a set term nor a derivative term (which both are also automata terms). This is also one of the main differences from [29] where every term has its own language, which makes the reasoning and the correctness proofs in the current paper significantly more involved.

Lemma 2. *For automata terms A_1, A_2 and a set term S, the following holds:*

$$\mathcal{L}(\{A_1\}) = \mathcal{L}(A_1) \quad (a) \qquad \mathcal{L}(\{\overline{A_1}\}) = \overline{\mathcal{L}(A_1)} \quad (d)$$
$$\mathcal{L}(\{A_1 + A_2\}) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2) \quad (b) \qquad \mathcal{L}(\{\pi_X(A_1)\}) = \pi_X(\mathcal{L}(A_1)) \quad (e)$$
$$\mathcal{L}(\{A_1 \& A_2\}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \quad (c) \qquad \mathcal{L}(S - \vec{0}^{\circledast}) = \mathcal{L}(S) - \vec{0}^{\circledast} \quad (f)$$

Terms of Formulae. Our algorithm in Sect. 4 will translate a WS2S formula φ into the automata term $t_\varphi = \{\langle\varphi\rangle\}$ representing a deterministic automaton with its only leaf state represented by the state term $\langle\varphi\rangle$. The base automata of t_φ include the automaton $\mathcal{A}_{\varphi_{atom}}$ for each atomic predicate φ_{atom} used in φ. The state term $\langle\varphi\rangle$ is then defined inductively to the structure of φ as shown in Fig. 5. In the definition, φ_0 is an atomic predicate, I_{φ_0} is the set of leaf states of $\mathcal{A}_{\varphi_{atom}}$, and φ and ψ denote arbitrary WS2S formulae. We note that the translation rules may create sub-terms of the form $\{\{t\}\}$, i.e., with nested set brackets. Since $\{\cdot\}$ semantically means determinisation by subset construction, such double determinisation terms can be always simplified to $\{t\}$ (cf. Lemma 2a). See Example 1 for a formula φ and its corresponding term t_φ. Theorem 1 establishes the correctness of the formula to term translation.

$$\langle\varphi_0\rangle = I_{\varphi_0} \quad (18)$$
$$\langle\varphi \wedge \psi\rangle = \langle\varphi\rangle \,\&\, \langle\psi\rangle \quad (19)$$
$$\langle\varphi \vee \psi\rangle = \langle\varphi\rangle + \langle\psi\rangle \quad (20)$$
$$\langle\neg\varphi\rangle = \overline{\langle\varphi\rangle} \quad (21)$$
$$\langle\exists X.\ \varphi\rangle = \{\pi_X(\langle\varphi\rangle)\} - \vec{0}^{\circledast} \quad (22)$$

Fig. 5. From formulae to state-terms.

Theorem 1. *Let φ be a WS2S formula. Then $\mathcal{L}(\varphi) = \mathcal{L}(t_\varphi)$.*

The proof of Theorem 1 uses structural induction, which is greatly simplified by Lemma 2, but since Lemma 2 does not (and cannot, as discussed above) cover all used types of terms, the induction step must in some cases still rely on reasoning about the definition of the transition relation on terms.

4 An Efficient Decision Procedure

The development in Sect. 3 already implies a naïve automata term-based satisfiability check. Namely, by Theorem 1, we know that a formula φ is satisfiable iff $\mathcal{L}(\mathcal{A}_{t_\varphi}) \neq \emptyset$. After translating φ into t_φ using rules (18)–(22), we may use the definitions of the transition function and root states of $\mathcal{A}_{t_\varphi} = (Q, \Delta, I, F)$ in Sect. 3 to decide the language emptiness through evaluating the root state test $\mathcal{R}(reach_\Delta(I))$. It is enough to implement the equalities and equivalences (8)–(17) as recursive functions. We will further refer to this algorithm as the *simple recursion*. The evaluation of $reach_\Delta(I)$ induces nested evaluations of the fixpoint (14): the one on the top level of the language emptiness test and another one for every expansion of a derivative sub-term. The termination of these fixpoint computations is guaranteed due to Lemma 1.

