EPiC Series in Computing
Volume 73, 2020, Pages 298–316
LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Antiprenexing for WSkS: A Little Goes a Long Way

Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, Ondřej Valeš, and Tomáš Vojnar

FIT, IT4I Centre of Excellence, Brno University of Technology, Czech Republic

Abstract

We study light-weight techniques for preprocessing of WSkS formulae in an automata-based decision procedure as implemented, e.g., in Mona. The techniques we use are based on *antiprenexing*, i.e., pushing quantifiers deeper into a formula. Intuitively, this tries to alleviate the explosion in the size of the constructed automata by making it happen sooner on smaller automata (and have the automata minimization reduce the output). The formula transformations that we use to implement antiprenexing may, however, be applied in different ways and extent and, if used in an unsuitable way, may also cause an explosion in the size of the formula and the automata built while deciding it. Therefore, our approach uses informed rules that use an estimation of the cost of constructing automata for WSkS formulae. The estimation is based on a model learnt from runs of the decision algorithm on various formulae. An experimental evaluation of our technique shows that antiprenexing can significantly boost the performance of the base WSkS decision procedure, sometimes allowing one to decide formulae that could not be decided before.

1 Introduction

Weak monadic second-order logic of k successors (WSkS) is a logic for describing regular properties of finite k-ary trees. In addition to talking about trees, WSkS can also encode complex properties of a rich class of general graphs by referring to their tree backbones [33]. WSkS offers extreme succinctness at the price of non-elementary worst-case complexity of its satisfaction problem. As noticed first by the authors of [22] 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 [17, 28], 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 [33, 30, 31, 12, 49], string programs [41], array programs [50], parametric systems [5, 8, 10], distributed systems [26, 39], hardware verification [4], automated synthesis [38, 25, 23], and even computational linguistics [34]. Despite these successes, scalability of WSkS solvers is unpredictable. Technology based on solving WSkS is therefore fragile, and users of WSkS are often forced to either find various workarounds, such as in [31], or give up using WSkS altogether—possibly at the price of restricting the input of their approach [46]. Clearly, more research effort needs to be invested to make WSkS solvers truly practical.

Although there have appeared several newer approaches and prototype tools that may beat Mona on restricted sets of formulae [43, 32, 44, 21], including our own approaches [19, 20, 24],

Mona is still the most robust tool and handles by far the largest class of practical formulae. In this work, we focus on further improving the efficiency of Mona. Namely, we elaborate on the preprocessing technique known as *antiprenexing*, which pushes quantifiers deeper into a formula, narrowing their scope. We develop a formula preprocessing technique tuned specifically for Mona (although the approach is, in principle, relevant to all automata-based WSkS solvers).

Antiprenexing is advantageous for the satisfiability test of Mona for the following reason. Mona builds an automaton representing all models of the formula and then tests emptiness of its language. An automaton for a formula is built inductively, starting from predefined atomic automata for atomic formulae and using automata operations that model logical connectives to combine automata for sub-formulae to automata for larger formulae. The bottleneck is the size of the automata built during the process, which may grow with every automata operation, leading, in the worst case, to a tower of exponentials. For Mona, the logical connective with the most expensive automata counterpart is quantification, which involves determinization and is, therefore, exponential in the worst case[1]. Antiprenexing pushes quantifiers deeper in the formula, which causes that the costly quantification is applied on formulae with (hopefully) smaller automata that appear closer to atoms. Morever, since the non-elementary worst-case complexity comes from the number of quantifier alternations, pushing quantifiers to the atoms can decrease this number and, therefore, get a formula with a simpler quantifier structure that can be decided easier by Mona.

Our formula preprocessing is implemented as a set of *syntactic rewriting rules*, most of which are well-known rules (or variants of rules) from transformations to the negation normal form, prenex normal form, or disjunctive normal form. The rules may, however, be applied in different ways and extent, and, if used in an unsuitable way, they may cause an explosion in the size of the formula and the automata built while deciding it. This can happen, e.g., due to unrestricted distribution of disjunctions over conjunctions, which may lead to an exponential growth of the formula, which would outweigh all potential benefits. To resolve the issue, we use *informed rules* that allow us to control the transformations based on how they change the *cost* of deciding the formula, which is given by the size of all automata to be constructed during the decision procedure. Since we, of course, cannot construct the automata beforehand to get their precise size, we *estimate* their sizes using a *linear model* trained from runs of the decision procedure on various formulae using *linear regression*.

We have identified parameters of our preprocessing technique that control the balance of certain trade-offs. Although we have identified several settings of these parameters that appear to be generally advantageous, they are by no means optimal in all cases. Different classes of formulae tend to have different optimal settings. Searching through the space of parameter settings thus gives a good opportunity to solve otherwise unsolvable formulae, or to increase the efficiency of Mona for specific classes of similar formulae.

We demonstrate on a benchmark set (including all WSkS formulae we could gather) that our formula preprocessing significantly improves the overall efficiency of Mona (in some cases, the improvement was in the order of two to three orders of magnitude). Indeed, it allows Mona to solve several formulae of practical interest that were beyond capabilities of any WSkS solver (including Mona).

[1] Mona only works with complete deterministic automata, therefore, complementation, which is usually considered an expensive automata operation, takes only constant time.

2 Preliminaries

In this section, we introduce basic notation and essential preliminaries on trees, tree automata, WSkS, and its decision procedure as implemented in MONA.

2.1 Basics, Trees, and 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 $\mathrm{dom}(f) = \{x \in X \mid \exists y : x \mapsto y \in f\}$, and its *restriction* to a set Z is the function $f_{|Z} = f \cap (Z \times Y)$. We use $f \triangleleft \{x \mapsto y\}$ where $x \in X$ and $y \in Y$ to denote the mapping $(f \setminus (\{x\} \times Y)) \cup \{x \mapsto y\}$.

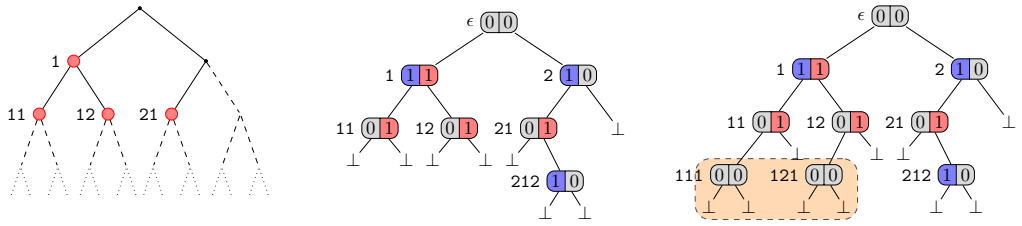
Trees. We will consider ordered k-ary trees. We call a word $p \in \{\mathtt{1}, \ldots, \mathtt{k}\}^*$ a tree *position*, and, for each $\mathtt{i} \in \{\mathtt{1}, \ldots, \mathtt{k}\}$, we call $p.\mathtt{i}$ its i-th *child*. Given an alphabet Σ s.t. $\bot \notin \Sigma$, a *tree* over Σ is a finite partial function $\tau : \{\mathtt{1}, \ldots, \mathtt{k}\}^* \to (\Sigma \cup \{\bot\})$ such that (i) $\mathrm{dom}(\tau)$ is non-empty and prefix-closed (i.e., for $w \in \Sigma^*$ and $a \in \Sigma$, if $wa \in \mathrm{dom}(\tau)$, then also $w \in \mathrm{dom}(\tau)$), and (ii) for all positions $p \in \mathrm{dom}(\tau)$, either $\tau(p) \in \Sigma$ and p has all k children, or $\tau(p) = \bot$ and p has no children, in which case it is called a *leaf*. The position ϵ is called the *root*. We write $\Sigma^{\text{木}}$ to denote the set of all trees over Σ and use $a^{\text{木}}$ to denote $\{a\}^{\text{木}}$ for $a \in \Sigma$.[2]

Tree Automata. A k-ary *tree automaton* (TA) over an alphabet Σ is a quadruple $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ where Q is a finite set of *states*, $\delta : Q^k \times \Sigma \to 2^Q$ is a *transition function*, $I \subseteq Q$ is a set of *leaf states*, and $R \subseteq Q$ is a set of *root* states. By $|\mathcal{A}|$, we denote the number of states of $\mathcal{A}$. We use $(q_1, \ldots, q_k) \text{-}\{a\}\!\mapsto s$ to denote that $s \in \delta((q_1, \ldots, q_k), a)$. 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 $(\rho(p.\mathtt{1}), \ldots, \rho(p.\mathtt{k})) \text{-}\{a\}\!\mapsto \rho(p)$ 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 a *(bottom-up) deterministic* TA (DTA) if $|I| = 1$ and $\forall q_1, \ldots, q_k \in Q, a \in \Sigma : |\delta((q_1, \ldots, q_k), a)| \leq 1$, and *complete* if $I \geq 1$ and $\forall q_1, \ldots, q_k \in Q, a \in \Sigma : |\delta((q_1, \ldots, q_k), a)| \geq 1$. A DTA is *minimal* if it is complete and there is no complete DTA with strictly less states that accepts the same language. Last, for $a \in \Sigma$, we shorten $\delta((q_1, \ldots, q_k), a)$ as $\delta_a(q_1, \ldots, q_k)$, and we use $\delta_\Gamma(q_1, \ldots, q_k)$ to denote $\bigcup\{\delta_a(q_1 \ldots, q_k) \mid a \in \Gamma\}$ for a set $\Gamma \subseteq \Sigma$.

2.2 WSkS

Syntax and Semantics of WSkS. WSkS 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 $\{\mathtt{1}, \ldots, \mathtt{k}\}^*$ (the finiteness of variable assignments is reflected in the name *weak* and k denotes a number of successors). Atomic formulae (atoms) of WSkS are of the form (i) $X \subseteq Y$ and (ii) $X = S_\mathtt{i}(Y)$ for $\mathtt{i} \in \{\mathtt{1}, \ldots, \mathtt{k}\}$. Intuitively, the $S_\mathtt{i}(Y)$ function returns all positions from Y shifted to their i-th child. Formulae are constructed from atoms using the logical connectives $\wedge, \neg$, and the quantifier $\exists \mathcal{X}$ where $\mathcal{X}$ is a finite set of variables (we write $\exists X$ when $\mathcal{X}$ is the singleton set $\{X\}$). Other connectives (such as $\vee$ or $\forall$) and predicates can be

[2] 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).



(a) Positions assigned to Y (b) The minimal encoding of κ (c) A non-minimal encoding of κ

Figure 1: Consider a set of variables $\mathcal{X} = \{X, Y\}$ and an assignment $\kappa = \{X \mapsto \{\mathtt{1}, \mathtt{2}, \mathtt{212}\}, Y \mapsto \{\mathtt{1}, \mathtt{11}, \mathtt{12}, \mathtt{21}\}\}$ in WS2S. In (a), we show positions assigned to variable Y. The minimal encoding of κ into a binary tree is shown in (b), and an encoding that is not minimal is shown in (c).

obtained as syntactic sugar. Despite it not being a basic connective, we mention the disjunction below too since we use specific optimisations to deal with it in the following sections.

A *model* of a WSkS formula $\varphi(\mathcal{X})$ with the set of free variables $\mathcal{X}$ is an assignment $\nu : \mathcal{X} \to 2^{\{\mathtt{1},\ldots,\mathtt{k}\}^*}$ of the free variables of φ to finite subsets of $\{\mathtt{1}, \ldots, \mathtt{k}\}^*$ for which the formula is *satisfied*, written $\nu \models \varphi$. Satisfaction of WSkS formulae is defined as follows:

(i) $\nu \models X \subseteq Y$ iff $\nu(X) \subseteq \nu(Y)$,

(ii) $\nu \models X = S_{\mathtt{i}}(Y)$ iff $\nu(X) = \{p.\mathtt{i} \mid p \in \nu(Y)\}$ for $\mathtt{i} \in \{\mathtt{1}, \ldots, \mathtt{k}\}$,

(iii) $\nu \models \varphi_1 \wedge \varphi_2$ iff $\nu \models \varphi_1$ and $\nu \models \varphi_2$,

(iv) $\nu \models \varphi_1 \vee \varphi_2$ iff $\nu \models \varphi_1$ or $\nu \models \varphi_2$,

(v) $\nu \models \neg\varphi$ iff not $\nu \models \varphi$, and

(vi) $\nu \models \exists X.\ \varphi$ iff there is a finite $S \subseteq \{\mathtt{1}, \ldots, \mathtt{k}\}^*$ s.t. $\nu \triangleleft \{X \mapsto S\} \models \varphi$.

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. Further, we denote the set of free variables of φ by $\mathsf{fv}(\varphi)$ and the set of all sub-formulae of φ (including φ) by $\mathsf{sf}(\varphi)$.

Representing Models as Trees. Let $\mathcal{X}$ be a finite set of variables. A *symbol* ξ over $\mathcal{X}$ is a (total) function $\xi : \mathcal{X} \to \{0, 1\}$; e.g., $\xi = \{X \mapsto 0, Y \mapsto 1\}$ is a symbol over $\mathcal{X} = \{X, Y\}$. We use $\Sigma_{\mathcal{X}}$ to denote the set of all symbols over $\mathcal{X}$ and $\vec{0}_{\mathcal{X}}$ to denote the symbol mapping all variables in $\mathcal{X}$ to 0, i.e., $\vec{0}_{\mathcal{X}} = \{X \mapsto 0 \mid X \in \mathcal{X}\}$. When $\mathcal{X}$ is clear from the context, we write $\vec{0}$.

A finite assignment $\nu : \mathcal{X} \to 2^{\{\mathtt{1},\ldots,\mathtt{k}\}^*}$ of the free variables of a formula φ can be encoded as a finite tree τ_ν of symbols over $\mathcal{X}$ where every position $p \in \{\mathtt{1}, \ldots, \mathtt{k}\}^*$ 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 $\tau_\nu(p') = \bot$ for some prefix p' of p (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 minimal encoding τ_ν^{min} and (infinitely many) extensions of τ_ν^{min} with $\vec{0}_{\mathcal{X}}$-only trees. See Figure 1 for an example of an assignment and its encodings. The *language* of φ is defined as the set of all encodings of its models $\mathcal{L}(\varphi) = \{\tau_\nu \in \Sigma_{\mathcal{X}}^{\bar{\star}} \mid \nu \models \varphi \text{ and } \tau_\nu \text{ is an encoding of } \nu\}$.

Let ξ be a symbol over $\mathcal{X}$. For a set of variables $\mathcal{Y} \subseteq \mathcal{X}$, we define the *projection* of ξ wrt $\mathcal{Y}$ as $\pi_{\mathcal{Y}}(\xi) = \xi_{|\mathcal{X}\setminus\mathcal{Y}}$. Intuitively, the projection removes the original assignments of variables in $\mathcal{Y}$, i.e., $\mathcal{Y}$'s variables can have any value after the projection. We also define the inverse projection as $\pi_{\mathcal{Y}}^{-1}(\xi) = \{\xi' \mid \xi = \pi_{\mathcal{Y}}(\xi')\}$. We define $\pi_{\mathcal{Y}}(\bot) = \bot$, and write π_Y if $\mathcal{Y}$ is the singleton set $\{Y\}$. As an example, for $\mathcal{X} = \{X, Y\}$, the projection of $\vec{0}_{\mathcal{X}}$ wrt $\{X\}$ is given as $\pi_X(\vec{0}_{\mathcal{X}}) = \{Y \mapsto 0\}$. The definition of projection can be extended to a tree τ over $\Sigma_{\mathcal{X}}$ so that $\pi_{\mathcal{Y}}(\tau)$ is given as $\pi_{\mathcal{Y}}(\tau) = \pi_{\mathcal{Y}} \circ \tau$ and, subsequently, to a language L so that $\pi_{\mathcal{Y}}(L) = \{\pi_{\mathcal{Y}}(\tau) \mid \tau \in L\}$.

The Decision Procedure for WSkS in Mona. We now recall the variant of the classical decision procedure for WSkS implemented in Mona. Compared to the textbook version of the decision procedure (cf. [13]), Mona is specific mainly in that it works with complete deterministic automata only and uses DTA minimization extensively. We note that Mona uses a number of other crucial optimizations, such as a symbolic, BDD-based representation of the transition relation or so-called three-valued automata with "don't care" states [29], but these are not directly relevant to our contribution, and so we will not discuss them in this text.

When, e.g., testing satisfiability of a formula φ, Mona constructs the minimal DTA $\mathcal{A}_\varphi$ over the alphabet $\Sigma_{\mathsf{fv}(\varphi)}$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and then tests whether $\mathcal{L}(\mathcal{A}_\varphi) = \emptyset$. The emptiness test is a standard least fixpoint computation of the set of states with non-empty languages. It starts with the set of leaf states and keeps adding states that can be reached from the so-far encountered states by an application of a transition rule (in a bottom-up direction) until a fixpoint is reached. The language is non-empty if the fixpoint contains a root state.