Such a naïve implementation is, however, inefficient and has only disadvantages in comparison to the classical decision procedure. In this section, we will discuss how it can be optimized. Besides an essential *memoization* needed to implement the recursion efficiently, we will show that the automata term representation is amenable to optimizations that cannot be used in the classical construction. These are techniques of state space pruning: the fact that the emptiness can be tested on the fly during the automata construction allows one to avoid exploration of state space irrelevant to the test. The pruning is done through the techniques of *lazy evaluation* and *subsumption*. We will also discuss an optimization of the transition function of Sect. 3 through *product flattening*, which is an analogy to standard implementations of automata intersection.

4.1 Memoization

The simple recursion repeats the fixpoint computations that saturate derivative terms from scratch at every call of the transition function or root test. This is easily countered through *memoization*, known, e.g., from compilers of functional languages, which caches results of function calls in order to avoid their re-evaluation. Namely, after saturating a derivative sub-term $t = S - \vec{0}^{\circledast}$ of t_φ for the first time, we simply *replace* t in t_φ by the saturation $t^{\mathbf{s}} = reach_{\Delta_{\vec{0}}}(S)$. Since a derivative is a symbolic representation of its saturated version, the replacement does not change the language of t_φ. Using memoization, every fixpoint computation is then carried out once only.

4.2 Lazy Evaluation

The *lazy* variant of the procedure uses *short-circuiting* to optimize connectives $\wedge$ and $\vee$, and *early termination* to optimize fixpoint computation in derivative saturations. Namely, assume that we have a term $t_1 + t_2$ and that we test whether $\mathcal{R}(t_1 + t_2)$. Suppose that we establish that $\mathcal{R}(t_1)$; we can *short circuit* the evaluation and immediately return *true*, completely avoiding touching the potentially complex term t_2 (and analogously for a term of the form $t_1 \,\&\, t_2$ when one branch is *false*).

Furthermore, *early termination* is used to optimize fixpoint computations used to saturate derivatives within tests $\mathcal{R}(S - \vec{0}^{\circledast})$ (obtained from sub-formulae such as $\exists X.\ \psi$). Namely, instead of first unfolding the whole fixpoint into a set $\{t_1, \ldots t_n\}$ and only then testing whether $\mathcal{R}(t_i)$ is true for some t_i, the terms t_i can be tested as soon as they are computed, and the fixpoint computation can be stopped early, immediately when the test succeeds on one of them. Then, instead of replacing the derivative sub-term by its full saturation, we replace it by the partial result $\{t_1, \ldots, t_i\} - \vec{0}^{\circledast}$ for $i \leq n$.

Finishing the evaluation of the fixpoint computation might later be required in order to compute a transition from the derivative. We note that this corresponds to the concept of *continuations* from functional programming, used to represent a paused computation that may be required to continue later.

Example 3. Let us now illustrate the lazy decision procedure on our running example formula $\varphi \equiv \neg\exists X.\ \mathrm{Sing}(X) \wedge X = \{\epsilon\}$ and the corresponding automata term $t_\varphi = \{\, \overline{\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^{\,\circledast}} \,\}$ from Example 1. The task of the procedure is to compute the value of $\mathcal{R}(\mathit{reach}_\Delta(t_\varphi))$, i.e., whether there is a root state reachable from the leaf state $\langle\varphi\rangle$ of $\mathcal{A}_{t_\varphi}$. The fact that φ is ground allows us to slightly simplify the problem because any ground formula ψ is satisfiable iff $\bot \in \mathcal{L}(\psi)$, i.e., iff the leaf state $\langle\psi\rangle$ of $\mathcal{A}_{t_\psi}$ is also a root. It is thus enough to test $\mathcal{R}(\langle\varphi\rangle)$ where $\langle\varphi\rangle = \overline{\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^{\,\circledast}}$.

The computation proceeds as follows. First, we use (5) from Fig. 3 to propagate the root test towards the derivative, i.e., to obtain that $\mathcal{R}(\langle\varphi\rangle)$ iff $\neg\mathcal{R}(\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^{\,\circledast})$. Since the $\mathcal{R}$-test cannot be directly evaluated on a derivative term, we need to start saturating it into a set term, evaluating $\mathcal{R}$ on the fly, hoping for early termination. We begin with evaluating the $\mathcal{R}$-test on the initial element $t_0 = \pi_X(\{q_0\}\{p_0\})$ of the set. The test propagates through the projection π_X due to (4) and evaluates as *false* on the left conjunct (through, in order, (3), (6), and (7) since the state q_0 is not a root state. As a trivial example of short circuiting, we can skip evaluating $\mathcal{R}$ on the right conjunct $\{p_0\}$ and conclude that $\mathcal{R}(t_0)$ is *false*.

The fixpoint computation then continues with the first iteration, computing the $\vec{0}$-successors of the set $\{t_0\}$. We will obtain $\Delta_{\vec{0}}(t_0, t_0) = \{t_0, t_1\}$ with $t_1 = \pi_X(\{q_1\} \,\&\, \{p_1\})$. The test $\mathcal{R}(t_1)$ now returns *true* because both q_1 and p_1 are root states. With that, the fixpoint computation may terminate early, with the $\mathcal{R}$-test on the derivative sub-term returning *true*. Memoization then replaces the derivative sub-term in $\langle\varphi\rangle$ by the partially evaluated version $\{t_0, t_1\} - \vec{0}^{\,\circledast}$, and $\mathcal{R}(\langle\varphi\rangle)$ is evaluated as *false* due to (5). We therefore conclude that φ is unsatisfiable (and invalid since it is ground). □

4.3 Subsumption

The next technique we use is based on pruning out parts of a search space that are *subsumed* by other parts. In particular, we generalize (in a similar way as we did for WS1S in our previous work [29]) the concept used in *antichain* algorithms for efficiently deciding language inclusion and universality of finite word and tree automata [22–25]. Although the problems are in general computationally infeasible (they are PSPACE-complete for finite word automata and EXPTIME-complete for finite tree automata), antichain algorithms can solve them efficiently in many practical cases.

We apply the technique by keeping set terms in the form of antichains of *simulation-maximal* elements and prune out any other simulation-smaller elements. Intuitively, the notion of a term t being simulation-smaller than t' implies that trees that might be generated from the leaf states $T \cup \{t\}$ can be generated from $T \cup \{t'\}$ too, hence discarding t does not hurt. Formally, we introduce the following rewriting rule:

$$\{t_1, t_2, \ldots, t_n\} \rightsquigarrow \{t_2, \ldots, t_n\} \qquad \text{for } t_1 \sqsubseteq t_2, \tag{23}$$

which may be used to simplify set sub-terms of automata terms. The rule (23) is applied after every iteration of the fixpoint computation on the current partial result.

Hence the sequence of partial results is monotone, which, together with the finiteness of $reach_\Delta(t)$, guarantees termination. The *subsumption* relation $\sqsubseteq$ used in the rule is defined in Fig. 6 where $S \sqsubseteq^{\forall\exists} S'$ denotes $\forall t \in S\ \exists t' \in S'.\, t \sqsubseteq t'$. Intuitively, on base TAs, subsumption corresponds to inclusion of the set terms (the left disjunct of (24). This clearly has the intended outcome: a larger set of states can always simulate a smaller set in accepting a tree. The rest of the definition is an inductive extension of the base case. It can be shown that $\sqsubseteq$ for any automata term t is an upward simulation on $\mathcal{A}_t$ in the sense of [25]. Consequently, rewriting sub-terms in an automata term according to the new rule (23) does not change its language. Moreover, the fixpoint computation interleaved with application of rule (23) terminates.