The automaton $\mathcal{A}_\varphi$ is constructed by induction on the structure of φ. Namely, if φ is an atomic formula with free variables $\mathcal{X}$, then $\mathcal{A}_\varphi$ is a pre-defined *base* minimal complete DTA over $\Sigma_{\mathcal{X}}$. The particular base automata for the atomic predicates can be found, e.g., in [13]. Otherwise, if φ is not atomic, then $\mathcal{A}_\varphi = \mathsf{min}(\mathcal{A}^\partial_\varphi)$, i.e., it is obtained by minimizing the DTA $\mathcal{A}^\partial_\varphi$, which is created from the automata for φ's sub-formulae by the automata operation corresponding to the top-level logical operator of φ in the following way (the automata operations preserve determinism and completeness):

- If $\varphi = \psi \star \psi'$ with $\star \in \{\wedge, \vee\}$, then, for $\mathcal{A}_\psi = (Q, \Sigma_{\mathcal{X}}, \delta, I, R)$ and $\mathcal{A}_{\psi'} = (Q', \Sigma_{\mathcal{X}'}, \delta', I', R')$, the DTA $\mathcal{A}^\partial_\varphi$ is obtained by the so-called *cylindrification* and subsequent *product construction*. Namely, cylindrification turns the original two transition relations into a new pair $\hat{\delta}$ and $\hat{\delta}'$ with compatible alphabets, i.e., over variables $\mathcal{X} \cup \mathcal{X}'$. Namely, $\hat{\delta}$ consists of all variants of transitions in δ where the original symbol is extended with any possible assignment to variables in $\mathcal{X}' \setminus \mathcal{X}$, and $\hat{\delta}'$ is obtained symmetrically from δ'. The product construction then produces the DTA $\mathcal{A}^\partial_\varphi = (Q \times Q', \Sigma_{\mathcal{X}\cup\mathcal{X}'}, \delta^\times, I \times I', R^\star)$ with $\delta^\times_a((q_1, q'_1), \ldots, (q_k, q'_k)) = \hat{\delta}_a(q_1, \ldots, q_k) \times \hat{\delta}'_a(q'_1, \ldots, q'_k)$ and $R^\star = \{(q, q') \in Q \times Q' \mid q \in R \star q' \in R'\}$.
- If $\varphi = \neg\psi$, then $\mathcal{A}^\partial_\varphi$ is obtained from $\mathcal{A}_\psi$ by complementing its set of root states. The operation preserves determinism, minimality, and completeness of the transition relation, hence $\mathcal{A}_\varphi$ is taken directly as $\mathcal{A}^\partial_\varphi$, without calling the minimization.
- If $\varphi = \exists X.\ \psi$, then the automaton $\mathcal{A}^\partial_\varphi$ is constructed as $\mathsf{det}(\pi_X(\mathcal{A}_\psi) - \vec{0}^{\star})$ where π_X projects X from $\mathcal{A}_\psi$'s alphabet, the operation $-\vec{0}^{\star}$ saturates the set of leaf states of the intermediate (in general nondeterministic) TA such that all $\vec{0}$-labelled sub-trees are accepted[3], and det determinizes the result. In more detail, given a TA $\mathcal{A} = (Q, \Sigma_{\mathcal{X}}, \delta, I, R)$,

[3]Intuitively, saturating the set of leaf states is needed to ensure that *every* encoding of every model is accepted. Indeed, if some were not accepted, the inductive construction could produce a wrong re-

the three operations proceed as follows: (1) The projection of a variable X produces the automaton $\pi_X(\mathcal{A}) = (Q, \Sigma_{\mathcal{X}\setminus\{X\}}, \delta^{\pi_X}, I, R)$ with $\delta_a^{\pi_X}(q_1, \ldots, q_k) = \delta_{\pi_X^{-1}(a)}(q_1, \ldots, q_k)$. (2) The saturation of the set of leaf states uses a least fixpoint computation similar to the computation of the set of states with non-empty languages, but it takes into account transitions labeled by $\vec{0}$ only. (3) The determinization uses the subset construction to produce the automaton $\mathsf{det}(\mathcal{A}) = (2^Q, \Sigma_{\mathcal{X}}, \mathsf{det}(\delta), \{I\}, \mathsf{det}(R))$ where $\mathsf{det}(\delta_a)(S_1, \ldots, S_k) = \bigcup_{q_1 \in S_1, \ldots, q_k \in S_k} \delta_a(q_1, \ldots, q_k)$ and $\mathsf{det}(R) = \{S \subseteq Q \mid S \cap R \neq \emptyset\}$.

3 Formula Transformations

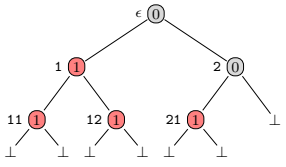
In this section, we describe our formula transformation algorithm. It is based on well-known rules for transformation of formulae to the *negation normal form* (NNF) and the *antiprenex form* (APF) [16], together with distributive laws. Recall that, during transformation into NNF, negations are pushed deeper into the formula so that they occur in front of atoms only, and during transformation into APF, quantifiers are pushed deeper into the formula in order to minimize their scopes (APF can be viewed as the opposite of the prenex normal form). In the following, we assume that the processed formula contains only existential quantifiers $\exists$ and Boolean connectives $\wedge, \vee$, and $\neg$. (Note that, strictly speaking, we do not transform the formulae into the NNF; instead, we transform them into a form where negations are in front of atoms or $\exists$ quantifiers.)

We will discuss heuristics for choosing which formula transformation rules to apply and when to apply them so that the resulting formula is as easy as possible for the automata solver. We fine-tune the heuristics particularly for the algorithm of MONA and with respect to the specific way it processes individual logical connectives (cf. Section 2.2). Namely, MONA always works with complete DTAs (its data structures cannot even directly represent nondeterminism). Negation is implemented as complementation of such DTAs, which is cheap (it is sufficient to just invert the acceptance condition). The $\wedge$ and $\vee$ connectives are implemented through an automata product construction, which is quadratic (the two constructions differ only in their treatment of the acceptance condition). Although the potential quadratic blow-up is not the source of the worst-case non-elementary complexity of WSkS, a sequence of product constructions is still exponential and in practice is often the main cause of a state explosion. Projection performed while processing the $\exists$ connective is the only operation that introduces non-determinism, and is therefore done together with determinization, which is, in the worst case, exponential. The non-elementary worst-case complexity of the procedure stems from here.

An important factor in MONA's performance is that it minimizes automata after every operation. Without minimization, the results of operations would often be many times larger than the operands, and the construction would quickly explode in size. Minimization is usually able to keep automata sizes at bay. Its effect is particularly well visible after existential quantification (which includes determinization) after which the size of the result might explode significantly, but the minimized automaton is in many cases smaller than the original.

sult. For instance, language complementation would not complement the set of encoded models since some model could have encodings in both the original language and its complement.

Consider, for example, a formula ψ having the language $\mathcal{L}(\psi)$ given by the tree τ_ν in Figure 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 minimal encoding shown in the figure on the right, only its extensions (the minimal one would hence be present in the complement of the language). The saturation of the set of leaf states includes the minimal encoding into the language. See [13] for more details.



3.1 Cost of Deciding a Formula

Some of the rewriting rules that will be discussed below are driven by heuristic estimates of the cost of building the DTA representing the transformed formula. The cost of deciding a formula in automata-based decision procedures is essentially proportional to the sum of the sizes of all automata that are built while the formula is being decided. Recall that MONA builds two automata for every non-atomic sub-formula φ: the automaton $\mathcal{A}^{\partial}_{\varphi}$, resulting directly from applying the root operator of the sub-formula, and its minimized version $\mathcal{A}_{\varphi} = \mathsf{min}(\mathcal{A}^{\partial}_{\varphi})$. On the other hand, the base automata $\mathcal{A}_{\varphi}$ for atomic formulae are directly generated minimal, without an intermediate $\mathcal{A}^{\partial}_{\varphi}$. The cost of deciding the formula is hence proportional to the sum

$$\|\varphi\| = \sum_{\psi \in \mathsf{sf}(\varphi)} |\mathcal{A}_{\psi}| + |\mathcal{A}^{\partial}_{\psi}| \tag{1}$$

where $\mathsf{sf}(\varphi)$ is the set of all sub-formulae of φ and $|\mathcal{A}^{\partial}_{\psi}| = 0$ if ψ is an atomic formula.

Computing the cost of a formula $\|\varphi\|$ precisely would require one to actually run through the entire decision procedure for φ and build all the TAs, which is clearly impractical as a means of optimizing the very same computation. We therefore use a cheap estimate $\|\varphi\|^{\sim}$. The means of obtaining the estimate, by linear regression from a sample set of formulae, are discussed in the next section. In this section, we focus on how the estimates are used to drive the rewriting.