$$S \sqsubseteq S' \quad \Leftrightarrow S \subseteq S' \vee S \sqsubseteq^{\forall\exists} S' \tag{24}$$

$$t \,\&\, u \sqsubseteq t' \,\&\, u' \Leftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u' \tag{25}$$

$$t + u \sqsubseteq t' + u' \Leftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u' \tag{26}$$

$$\bar{t} \sqsubseteq \overline{t'} \quad \Leftrightarrow t' \sqsubseteq t \tag{27}$$

$$\pi_X(t) \sqsubseteq \pi_X(t') \Leftrightarrow t \sqsubseteq t' \tag{28}$$

Fig. 6. The subsumption relation $\sqsubseteq$

4.4 Product Flattening

Product flattening is a technique that we use to reduce the size of fixpoint saturations that generate conjunctions and disjunctions of sets as their elements.Consider a term of the form $D = \{\pi_X(S_0 \,\&\, S_0')\} - \vec{0}^{\,\circledast}$ for a pair of sets of terms S_0 and S_0' where the TAs $\mathcal{A}_{S_0}$ and $\mathcal{A}_{S_0'}$ have sets of states Q and Q', respectively.The saturation generates the set $\{\pi_X(S_0 \,\&\, S_0'), \ldots, \pi_X(S_n \,\&\, S_n')\}$ with $S_i \subseteq Q, S_i' \subseteq Q'$ for all $0 \leq i \leq n$. The size of this set is $2^{|Q|\cdot|Q'|}$ in the worst case. In terms of the automata operations, this fixpoint expansion corresponds to first determinizing both $\mathcal{A}_{S_0}$ and $\mathcal{A}_{S_0'}$ and only then using the product construction (cf. Sect. 2). The automata intersection, however, works for nondeterministic automata too—the determinization is not needed. Implementing this standard product construction on terms would mean transforming the original fixpoint above into the following fixpoint with a *flattened product*: $D = \{\pi_X(S\,[\&]\,S')\} - \vec{0}^{\,\circledast}$ where $[\&]$ is the augmented product for conjunction. This way, we can decrease the worst-case size of the fixpoint to $|Q| \cdot |Q'|$. A similar reasoning holds for terms of the form $\{\pi_X(S_0 + S_0')\} - \vec{0}^{\,\circledast}$. Formally, the technique can be implemented by the following pair of sub-term rewriting rules where S and S' are non-empty sets of terms:

$$S + S' \rightsquigarrow S\,[+]\,S', \tag{29}$$

$$S \,\&\, S' \rightsquigarrow S\,[\&]\,S'. \tag{30}$$

Observe that for terms obtained from WS2S formulae using the translation from Sect. 3, the rules are not really helpful as is. Consider, for instance, the term $\{\pi_X(\{r\} \,\&\, \{q\})\} - \vec{0}^{\,\circledast}$ obtained from a formula $\exists X.\, \varphi \wedge \psi$ with φ and ψ being atoms. The term would be, using rule (30), rewritten into the term $\{\pi_X(\{r \,\&\, q\})\} - \vec{0}^{\,\circledast}$. Then, during a subsequent fixpoint computation, we might obtain a fixpoint of the following form: $\{\pi_X(\{r \,\&\, q\}), \pi_X(\{r \,\&\, q, r_1 \,\&\, q_1\}),\ \pi_X(\{r_1 \,\&\, q_1, r_2 \,\&\, q_2\})\}$, where the occurrences

of the projection π_X disallow one to perform the desired union of the inner sets, and so the application of rule (30) did not help. We therefore need to equip our procedure with a rewriting rule that can be used to push the projection inside a set term S:

$$\pi_X(S) \rightsquigarrow \{\pi_X(t) \mid t \in S\}. \tag{31}$$

In the example above, we would now obtain the term $\{\pi_X(r \,\&\, q)\} - \vec{0}^{\star}$ (we rewrote $\{\{\cdot\}\}$ to $\{\cdot\}$ as mentioned in Sect. 3) and the fixpoint $\{\pi_X(r \,\&\, q), \pi_X(r_1 \,\&\, q_1), \pi_X(r_2 \,\&\, q_2)\}$. The correctness of the rules is guaranteed by the following lemma:

Lemma 3. *For sets of terms S, S' s.t. $S \neq \emptyset$, $S' \neq \emptyset$ we have:*

$$\mathcal{L}\left(\{S + S'\}\right) = \mathcal{L}\left(\{S\,[+]\,S'\}\right), \quad (a) \qquad \mathcal{L}\left(\{\pi_X(S)\}\right) = \mathcal{L}\left(\{\pi_X(t) \mid t \in S\}\right). \quad (c)$$
$$\mathcal{L}\left(\{S \& S'\}\right) = \mathcal{L}\left(\{S\,[\&]\,S'\}\right), \quad (b)$$

However, we still have to note that there is a danger related with the rules (29)–(31). Namely, if they are applied to some terms in a partially evaluated fixpoint but not to all, the form of these terms might get different (cf. $\pi_X(\{r \,\&\, q\})$ and $\pi_X(r \,\&\, q)$), and it will not be possible to combine them as source states of TA transitions when computing Δ_a, leading thus to an incorrect result. We resolve the situation such that we apply the rules as a pre-processing step only before we start evaluating the top-level fixpoint, which ensures that all terms will subsequently be generated in a compatible form.

5 Experimental Evaluation

We have implemented the above introduced technique in a prototype tool written in Haskell.[3] The base automata, hard-coded into the tool, were the TAs for the basic predicates from Sect. 2, together with automata for predicates $\mathrm{Sing}(X)$ and $X = \{p\}$ for a variable X and a fixed tree position p. As an optimisation, our tool uses the so-called *antiprenexing* (proposed already in [29]), pushing quantifiers down the formula tree using the standard logical equivalences. Intuitively, antiprenexing reduces the complexity of elements within fixpoints by removing irrelevant parts outside the fixpoint.

We have performed experiments with our tool on various formulae and compared its performance with that of Mona. We applied Mona both on the original form of the considered formulae as well as on their versions obtained by antiprenexing (which is built into our tool and which—as we realised—can significantly help Mona too). Our preliminary implementation of product flattening (cf. Sect. 4.4) is restricted to parts below the lowest fixpoint, and our experiments showed that it does not work well when applied on this level, where the complexity is not too high, so we turned it off for the experiments. We ran all experiments on a 64-bit Linux Debian workstation with the Intel(R) Core(TM) i7-2600 CPU running at 3.40 GHz with 16 GiB of RAM. We used a timeout of 100 s.

We first considered various WS2S formulae on which Mona was successfully applied previously in the literature. On them, our tool is quite slower than Mona, which is not much surprising given the amount of optimisations built into Mona (for instance,

[3] The implementation is available at https://github.com/vhavlena/lazy-wsks.

Table 1. Experimental results over the family of formulae $\varphi_n^{pt} \equiv \forall Z_1, Z_2.\ \exists X_1, \ldots, X_n.\ edge(Z_1, X_1) \wedge \bigwedge_{i=1}^{n} edge(X_i, X_{i+1}) \wedge edge(X_n, Z_2)$ where $edge(X, Y) \equiv edge_{\mathtt{L}}(X, Y) \vee edge_{\mathtt{R}}(X, Y)$ and $edge_{\mathtt{L/R}}(X, Y) \equiv \exists Z.\ Z = S_{\mathtt{L/R}}(X) \wedge Z \subseteq Y$.

	Running time (sec)			# of subterms/states		
n	*Lazy*	*Mona*	*Mona+AP*	*Lazy*	*Mona*	*Mona+AP*
1	0.02	0.16	0.15	149	216	216
2	0.50	-	-	937	-	-
3	0.83	-	-	2487	-	-
4	34.95	-	-	8391	-	-
5	60.94	-	-	23827	-	-