3.2 Quantifier Distribution and Scope Narrowing

The core of our formula rewriting are rules for narrowing the scope of quantifiers by moving them towards literals. The most important rule is *quantifier distribution* over disjunction:

$$\exists \mathcal{X}.\ \varphi \vee \psi \quad \rightsquigarrow \quad (\exists \mathcal{X}.\ \varphi) \vee (\exists \mathcal{X}.\ \psi) \tag{QuantDistr}$$

Using this rule is generally beneficial for the following reasons[4]. Disjunction is expensive, often quadratic, and the result is often significantly more complex than the arguments, even after the result's minimization. The arguments of the quantifications on the right-hand side of the rule, $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\psi}$, are hence likely to be substantially smaller than the argument of quantification on the left-hand side of the rule, $\mathcal{A}_{\varphi \vee \psi}$. This is desirable since quantification is often the most expensive operation, exponential in the worst case. Moreover, minimization often reduces the size of the result of quantification to even smaller than the size of the automaton before quantification, in which case the product construction is applied on smaller arguments after the transformation than before it. We therefore use quantifier distribution whenever applicable.

Quantifier *scope narrowing* is another way of moving a quantifier towards literals, this time through a conjunction:

$$\exists \mathcal{X}.\ \varphi \wedge \psi \quad \rightsquigarrow \quad \varphi \wedge (\exists \mathcal{X}.\ \psi) \ \text{ provided } \mathcal{X} \text{ are not free in } \varphi. \tag{ScopeNarrow}$$

The rule is justified in a similar way as (QuantDistr): $\mathcal{A}_{\psi}$ is probably smaller than $\mathcal{A}_{\varphi \wedge \psi}$ and applying quantification on a smaller operand is preferable. Secondly, the automaton $\mathcal{A}_{\exists \mathcal{X}.\psi}$ may be smaller than $\mathcal{A}_{\psi}$, making the product construction cheaper after the transformation too.

[4] This may be contrary to first-order theorem proving, where the benefit of performing quantifier distribution comes at the price of an increased number of function symbols when the formula is Skolemized afterwards.

3.3 Supporting Rules

Further rewriting rules we use push negation deeper into the formula, distribute $\wedge$ over $\vee$, or restructure $\wedge$. These rules only have a supporting role; their purpose is to enable (QuantDistr) and (ScopeNarrow).

Pushing Negation. First, the following rules, standard in the transformation into NNF, are used essentially whenever applicable for *pushing negation* inwards by De Morgan's laws and for eliminating double negation:

$$\begin{aligned} \neg(\varphi \wedge \psi) &\rightsquigarrow \neg\varphi \vee \neg\psi \\ \neg(\varphi \vee \psi) &\rightsquigarrow \neg\varphi \wedge \neg\psi \\ \neg\neg\varphi &\rightsquigarrow \varphi \end{aligned} \tag{PushNeg}$$

Negation has a negligible cost with DTAs, hence an application of these rules alone does not normally change the running time of Mona much. Their purpose is to enable applicability of all other rules. The rules in (PushNeg) ultimately push negations to atoms or to quantifiers. Negations at atoms can be completely eliminated by DTA complementation. Negations in front of the $\exists$ quantifier cannot be pushed inside unless the quantifier itself is first pushed inwards by (QuantDistr) or (ScopeNarrow) after which the negation can also follow.

Distribution of Conjunction. Second, we use *distribution of conjunction* (over disjunction under quantification):

$$\exists \mathcal{X}.\ \varphi \wedge (\psi \vee \omega) \quad \rightsquigarrow \quad \exists \mathcal{X}.\ (\varphi \wedge \psi) \vee (\varphi \wedge \omega) \tag{2}$$

Applying the rule enables quantifier distribution (rule (QuantDistr)). Its application may, however, result in a formula that is more difficult to decide due to the following reasons:

(i) The threefold product on the right of the rule might be larger and more expensive than the twofold product on the left, even after quantifier distribution, especially if $\mathcal{A}_\varphi$ is large.

(ii) Even though Mona represents a formula as a DAG of its sub-formulae in which all occurrences of the same sub-formula φ correspond to a single node, an iterated application of distribution that duplicates larger and larger φ, might ultimately lead to an exponential explosion in the size of the formula and its DAG (the formula might be ultimately turned to the exponentially larger disjunctive normal form).

Therefore, we make the question of whether to apply the distribution of conjunction subject to a heuristic decision based on the estimated cost of φ. Namely, the rule is used in the form

$$\exists \mathcal{X}.\ \varphi \wedge (\psi \vee \omega) \quad \rightsquigarrow \quad \exists \mathcal{X}.\ (\varphi \wedge \psi) \vee (\varphi \wedge \omega) \quad \text{if } \|\varphi\|^{\sim} \leq \textsc{DistrThres} \tag{$\wedge$-Distr}$$

where DistrThres is a parameter specifying the maximum estimated cost of the formula for which we allow application of the rule.

Restructuring Conjunctions. Our last transformation rule is used to facilitate quantifier scope narrowing (rule (ScopeNarrow)). It is the rule of *restructuring of conjunctions*, denoted as (Restr&Narrow). Consider a formula $\exists X_1 \ldots \exists X_m.\ \varphi$ where φ is a (possibly large and nested) conjunction. The rule can be seen as performing the following three actions: (i) reordering the

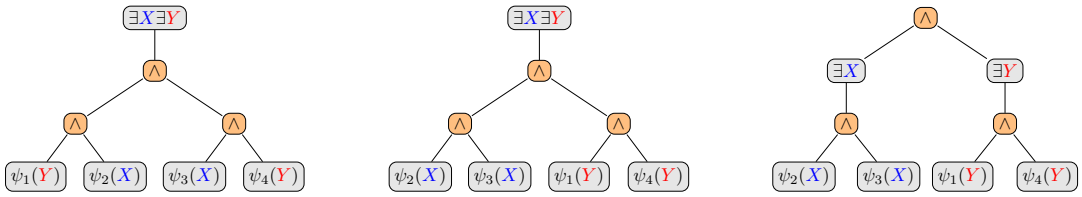


Figure 2: An example of how conjunction restructuring can help antiprenexing. The tree in (a) represents the formula $\omega_1 : \exists X \exists Y.\ (\psi_1 \wedge \psi_2) \wedge (\psi_3 \wedge \psi_4)$ where Y is the only free variable of ψ_1 and ψ_4 and X is the only free variable of ψ_2 and ψ_3. Notice that none of the quantifiers can be pushed inside. The tree in (b) represents a formula obtained from ω_1 by applying the associative and commutative laws to gather sub-formulae with the same free variables together, enabling the use of the (ScopeNarrow) rule. The tree in (c) is obtained from (b) by applying the rule twice.

sequence of quantifiers $\exists X_1 \dots \exists X_m$ into the form most suitable for the next step, (ii) using the laws of associativity and commutativity for $\wedge$ to restructure the top-most conjunctions of φ so that the scope narrowing (wrt the order induced in the previous step) can have the greatest possible effect, and (iii) performing (ScopeNarrow) to push quantifiers as deep as possible.

We will start by describing how the restructuring itself is performed. Let p be a permutation of the set $\{1, \dots, m\}$, which induces the following reordering of the quantifiers in the transformed formula: $\exists X_{p(1)} \dots \exists X_{p(m)}$ (we will describe how we obtain p later). Consider a sequence of quantifiers $\rho = \exists X_{p(1)} \dots \exists X_{p(m)}$ and a formula ω. Further, let ω' be a formula obtained from $\rho.\ \omega$ by applying (ScopeNarrow) on the top-most conjunctions as long as possible. We then say that ω is *optimal for narrowing wrt* ρ if no sequence of applications of the commutativity and associativity laws on the top-most conjunctions of ω' enables any more application of (ScopeNarrow). We use φ_p to denote a formula obtained by restructuring φ's top-most conjunctions using associativity and commutativity laws that is optimal for narrowing wrt $\exists X_{p(1)} \dots \exists X_{p(m)}$ (there might be more such formulae; picking any of them works for us). See Figure 2 for an example of how $\wedge$-restructuring enables using the (ScopeNarrow) rule.

Constructing φ_p for a given permutation p is implemented as a call to the function `Restr&Narrow(`p`)`, given in the right. The function not only creates φ_p, but also performs, on the fly, quantifier scope narrowing (so that it is not necessary to apply (ScopeNarrow) on the result), producing a formula denoted as φ_p^{sn}. Assume that φ can be written modulo commutativity and associativity of conjunction as $\bigwedge_{1 \le i \le n} \psi_i$ where no ψ_i is itself a conjunction. The function maintains the set Ψ of the leaves of the current conjunction, initialised as the set $\{\psi_i\}_{i=1}^n$. It then iterates through numbers j from m to 1, and, in each iteration, collects into Φ_j all formulae in Ψ that contain $X_{p(j)}$ as a free variable, and replaces them in Ψ by the formula $\exists X_{p(j)}.\ \bigwedge \Phi_j$. After the m-th iteration, Ψ contains the formula $\exists X_{p(1)}.\ \Phi_1$, and also the original formulae ψ_i that contain no variable from $X_1, \dots, X_m$. `Restr&Narrow(`p`)` then returns the conjunction of all those formulae. (We note that the function works with the generalized n-ary conjunction; when implemented, the returned formula uses only binary conjunctions.)

Function `Restr&Narrow(`p`)`**:**
  $\Psi := \{\psi_i\}_{i=1}^n$;
  for $j := m$ **downto** 1 **do**
    $\Phi_j := \{\psi_i \in \Psi \mid X_{p(j)} \in \mathsf{fv}(\psi_i)\}$;
    $\Psi := (\Psi \setminus \Phi_j) \cup \{\exists X_{p(j)}.\ \bigwedge \Phi_j\}$;
  return $(\bigwedge \Psi)$;

Note that, in the previous paragraphs, we were using a permutation p of the quantifiers as a parameter of the restructuring. The way how quantifiers are ordered is important because it determines how well the restructuring can be done (see Figure 3 for an example).

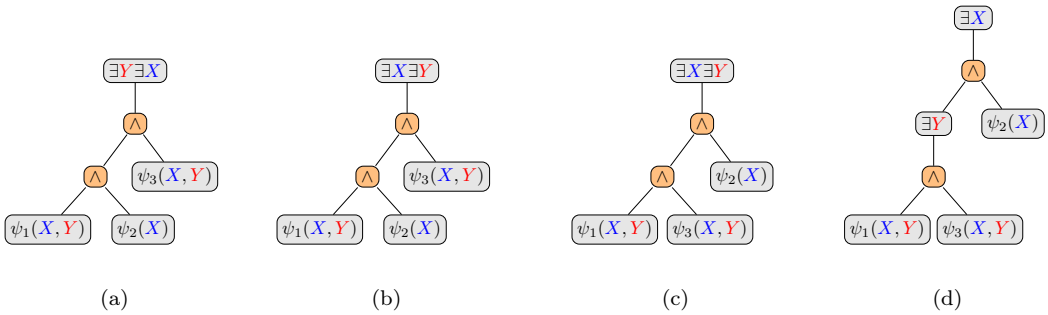


Figure 3: An example of how reordering quantifiers can help with quantifier scope narrowing. The tree in (a) represents the formula $\omega_2 : \exists Y \exists X.\ (\psi_1(X,Y) \wedge \psi_2(X)) \wedge \psi_3(X,Y)$. Notice that for the order of quantifiers $\exists Y \exists X$, the $\wedge$-tree is optimal for narrowing. In (b), we changed the order of quantifiers to $\exists X \exists Y$. This change enables restructuring the $\wedge$-tree into a more suitable form (the tree in (c)), which, in turn, allows quantifier scope narrowing (the tree in (d)).

(Restr&Narrow) then works as follows: it searches through all permutations p of the set $\{1,\ldots,m\}$. For each p, it constructs the formula φ_p^{sn} using `Restr&Narrow(p)` and computes its estimated cost $\|\varphi_p^{sn}\|^{\sim}$. Finally, the formula φ_p^{sn} with the smallest estimated cost is returned.

The basic version of (Restr&Narrow) described above enumerates all permutations p of the set $\{1,\ldots,m\}$ and for each constructs the formula φ_p^{sn} and computes an estimate of its cost. Performing this computation for a larger number of quantifiers is obviously infeasible (there are $m!$ permutations over them). We therefore propose the following three heuristics: First, we do not distinguish permutations that induce formulae whose cost is obviously the same. In particular, we group together variables that (i) always or (ii) never occur free together in all formulae ψ_i from $\{\psi_i\}_{i=1}^n$. We then treat each such group as a single variable when generating the permutations (when later generating the final formula, we fix an arbitrary permutation of the variables within each group). For example, in the formula $\exists X \exists Y \exists Z.\ \psi_1(X,Z) \wedge \psi_2(Y,Z)$, the variables X and Y never appear together, so we consider only the following two orderings of quantifiers: (i) $\exists\{X,Y\}\exists Z.\ \psi_1(X,Z) \wedge \psi_2(Y,Z)$ and (ii) $\exists Z \exists\{X,Y\}.\ \psi_1(X,Z) \wedge \psi_2(Y,Z)$.

The second heuristic works as follows. If the number of possible orderings is still too high, we split the sequence of quantifiers $\exists X_1 \ldots \exists X_m$ into subsequences of the length h (except the last one which can be shorter), i.e., $\exists X_1 \ldots \exists X_h; \exists X_{h+1} \ldots \exists X_{2h}; \ldots; \exists X_{jh} \ldots \exists X_m$, for some j, and try to find the best ordering for every such subsequence independently. The constant h is controlled by the parameter R&NSeqMax. The third heuristic addresses the situation when, despite the optimizations above, the application of (Restr&Narrow) may still be too costly. We therefore use the parameter R&NThres to bound the maximum size of the conjunction $\{\psi_i\}_{i=1}^n$ for which (Restr&Narrow) can be applied.

3.4 Top-level Algorithm

The top-level algorithm executing formula transformations works as follows. It runs in *iterations*, the number of which is controlled by the parameter Iters. In each iteration, the rules are applied in one of the following two sequences Full and Simple:

$$\textsc{Full} = (\text{PushNeg})^{\downarrow}; \big((\text{Restr\&Narrow}) + (\text{QuantDistr})\big)^{\downarrow}; (\wedge\text{-Distr})^{\uparrow} \quad \text{and}$$
$$\textsc{Simple} = (\text{PushNeg})^{\downarrow}; \big((\text{ScopeNarrow}) + (\text{QuantDistr})\big)^{\downarrow}.$$

where ";" denotes sequential composition of operations and "+" denotes interleaved application of operations; "$^{\downarrow}$" denotes that the rewriting rule is applied using a pre-order traversal of the syntax tree of a formula (i.e., top-down), while "$^{\uparrow}$" applies the rules in a post-order traversal (i.e., bottom-up). The majority of the rules are applied top-down; this corresponds with the fact that the rules are pushing quantifiers *inside* the formula. The only rule applied bottom-up is ($\wedge$-Distr); the reason for this is that if we applied it top-down, the distribution would be done on larger formulae, while when applied bottom-up, the formulae it is applied on are smaller (since this rule does not push quantifiers inside, but only *enables* the pushing, we perform an additional (QuantDistr)$^{\downarrow}$ after the last iteration).

In both sequences, each iteration is started by pushing negations deeper into a formula using (PushNeg). Then, in FULL, the rule (Restr&Narrow) is interleaved with (QuantDistr) to push quantifiers into conjunctions and distribute $\exists$ over $\vee$. Finally, ($\wedge$-Distr) is used to distribute $\wedge$ over $\vee$. On the other hand, in SIMPLE, (PushNeg) is followed just by interleaving quantifier scope narrowing with distributing $\exists$ over $\vee$. (Note that it may seem that (ScopeNarrow) is only used by SIMPLE and not by FULL; in fact, the rule is used in FULL internally within (Restr&Narrow).) The particular sequence of operations (FULL or SIMPLE) to be used is determined by the size of the input formula φ. In particular, if $|\mathsf{sf}(\varphi)| \leq$ SIMPLETHRES, we pick the more expensive FULL, otherwise we pick the cheaper SIMPLE, where SIMPLETHRES is a parameter whose value specifies the threshold.

Predicate Inlining. The last preprocessing step we use is inlining of *user-defined predicates*, a specific syntactic feature of MONA. User-defined predicates are named formulae with free variables that can be used (non-recursively) in other formulae. Their use improves the readability of formulae in MONA, but, on the other hand, restricts applications of our transformation rules (e.g., we cannot push quantifiers beyond a predicate boundary). We therefore introduce a Boolean parameter INLINE that, when set to *true*, enables inlining all user-defined predicates.