for the benchmarks from [5], MONA on average took 0.1 s, while we timeouted).[4] Next, we identified several parametric families of formulae (adapted from [29]), such as, e.g., $\varphi_n^{horn} \equiv \exists X.\ \forall X_1.\ \exists X_2, \ldots X_n.\ ((X_1 \subseteq X \wedge X_1 \neq X_2) \Rightarrow X_2 \subseteq X) \wedge \ldots \wedge ((X_{n-1} \subseteq X \wedge X_{n-1} \neq X_n) \Rightarrow X_n \subseteq X)$, where our approach finished within 10 ms, while the time of MONA was increasing when increasing the parameter n, going up to 32 s for $n = 14$ and timeouting for $k \geq 15$. It turned out that MONA could, however, easily handle these formulae after antiprenexing, again (slightly) outperforming our tool. Finally, we also identified several parametric families of formulae that MONA could handle only very badly or not at all, even with antiprenexing, while our tool can handle them much better. These formulae are mentioned in the captions of Tables 1, 2 and 3, which give detailed results of the experiments.

Table 2. Experimental results over the family of formulae $\varphi_n^{cnst} \equiv \exists X.\ X = \{(\mathtt{LR})^4\} \wedge X = \{(\mathtt{LR})^n\}$.

	Running time (sec)			# of subterms/states		
n	*Lazy*	*Mona*	*Mona+AP*	*Lazy*	*Mona*	*Mona+AP*
80	14.60	40.07	40.05	1146	27913	27913
90	21.03	64.26	64.20	1286	32308	32308
100	28.57	98.42	98.91	1426	36258	36258
110	38.10	-	-	1566	-	-
120	49.82	-	-	1706	-	-

Particularly, Columns 2–4 give the running times (in seconds) of our tool (denoted *Lazy*), MONA, and MONA with antiprenexing. Columns 5–7 characterize the size of the generated terms and automata. Namely, for our approach, we give the

Table 3. Experiments over the family $\varphi_n^{sub} = \forall X_1, \ldots, X_n\ \exists X.\ \bigwedge_{i=1}^{n-1} X_i \subseteq X \Rightarrow (X_{i+1} = S_{\mathtt{L}}(X) \vee X_{i+1} = S_{\mathtt{R}}(X))$.

	Running time (sec)			# of subterms/states		
n	*Lazy*	*Mona*	*Mona+AP*	*Lazy*	*Mona*	*Mona+AP*
3	0.01	0.00	0.00	140	92	92
4	0.04	34.39	34.47	386	170	170
5	0.24	–	–	981	–	–
6	2.01	–	–	2376	–	–

[4] Building an optimised and overall competitive implementation is a subject of our further work. Our results with an implementation of a lazy decision procedure for WS1S from [29] suggest that this is possible.

number of nodes in the final term tree (with the leaves being states of the base TAs). For MONA, we give the sum of the numbers of states of all the minimal deterministic TAs constructed by MONA when evaluating the formula. The "–" sign means a timeout or memory shortage.

The formulae considered in Tables 1, 2 and 3 speak about various paths in trees. We were originally inspired by formulae kindly provided by Josh Berdine, which arose from attempts to translate separation logic formulae to WS2S (and use MONA to discharge them), which are beyond the capabilities of MONA (even with antiprenexing). We were also unable to handle them with our tool, but our experimental results on the tree path formulae indicate (despite the prototypical implementation) that our techniques can help one to handle some complex graph formulae that are out of the capabilities of MONA. Thus, they provide a new line of attack on deciding hard WS2S formulae, complementary to the heuristics used in MONA. Improving the techniques and combining them with the classical approach of MONA is a challenging subject for our future work.

6 Related Work

The seminal works [32,33] on the automata-logic connection were the milestones leading to what we call here the classical tree automata-based decision procedure for WSkS [34]. Its non-elementary worst-case complexity was proved in [35], and the work [2] presents the first implementation, restricted to WS1S, with the ambition to use heuristics to counter the high complexity. The authors of [31] provide an excellent survey of the classical results and literature related to WSkS and tree automata.