4 Automata Size Estimation

We will now discuss how to cheaply compute the estimate $\|\varphi\|^{\sim}$ of the formula cost $\|\varphi\|$, which is a parameter of the rules in the previous section (in particular, the rules performing informed distribution and conjunction restructuring). Computing the precise number would be as difficult as deciding the formula itself, hence we seek an inexpensive, yet good approximation. The approximation we use is based on the estimates $|\mathcal{A}_\psi|^{\sim}$ and $|\mathcal{A}^{\partial}_\psi|^{\sim}$ of the sizes of the DTAs $\mathcal{A}_\psi$ and $\mathcal{A}^{\partial}_\psi$, respectively, for each sub-formula ψ of φ. Namely, we compute $\|\varphi\|^{\sim}$ in the form

$$\|\varphi\|^{\sim} = \sum_{\psi \in \mathsf{sf}(\varphi)} |\mathcal{A}_\psi|^{\sim} + |\mathcal{A}^{\partial}_\psi|^{\sim} \,. \tag{3}$$

We propose an approach that learns a function estimating automata sizes based on the following: (i) the estimates of the sizes of automata resulting from the direct sub-formulae of φ, and (ii) the type of the top-level logical connective of φ. Moreover, if φ is a conjunction or disjunction, we include as the third parameter of the estimation function the number of shared variables between the conjuncts/disjuncts. In our experience, this number tends to strongly correlate with the size of the resulting TA. Formally, we learn estimation functions ℓ and ℓ^{∂}, indexed by

the formula top-level operator, which are then used to estimate automata sizes as follows:

$$\begin{array}{lll}
\varphi = \psi \wedge \psi' : & |\mathcal{A}_\varphi|^\sim = \ell_\wedge(|\mathcal{A}_\psi|^\sim, |\mathcal{A}_{\psi'}|^\sim, n) & |\mathcal{A}^\partial_\varphi|^\sim = \ell^\partial_\wedge(|\mathcal{A}_\psi|^\sim, |\mathcal{A}_{\psi'}|^\sim, n) \\
\varphi = \psi \vee \psi' : & |\mathcal{A}_\varphi|^\sim = \ell_\vee(|\mathcal{A}_\psi|^\sim, |\mathcal{A}_{\psi'}|^\sim, n) & |\mathcal{A}^\partial_\varphi|^\sim = \ell^\partial_\vee(|\mathcal{A}_\psi|^\sim, |\mathcal{A}_{\psi'}|^\sim, n) \\
\varphi = \exists X.\psi : & |\mathcal{A}_\varphi|^\sim = \ell_\exists(|\mathcal{A}_\psi|^\sim) & |\mathcal{A}^\partial_\varphi|^\sim = \ell^\partial_\exists(|\mathcal{A}_\psi|^\sim) \\
\varphi = \neg\psi : & |\mathcal{A}_\varphi|^\sim = |\mathcal{A}_\psi|^\sim & |\mathcal{A}^\partial_\varphi|^\sim = 0 \\
\varphi = \psi_a : & |\mathcal{A}_\varphi|^\sim = |\mathcal{A}_{\psi_a}| & |\mathcal{A}^\partial_\varphi|^\sim = 0
\end{array}$$

Above, $n = |\mathsf{fv}(\psi) \cap \mathsf{fv}(\psi')|$ and ψ_a is an atomic formula. Since MONA uses minimal DTAs, the automaton for $\neg\psi$ is the same as $\mathcal{A}_\psi$ except the set of root states, hence no intermediate DTA is generated. Similarly, base automata are generated directly minimal and deterministic, hence there is no intermediate automaton $\mathcal{A}^\partial_{\psi_a}$.

The first obvious choice for the functions ℓ and ℓ^∂ would be to use the *worst-case* size of the automata, i.e., $\ell^\partial_\star(|\mathcal{A}_\psi|^\sim, |\mathcal{A}_{\psi'}|^\sim, n) = |\mathcal{A}_\psi|^\sim \cdot |\mathcal{A}_{\psi'}|^\sim$ for $\star \in \{\wedge, \vee\}$, $\ell^\partial_\exists(|\mathcal{A}_\psi|^\sim) = 2^{|\mathcal{A}_\psi|^\sim}$, and $\ell_\bullet = \ell^\partial_\bullet$ for $\bullet \in \{\wedge, \vee, \exists\}$ (in the worst case, minimization is performed, but has no effect). When analyzing the sizes of automata produced by MONA, we, however, noticed that the worst case happens only exceptionally, and in reality, the sizes are much smaller. In particular, the size of $\mathcal{A}^\partial_{\exists X.\psi}$ (resp. $\mathcal{A}_{\exists X.\psi}$) is usually linear to the size of $\mathcal{A}_\psi$ rather than exponential (with a few outliers where the explosion happened). Furthermore, we also noticed that there is a linear correlation between the size of $\mathcal{A}^\partial_{\varphi \star \psi}$ (resp. $\mathcal{A}_{\varphi \star \psi}$) and the value $\|\mathcal{A}_\varphi\| \cdot \|\mathcal{A}_\psi\|$.

Therefore, we chose to use linear functions for $\ell_\bullet$ and $\ell^\partial_\bullet$. In particular, the functions $\ell_\star$ and $\ell_\exists$ are represented as the lines (the lines for ℓ^∂ are similar with different parameters)

$$\begin{array}{c}
\ell_\star(|\mathcal{A}_\psi|^\sim, |\mathcal{A}_{\psi'}|^\sim, n) = a^\star_n \cdot (|\mathcal{A}_\psi|^\sim \cdot |\mathcal{A}_{\psi'}|^\sim) + b^\star_n \quad \text{and} \\
\ell_\exists(|\mathcal{A}_\psi|^\sim) = a^\exists \cdot (|\mathcal{A}_\psi|^\sim) + b^\exists
\end{array} \tag{4}$$

(strictly speaking, $\ell_\star$ is a more general curve in a three dimensional space, but we always work with the product $|\mathcal{A}_\psi|^\sim \cdot |\mathcal{A}_{\psi'}|^\sim$ as the input). We obtain the particular parameters $a^\exists, b^\exists$ and $a^\star_n, b^\star_n$ for every n (and their variants for ℓ^∂) by learning from runs of MONA. As the learning algorithm, we used *linear regression* (its particular version is discussed in Section 5), which is an optimization technique based on fitting input data (points in a Euclidean space) with a hyperplane such that the least square error is minimized [36]. We chose this method for its simplicity and well-predictable behaviour.

5 Evaluation

We have implemented the antiprenexing transformations for WSkS formulae introduced in Section 3 as a Haskell/Python prototype tool named ANTIMONA (ANTIprenexing for MONA)[5]. The tool works as a preprocessor for MONA; it reads a file in the MONA format, applies the transformations, and produces a new file in the same format, which can then be passed to MONA. Our goal is to evaluate the impact of our optimization on MONA. Although there have recently appeared new techniques for deciding WSkS, e.g. [20, 19, 24, 14, 44, 21], we do not focus on comparing with them because the alternative tools are far less mature than MONA. Although they can win over MONA on limited classes of formulae, from our experience, MONA performs better overall and, up to our knowledge, can still be considered the only robust and practically usable tool.

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

Table 1: Parameters of the selected settings of AntiMona.

Name	Iters	DistrThres	SimpleThres	Inline	R&NSeqMax	R&NThres
AntiPrxInl	5	5,000	3,000	*true*	5	5
AntiPrxPr$_1$	5	5,000	3,000	*false*	5	5
AntiPrxPr$_2$	3	5,000	2,000	*false*	5	∞

We implemented only a light-weight estimation of the costs of formulae (cf. Section 4). In particular, our implementation does not consider Mona's *DAGification* optimization, which first transforms a formula into a DAG where nodes corresponding to similar sub-formulae[6] are merged into one, and then constructs automata only for the nodes in the DAG (see [29] for more details). Instead, we work with the syntax tree of a formula and therefore return an over-approximation of the formula's cost estimate (some nodes are counted multiple times) .

Experimental Settings. We have evaluated our technique on formulae we were able to find in the literature or obtain by personal communication (in cases where the appropriate research was not published due to problems with the scalability of Mona) and which our tool could parse. Particularly, our benchmark includes formulae from the Strand benchmark [31], formulae from the authors of Mona [29], benchmarks for synthesis of regular specifications [23], families of parametric WS1S formulae [20], LTL formulae from [48] translated to the MSO(Str)[7] logic [18], and an experimental translation of separation logic formulae into MSO(Str) [6]. In total, our benchmark set has 103 formulae (95 WS1S and 8 WS2S), available in the tool's repository.

The experiments were run on a 64-bit Debian GNU/Linux workstation with Intel(R) Xeon(R) E5-2630 v2 CPU running at 2.60 GHz with 32 GiB of RAM, using Mona v1.4-17.

Learning Formula's Cost Estimate. The function performing size estimates of automata constructed from formulae is learnt from runs of Mona on all sub-formulae obtained from a set of selected WS1S formulae (in total, this gave us 7,112 formulae).

We used the functions `lm` and `rlm` from R [1] to learn the linear estimation model. The `lm` function is a basic library function that infers a linear model using the method of least squares. On the other hand, `rlm` (from the R's `MASS` package) implements *robust fitting of linear models*, which uses a modification of the method of least squares that can deal with outliers (see the documentation of `rlm` for more information). We use the output from `rlm` whenever it is available; in some cases (e.g., when the number of data points was too small), the computation of `rlm` did not produce a result, and so we used the output of `lm` (this can happen because `rlm` works in iterations with giving data points different weights; if the computation does not converge in a set number of iterations, the function produces no output). Moreover, we analyzed the learnt models (in particular using the R^2 statistical measure [36]) and discarded those with a low fidelity—this affected models of $\ell_\wedge$ and $\ell_\vee$ with the number of shared variables n for which we did not have enough training data. The discarded models were substituted by a model for a number closest to n that had a high-fidelity. In some cases, we obtained linear models $ax + b$ with a large value of b, which caused a large bias in the computed values,

Table 2: Results of learning

op	n	a	a^∂
$\exists$	—	0.899	0.900
$\wedge$	0	0.666	0.667
$\wedge$	1	0.056	0.275
$\wedge$	2	0.086	0.087
$\vee$	3	0.066	0.073

[6] Two sub-formulae φ and φ' are *similar* if there is an order-preserving renaming of variables of φ such that after the renaming φ becomes identical to φ' [29].