The tool MONA [3] implements the classical decision procedures for both WS1S and WS2S. It is still the standard tool of choice for deciding WS1S/WSkS formulae due to its all-around most robust performance. The efficiency of MONA stems from many optimizations, both higher-level (such as automata minimization, the encoding of first-order variables used in models, or the use of multi-terminal BDDs to encode the transition function of the automaton) as well as lower-level (e.g. optimizations of hash tables, etc.) [36,37]. The M2L(Str) logic, a dialect of WS1S, can also be decided by a similar automata-based decision procedure, implemented within, e.g., JMOSEL [38] or the symbolic finite automata framework of [39]. In particular, JMOSEL implements several optimizations (such as second-order value numbering [40]) that allow it to outperform MONA on some benchmarks (MONA also provides an M2L(Str) interface on top of the WS1S decision procedure).

The original inspiration for our work are the antichain techniques for checking universality and inclusion of finite automata [22–25] and language emptiness of alternating automata [22], which use symbolic computation together with subsumption to prune large state spaces arising from subset construction. This paper is a continuation of our work on WS1S, which started by [41], where we discussed a basic idea of generalizing the antichain techniques to a WS1S decision procedure. In [29], we then presented a complete WS1S decision procedure based on these ideas that is capable to rival MONA on already interesting benchmarks. The work in [42] presents a decision procedure that, although phrased differently, is in essence fairly similar to that of [29]. This paper generalizes [29] to WS2S. It is not merely a straightforward generalization of the word concepts to trees. A nontrivial transition was needed from language terms of [29], with their semantics being defined straightforwardly from the semantics of sub-terms, to tree automata terms, with the semantics defined as a language of an automaton

with transitions defined inductively to the structure of the term. This change makes the reasoning and correctness proof considerably more complex, though the algorithm itself stays technically quite simple.

Finally, Ganzow and Kaiser [43] developed a new decision procedure for the weak monadic second-order logic on inductive structures within their tool Toss. Their approach completely avoids automata; instead, it is based on the Shelah's composition method. The paper reports that the Toss tool could outperform Mona on two families of WS1S formulae, one derived from Presburger arithmetics and one formula of the form that we mention in our experiments as problematic for Mona but solvable easily by Mona with antiprenexing.

Acknowledgement. We thank the anonymous reviewers for their helpful comments on how to improve the exposition in this paper. This work was supported by the Czech Science Foundation project 17-12465S, the FIT BUT internal project FIT-S-17-4014, and The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science—LQ1602.

References

1. Møller, A., Schwartzbach, M.: The pointer assertion logic engine. In: PLDI 2001. ACM Press (2001). Also in SIGPLAN Notices **36**(5) (2001)
2. Glenn, J., Gasarch, W.: Implementing WS1S via finite automata. In: Raymond, D., Wood, D., Yu, S. (eds.) WIA 1996. LNCS, vol. 1260, pp. 50–63. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63174-7_5
3. Elgaard, J., Klarlund, N., Møller, A.: MONA 1.x: new techniques for WS1S and WS2S. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 516–520. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028773
4. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. http://www.brics.dk/mona/. Revision of BRICS NS-98-3
5. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL 2011, pp. 611–622. ACM (2011)
6. Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using STRAND. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 43–59. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_8
7. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012)
8. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: POPL 2008, 349–361. ACM (2008)
9. Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Trans. Comput. Log. **22**(4), 33 (2013)
10. Zhou, M., He, F., Wang, B., Gu, M., Sun, J.: Array theory of bounded elements and its applications. J. Autom. Reasoning **52**(4), 379–405 (2014)
11. Baukus, K., Bensalem, S., Lakhnech, Y., Stahl, K.: Abstracting WS1S systems to verify parameterized networks. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 188–203. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_14