[7] MSO(Str), *monadic second-order logic over strings*, is a dialect of WS1S interpreted over finite strings where a formula describes a regular language. Every MSO(Str) formula can be easily translated into WS1S.

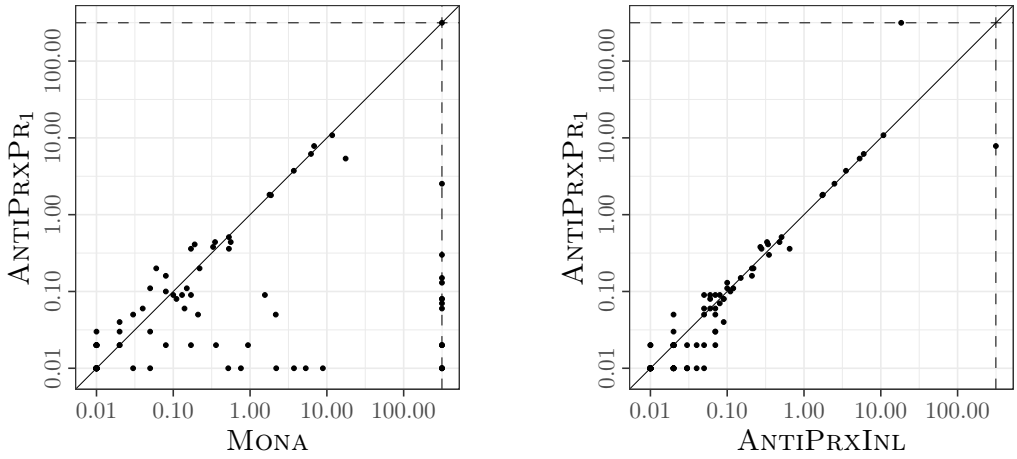


Figure 4: (left) A comparison of the runtime of Mona on unprocessed formulae and on formulae after AntiPrxPr$_1$, (right) a comparison of our two settings of the antiprenexing procedure. The axes are logarithmic and the dashed lines represent the cases where Mona ran out of memory or our antiprenexing did not finish (the timeout for antiprenexing was 300 s).

especially for smaller formulae; in those cases, we modified the model by setting $b = 0$. In Table 2, we provide learnt values of the parameters a (for ℓ_{op}) and a^{∂} (for ℓ^{∂}_{op}) for some cases.

Parameters of Antiprenexing. We experimented with different settings of parameters of our antiprenexing procedure from Section 3 and chose three that give the best overall performance on our benchmark set; their overview is in Table 1.

Results of Experiments. For each formula φ in our benchmark set, we compared the runtime of Mona on φ (denoted as Mona) with the runtime of Mona on the formulae obtained after the antiprenexing transformations (denoted by the corresponding transformation). We do not mention the timeout for Mona because when Mona failed to decide a formula, it was always because it ran out of memory (note that Mona is optimised for 32 bits, therefore it could not use all the available memory). The timeout for antiprenexing was set to 300 s.

In the left-hand plot of Figure 4, we give a comparison of Mona and AntiPrxPr$_1$. Note that there are many cases where antiprenexing significantly shortens the time to decide a formula (the data points at the bottom of the plot) and also many formulae that can be decided only after antiprenexing (the vertical dashed line).

In the right-hand plot of Figure 4, we compared AntiPrxInl with AntiPrxPr$_1$. The plot shows that AntiPrxPr$_1$ more often behaves better. There are, however, two interesting cases of formulae that can only be decided by one of the settings. These are the formulae `sl` (which can be decided by AntiPrxInl in 18.43 s; AntiPrxPr$_1$ and AntiPrxPr$_2$ timeout) and `von-neumann-add` (which can be decided by AntiPrxPr$_1$ in 7.83 s, by AntiPrxPr$_2$ in 5.36 s and by Mona in 6.87 s; AntiPrxInl timeouted). The formula `sl` comes from an experimental translation of separation logic into MSO(Str) [6] (in particular of the property of the existence of a path in a symbolic heap) and the formula `von-neumann-add` encodes the fact that an 8-bit von Neumann adder is equivalent to a standard carry-chain adder [29].

In Table 3, we give a selection of interesting benchmarks where the first half of the table contains formulae from practical scenarios and the second half contains artificially constructed (parameterised) formulae. The column |**DAG**| denotes the size of the DAG obtained by Mona

Table 3: A selection of interesting benchmarks; "–" denotes that MONA ran out of memory (OOM) or that our antiprenexing did not finish (the timeout for antiprenexing was 300 s). The column |**DAG**| gives a measure of the size of the input formula. The time reported does not include preprocessing.

Formula	|**DAG**|	MONA	ANTIPRXINL	ANTIPRXPR$_1$	ANTIPRXPR$_2$	**Source**
`four-weights`	145	0.77	0.03	0.01	0.02	[23]
`smoothing`	221	17.94	5.30	5.38	0.46	[23]
`tree-weights-min`	153	12.00	10.83	10.83	10.50	[23]
`von-neumann-add`	267	6.87	–	7.83	5.36	[29]
`sl`	77	–	18.43	–	–	[6]
`lift_8.ltl0`	255	–	0.35	0.3	–	[48]
`lift_b_7.ltl0`	380	–	0.10	0.13	–	[48]
`horn_sub17`	39	2.10	0.02	0.01	0.01	[20]
`horn_sub18`	41	5.42	0.01	0.01	0.01	[20]
`horn_sub19`	43	–	0.02	0.02	0.01	[20]
Total OOM		18	1	1	5	

from the input formula before further reductions and is used as a measure of the size of the input formula. Notice that antiprenexing can significantly decrease the runtime of MONA (`smoothing`) or be necessary for deciding a formula at all (e.g., the formula `sl` cannot be, to the best of our knowledge, solved by any current automatic tool other than ANTIMONA). In the second half of the table, observe the `horn_subN` family of formulae, which denotes formulae of the form $\exists X.\ \forall Y_1 \ldots \forall Y_{\mathtt{N}}.\ (Y_1 \subseteq X \Rightarrow Y_2 \subseteq X) \wedge \ldots \wedge (Y_{\mathtt{N}-1} \subseteq X \Rightarrow Y_{\mathtt{N}} \subseteq X)$. Note that the increase of `N` makes the formula significantly harder for MONA (for $\mathtt{N} = 19$, MONA cannot handle the formula at all). Antiprenexing seems to mitigate this exponential behaviour of MONA.

We note that our benchmark set does not contain two of the available benchmarks from [18], `lift_b_8.ltl0` and `lift_b_9.ltl0`, because MONA and all of the settings of ANTIMONA presented above timeouted on them (ANTIMONA during antiprenexing). Nonetheless, by slightly tuning the parameters of ANTIPRXINL2 (SIMPLETHRES $= 5,000$), we obtained a setting under which ANTIMONA quickly produced a formula that could be easily decided by MONA.

Discussion. The experimental results obtained from our prototype implementation show that our antiprenexing techniques can significantly reduce the time for deciding WSkS formulae—or allow the formula to be decided at all. The settings we have provided in Table 1 were selected for their ability to give good overall performance on the whole benchmark set, which mixes formulae of varying character. These settings are, however, not universally the best, the optimal settings for particular formulae may vary significantly. Hard formulae may be decided through tailoring the parameters to fit, as is indeed witnessed by the two last formulae mentioned above. Our parametric framework also makes it possible to fine-tune the parameters for a specific *class* of similar formulae, which typically come from specific application domains (such as verification conditions of programs of a certain kind or by translation from some given logic). As a part of our future work, we wish to automate the process of tuning the parameters for a given class of WSkS formulae (possibly using some machine learning approach again).

6 Related Work

The seminal works [11, 35] on the automata-logic connection were the milestones leading to what we call here the classical tree automata-based decision procedure for WSkS [42]. Its

non-elementary worst-case complexity was proved in [40], and the work [22] presents the first implementation, restricted to WS1S, with the ambition to use heuristics to counter the high complexity. The authors of [13] provide an excellent survey of the classical results and literature related to WSkS and tree automata.

The tool Mona [17] 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 (any WSkS formula can be encoded into WS2S). 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.) [29, 27]. The M2L(Str) logic, a dialect of WS1S, can also be decided by a similar automata-based decision procedure, implemented within, e.g., jMosel [43] or the symbolic finite automata framework of [14]. In particular, jMosel implements several optimizations (such as second-order value numbering [32]) that allow it to outperform Mona on some benchmarks (Mona also provides an M2L(Str) interface on top of the WS1S decision procedure).

Recently, several works on lazy approaches to the automata decision procedure appeared, namely our works [24, 19, 20] and a similarly focused work [44]. The idea behind these approaches is that, instead of constructing automata explicitly, they work with an implicit transition relation, which is defined inductively to the structure of a formula. These works allow for efficient on-the-fly pruning of the state space inspired by the automata antichain algorithms such as [15, 47, 9, 2] and for an efficient use of early termination, allowing entirely skipping large irrelevant portions of the state space of automata. The lazy approaches, however, require heavier data structures and are incompatible with automata minimization. Another alternative to the classical procedure was proposed by Ganzow and Kaiser [21] who developed a decision procedure for the weak monadic second-order logic on inductive structures (a generalization of WSkS) within their tool Toss. Their approach completely avoids automata; instead, it is based on Shelah's composition method.

Implementations of these alternative methods outperform Mona on certain classes of formulae. In practice, however, Mona is still substantially most robust by a large margin (partially owing to the relative immaturity of the alternative tools). Variants of our antiprenexing techniques would certainly be relevant for the lazy automata approaches [24, 19, 20, 44]. In [19, 20], we have actually successfully used a simple variant of antiprenexing (namely the quantifier distribution rule (QuantDistr)). Our more advanced techniques that use other rules according to a cost estimate cannot, however, be used directly. One would have to come up with different strategies and cost estimation techniques specific to these algorithms. For this reason, and also because most of the formulae from our benchmark are beyond the reach of other tools than Mona, we do not consider them in our experimental evaluation.

Basic principles of the transformation to antiprenex (or miniscope) form are a well-known folklore in theorem proving, QBF, and SMT solving. Its values were recognised, for instance, in [16, 7, 3, 37], and its origins reach at least to [45].

Acknowledgement

We thank the anonymous reviewers for their helpful comments on how to improve the exposition in this paper. The work on this paper was supported by the Czech Science Foundation project 20-07487S, the FIT BUT internal project FIT-S-20-6427, and the Czech Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science—LQ1602, and the ERC.CZ project LL1908.

References

[1] The R project for statistical computing, 2020. https://www.r-project.org/.

[2] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 158–174. Springer, 2010.

[3] Matthias Baaz and Alexander Leitsch. On Skolemization and proof complexity. *Fundam. Inform.*, 20(4):353–379, 1994.

[4] David Basin and Nils Klarlund. Automata based symbolic reasoning in hardware verification. In *Proc. of CAV'98*, LNCS, pages 349–361. Springer, 1998.

[5] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In *Proc. of TACAS'00*, volume 1785 of *LNCS*, pages 188–203. Springer, 2000.

[6] Josh Berdine. Private communication, 2015.

[7] Wolfgang Bibel. An approach to a systematic theorem proving procedure in first-order logic. *Computing*, 12(1):43–55, 1974.

[8] Jean-Paul Bodeveix and Mamoun Filali. FMona: A tool for expressing validation techniques over infinite state systems. In *Proc. of TACAS'00*, volume 1785 of *LNCS*, pages 204–219. Springer, 2000.

[9] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *Proc. of CIAA'08*, volume 5148 of *LNCS*, pages 57–67. Springer, 2008.

[10] Marius Bozga, Radu Iosif, and Joseph Sifakis. Structural invariants for parametric verification of systems with almost linear architectures. Technical Report arXiv:1902.02696, 2019.

[11] Julius R. Büchi. On a decision method in restricted second-order arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[12] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[13] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008.

[14] Loris D'Antoni and Margus Veanes. Monadic second-order logic on finite sequences. In *Proc. of POPL'17*, pages 232–245. ACM, 2017.

[15] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 2–22. Springer, 2010.

[16] Uwe Egly. On the value of antiprenexing. In *Proc. of LPAR'94*, volume 822 of *LNCS*, pages 69–83. Springer, 1994.

[17] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: New techniques for WS1S and WS2S. In *Proc. of CAV'98*, volume 1427 of *LNCS*, pages 516–520. Springer, 1998.

[18] Loris D'Antoni et al. AutomatArk: LTL-finite (M2L-Str), 2018. https://github.com/lorisdanto/automatark/tree/master/m2l-str/LTL-finite.

[19] Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. Lazy automata techniques for WS1S. In *Proc. of TACAS'17*, volume 10205 of *LNCS*, pages 407–425. Springer, 2017.

[20] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Nested antichains for WS1S. *Acta Inf.*, 56(3):205–228, 2019.

[21] Tobias Ganzow and Lukasz Kaiser. New algorithm for weak monadic second-order logic on inductive structures. In *Proc. of CSL'10*, volume 6247 of *LNCS*, pages 366–380. Springer, 2010.

[22] James Glenn and William Gasarch. Implementing WS1S via finite automata. In *Workshop on Implementing Automata*, volume 1260 of *LNCS*, pages 50–63. Springer, 1996.

[23] Jad Hamza, Barbara Jobstmann, and Viktor Kuncak. Synthesis for regular specifications over unbounded domains. In *Proc. of FMCAD'10*, pages 101–109. IEEE, 2010.

[24] Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Automata terms in a lazy WSkS decision procedure. In *Proc. of CADE-27*, volume 11716 of *LNCS*, pages 300–318. Springer, 2019.

[25] Thomas Hune and Anders Sandholm. A case study on using automata in control synthesis. In *Proc. of FASE'00*, volume 1783 of *LNCS*, pages 349–362. Springer, 2000.

[26] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in automated verification based on trace abstractions. In *Formal System Specification, The RPC-Memory Specification Case Study*, volume 1169 of *LNCS*. Springer, 1996.

[27] Nils Klarlund. A theory of restrictions for logics and automata. In *Proc. of CAV'99*, volume 1633 of *LNCS*, pages 406–417. Springer, 1999.

[28] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from http://www.brics.dk/mona/. Revision of BRICS NS-98-3.

[29] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[30] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In *Proc. of POPL'11*, pages 611–622. ACM, 2011.

[31] P. Madhusudan and Xiaokang Qiu. Efficient decision procedures for heaps using STRAND. In *Proc. of SAS'11*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.

[32] Tiziana Margaria, Bernhard Steffen, and Christian Topnik. Second-order value numbering. In *Proc. of GraMoT'10*, volume 30 of *ECEASST*, pages 1–15. EASST, 2010.

[33] A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *Proc. of PLDI'01*. ACM, 2001.

[34] F. Morawietz and T. Cornell. The logic-automaton connection in linguistics. In *Proc. of LACL'97*, volume 1582 of *LNAI*. Springer, 1997.

[35] Michael O. Rabin. Decidability of second order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[36] John O. Rawlings, Sastry G. Pantula, and David A. Dickey. *Applied Regression Analysis: A Research Tool*. Springer-Verlag New York, New York, 2 edition, 1998.

[37] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[38] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for web services. In *Proc. of FASE'98*, volume 1382 of *LNCS*, pages 270–284. Springer, 1998.

[39] Mark A. Smith and Nils Klarlund. Verification of a sliding window protocol using IOA and MONA. In *Proc. of FORTE/PSTV'00*, volume 183 of *IFIP*, pages 19–34. Kluwer, 2000.

[40] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proc. of STOC'73*, pages 1–9, New York, NY, USA, 1973. ACM.

[41] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Comput. Log.*, 22(4):33:1–33:33, 2013.

[42] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory*, 2(1):57–81, 1968.

[43] Christian Topnik, Eva Wilhelm, Tiziana Margaria, and Bernhard Steffen. jMosel: A stand-alone tool and jABC plugin for M2L(Str). In *Proc. of SPIN'06*, volume 3925 of *LNCS*, pages 293–298. Springer, 2006.

[44] Dmitriy Traytel. A coalgebraic decision procedure for WS1S. In *Proc. of CSL'15*, volume 41 of *LIPIcs*, pages 487–503, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Infor-

matik.

[45] H. Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1):2–22, Jan 1960.

[46] Thomas Wies, Marco Muñiz, and Viktor Kuncak. An efficient decision procedure for imperative tree data structures. In *Proc. of CADE-23*, volume 6803 of *LNCS*, pages 476–491. Springer, 2011.

[47] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.

[48] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 63–77, 2008.

[49] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *Proc. of POPL'08*, pages 349–361. ACM, 2008.

[50] Min Zhou, Fei He, Bow-Yaw Wang, Ming Gu, and Jiaguang Sun. Array theory of bounded elements and its applications. *J. Autom. Reasoning*, 52(4):379–405, 2014.