12. Bodeveix, J.-P., Filali, M.: FMona: a tool for expressing validation techniques over infinite state systems. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 204–219. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_15
13. Bozga, M., Iosif, R., Sifakis, J.: Structural invariants for parametric verification of systems with almost linear architectures. Technical report arXiv:1902.02696 (2019)
14. Klarlund, N., Nielsen, M., Sunesen, K.: A case study in verification based on trace abstractions. In: Broy, M., Merz, S., Spies, K. (eds.) Formal Systems Specification. LNCS, vol. 1169, pp. 341–373. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0024435
15. Smith, M.A., Klarlund, N.: Verification of a sliding window protocol using IOA and MONA. In: Bolognesi, T., Latella, D. (eds.) Formal Methods for Distributed System Development. ITIFIP, vol. 55, pp. 19–34. Springer, Boston, MA (2000). https://doi.org/10.1007/978-0-387-35533-7_2
16. Basin, D., Klarlund, N.: Automata based symbolic reasoning in hardware verification. In: CAV 1998. LNCS, pp. 349–361. Springer (1998)
17. Sandholm, A., Schwartzbach, M.I.: Distributed safety controllers for web services. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 270–284. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053596
18. Hune, T., Sandholm, A.: A case study on using automata in control synthesis. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 349–362. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46428-X_24
19. Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for regular specifications over unbounded domains. In: FMCAD 2010, pp. 101–109. IEEE Computer Science (2010)
20. Morawietz, F., Cornell, T.: The MSO logic-automaton connection in linguistics. In: Lecomte, A., Lamarche, F., Perrier, G. (eds.) LACL 1997. LNCS (LNAI), vol. 1582, pp. 112–131. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48975-4_6
21. Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 476–491. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_36
22. Doyen, L., Raskin, J.-F.: Antichain algorithms for finite automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_2
23. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: a new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_5
24. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70844-5_7
25. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains (on checking language inclusion of NFAs). In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_14
26. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. Formal Methods Syst. Des. **41**(1), 83–106 (2012)

27. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.-F.: Antichains: alternative algorithms for LTL satisfiability and model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_6
28. De Wulf, M., Doyen, L., Raskin, J.-F.: A lattice theory for solving games of imperfect information. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 153–168. Springer, Heidelberg (2006). https://doi.org/10.1007/11730637_14
29. Fiedor, T., Holík, L., Janků, P., Lengál, O., Vojnar, T.: Lazy automata techniques for WS1S. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 407–425. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_24
30. Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Automata terms in a lazy WSkS decision procedure (technical report). Technical report arXiv:1905.08697 (2019)
31. Comon, H., et al.: Tree automata techniques and applications (2008)
32. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: International Congress on Logic, Methodology, and Philosophy of Science, pp. 1–11. Stanford University Press (1962)
33. Rabin, M.O.: Decidability of second order theories and automata on infinite trees. Trans. Am. Math. Soc. **141**, 1–35 (1969)
34. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Math. Syst. Theory **2**(1), 57–81 (1968)
35. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: Fifth Annual ACM Symposium on Theory of Computing, STOC 1973, pp. 1–9. ACM, New York (1973)
36. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. Int. J. Found. Comput. Sci. **13**(4), 571–586 (2002)
37. Klarlund, N.: A theory of restrictions for logics and automata. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 406–417. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_35
38. Topnik, C., Wilhelm, E., Margaria, T., Steffen, B.: jMosel: a stand-alone tool and jABC plugin for M2L(Str). In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 293–298. Springer, Heidelberg (2006). https://doi.org/10.1007/11691617_18
39. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: POPL 2014, pp. 541–554 (2014)
40. Margaria, T., Steffen, B., Topnik, C.: Second-order value numbering. In: GraMoT 2010. Volume 30 of ECEASST, pp. 1–15. EASST (2010)
41. Fiedor, T., Holík, L., Lengál, O., Vojnar, T.: Nested antichains for WS1S. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 658–674. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_59
42. Traytel, D.: A coalgebraic decision procedure for WS1S. In: 24th EACSL Annual Conference on Computer Science Logic (CSL 2015). Volume 41 of Leibniz International Proceedings in Informatics (LIPIcs), pp. 487–503. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015)
43. Ganzow, T., Kaiser, Ł.: New algorithm for weak monadic second-order logic on inductive structures. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 366–380. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15205-4_29